Efficient Architecture-Level Configuration of Large-Scale Embedded Software Systems

Razieh Behjati, Shiva Nejati

Efficient Architecture-Level Configuration of Large-Scale Embedded Software Systems

Razieh Behjati[1] and Shiva Nejati[2]
[1]Certus Software V&V Center, Simula Research Laboratory, Norway
[2]SnT Centre, University of Luxembourg, Luxembourg
behjati@simula.no, shiva.nejati@uni.lu

Abstract. Configuration is a recurring problem in many domains. In our earlier work, we focused on architecture-level configuration of large-scale embedded software systems and proposed a methodology that enables engineers to configure products by instantiating a given reference architecture model. Products have to satisfy a number of constraints specified in the reference architecture model. If not, the engineers have to backtrack their configuration decisions to rebuild a configured product that satisfies the constraints. Backtracking configuration decisions makes the configuration process considerably slow. In this paper, we improve our earlier work and propose a backtrack-free configuration mechanism. Specifically, given a cycle-free generic reference architecture model, we propose an algorithm that computes an ordering over configuration parameters that yields a consistent configuration without any need to backtrack. We evaluated our approach on a simplified model of an industrial case study. We show that our ordering approach eliminates backtracking. It reduces the overall configuration time by both reducing the required number of value assignments, and reducing the time that it takes to complete one configuration iteration. Furthermore, we show that the latter has a linear growth with the size of the configuration problem.
Keywords: Model-based configuration, CSP, Backtracking, UML/OCL.

1 Introduction

Configuration is a recurring problem in many embedded software system domains such as energy, automotive, and avionics. In these domains, product-line engineering approaches [27, 22] are largely applied to develop various configurations of a *reference architecture*. Briefly, a reference architecture provides a common, high-level, and customizable structure for all members of the product family [27] by specifying different types of components and configurable parameters, as well as, constraints capturing relationships between these parameters. Through configuration, engineers develop each product by creating component instances and assigning values to their parameters such that the constraints over parameters are satisfied.

Normally, configuring embedded systems involves assigning values to tens of thousands of interdependent parameters, and ensuring their consistency. Typically, 15 to 20 percent of these parameters are interdependent. Finding consistent values for interdependent parameters without any automated support is

challenging. Manual configuration – the common practice in many companies – is time-consuming and error-prone, especially for large-scale systems. During the last three decades, researchers have developed a large number of approaches to automate various configuration use cases [6]. Most of these approaches concern consistency of configuration decisions, and rely on constraint solvers (e.g., [13, 23, 19]) or SAT solvers (e.g., [20]) for ensuring consistency.

In our earlier work [4], we proposed an iterative approach for configuring large embedded systems where at each iteration the value for one parameter is specified. If at some point during configuration, a value assignment violates some constraints, then the engineers may have to *backtrack* some of their recent choices until they can find a configuration assignment consistent with the constraints in the reference architecture. Backtracking configuration decisions makes the configuration process considerably expensive.

In this paper, we extend our earlier work [4] and propose a new approach that eliminates backtracking during configuration by configuring parameters in a certain order. We explain how such an ordering is extracted from an acyclic[1] reference architecture model, and argue that if the ordering is followed, our algorithm generates consistent and complete configured products without any need to backtrack a decision. We argue that elimination of backtracking considerably improves the performance of our configuration approach. We show this by applying our approach to a simplified excerpt of an industrial case study from the oil and gas domain. The experiment shows that our ordering approach reduces the overall configuration time by both reducing the required number of value assignments, and reducing the time that it takes to complete one configuration iteration. Further, we demonstrate that in our backtrack-free configuration approach the time required for completing one configuration iteration grows linearly with the size of the configuration problem. In our original configuration approach, this time has a quadratic growth.

In the rest of the paper, we first present the related work and position our work in the literature. Section 3 provides an overview of the main concepts in product family modeling and configuration. Our ordering approach for eliminating backtracking, and the resulting backtrack-free configuration algorithm are presented in Section 4. In Section 5, we experimentally evaluate the efficiency and scalability of our approach. A discussion of the potentials and limitations of the work is presented in Section 6. Finally, we conclude the work in Section 7.

2 Related Work

Existing configuration approaches fall into two general categories, *non-interactive* and *interactive*. Most configuration approaches belong to the first category,

[1] In our approach, a reference architecture model consists of a component hierarchy, and a set of constraints (see Section 3.1). Such a reference architecture is acyclic if it neither contains any cycles in the component hierarchy nor in the constraints. This condition is required to ensure the termination of the configuration process as well as the complete elimination of backtracking (see Sections 4 and 6).

where the objective is to produce some final configured products without requiring intermediate input from users. They may either find an optimized solution based on some given optimization criteria (e.g., [13, 19]) or find all configuration solutions (e.g., [7, 11, 23]). The non-interactive approaches may either rely on meta-heuristic search approaches [14, 16, 21], or on systematic search techniques used in constraint solvers [9, 10, 18], or on symbolic decision procedures [8]. Among these, meta-heuristic search approaches are generally faster and require less memory. However, since meta-heuristic search is stochastic and incomplete, it cannot support an interactive process where engineers have to be provided with precise and complete guidance information at each iteration.

Interactive configuration methods (e.g., [17, 20, 30, 31]) mostly rely on constraint solvers or symbolic reasoning approaches. Backtracking is required whenever an inconsistency arises, even though it may make the process considerably slower. In general, constraint solvers alleviate the drawbacks of backtracking by employing heuristics such as back-jumping [12], identifying no-goods constraints [1, 2], and ordering the search [15]. None of these improvements, however, totally eliminates the possibility of backtracking. In addition, it is open whether these heuristics can be tailored to interactive configuration solutions.

Some more recent interactive configuration approaches [17, 30] have eliminated backtracking by adding an offline preprocessing phase to configuration, during which all consistent configurations are computed and used to direct the user during the interactive phase, preventing the user to make any decision that gives rise to an inconsistency. These approaches only scale when the space of all consistent configurations can be encoded and computed within the available memory. In the case of large-scale embedded software systems, the complexity of constraints and the size of the configuration space is so large[2], making it impossible to compute the set of all possible configurations in an offline mode.

In our work, using information provided in reference architecture model, we identify an ordering over variables and show that by following this ordering, backtracking does not arise during the configuration of a single product. Our approach applies to architecture-level configuration of embedded software systems with architectural dependencies and constraints specified in First-Order Logic (FOL) [29]. Computation of ordering in our work is fast and performed based on static analysis of architectural models and the constraints syntax.

3 Preliminaries

The work presented in this paper is based on a model-based configuration framework presented in [4]. In this section, we present the reference architecture model, and exemplify the main concepts in modeling and configuration of embedded software systems. In addition, we propose the notion of a configuration tree.

[2] Creating a product usually involves configuring tens of thousands of parameters. The configuration space, which is in fact the combinatorial space created for these parameters is, as well, significantly large.

3.1 The reference architecture model

In our approach, a reference architecture model defines a hierarchy of component types. Each component has a number of configurable parameters, and may as well contain other *configurable elements*. We consider configurable parameters as one type of configurable elements. Furthermore, the reference architecture model specifies constraints among the configurable parameters. The SimPL methodology [5] is an approach for creating such reference architecture models. As mentioned earlier, during configuration, products are created by creating component instances and configuring their parameters. In the following, we explain components as the main building blocks of products, and constraints as they play a key role is ensuring the consistency of products. A complete specification of the reference architecture model and its formal semantics is given in [4].

3.1.1 Components

Each component in a product is an instance of a component type in the corresponding reference architecture. We define a component c as a tuple $(id, \mathcal{V})$, where id is a unique identifier, and $\mathcal{V}$ is a set of configurable elements. Each configurable element in $\mathcal{V}$ is a tuple $e = (id_e, t_e)$, where id_e is the name of the element, and t_e is the type of the element. Figure 1 gives a grammar for the types of elements in $\mathcal{V}$.

```
type             ::= single_type | arrayed_type ;
single_type      ::= primitive_type | user_defined_type |
                     referenced_type ;
referenced_type  ::= '&' user_defined_type ;
arrayed_type     ::= single_type '[]' ;
```

Fig. 1. A simplified grammar for types.

In Figure 1, primitive_type represents a set of terminals for primitive types 'integer' and 'boolean' (we do not consider 'strings'), and user_defined_type denotes a set of terminals each corresponding to a component type defined in the reference architecture model.

Configurable elements of a primitive type or a referenced type represent configurable parameters. A configurable element of a user defined type represents a sub-component of c (i.e., the sub-component is a component $c' = (id', \mathcal{V}')$ itself).

3.1.2 Constraints

Let $c = (id, \mathcal{V})$ be a component. We use Φ_{id} to denote the set of constraints defined in the context of c (i.e., specifying relations among elements of c). Each member of the set Φ_{id} is a boolean expression denoting a constraint ϕ. A simplified grammar for the language of boolean expressions is given in Figure 2. This grammar maps to a subset of the Object Constraint Language (OCL) [26] that we use in the SimPL methodology. The grammar allows the basic OCL

operators including for-all, exists, arithmetic, relational and logical operators. In Figure 2, FA represents the universal quantifier, which maps to OCL forAll operator. Similarly, EX represents the existential quantifier, which maps to OCL exists operator.

```
bool_expr   ::= bool_term (OR bool_term)*;
bool_term   ::= bool_factor (AND bool_factor)*;
bool_factor ::= bool_literal | bool_qName | var |
                '(' bool_expr ')' | rel_expr | NOT bool_factor |
                FA '(' var 'in' array_qName ',' bool_expr ')' |
                EX '(' var 'in' array_qName ',' bool_expr ')' ;
rel_expr    ::= num_expr (GT | LT | GEQ | LEQ | EQ | NEQ) num_expr;
num_expr    ::= num_term ((PLUS | MINUS) num_term)*;
num_term    ::= num_factor ((MUL | DIV) num_factor)*;
num_factor  ::= num_literal | int_qName | var |
                '(' num_expr ')' | NEG num_factor;
```

Fig. 2. A simplified grammar of boolean formulas.

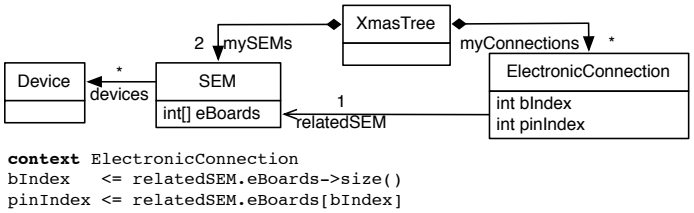
Three types of qualified names (i.e., bool_qName, int_qName, and array_qName) are used in the production rules of the grammar given in Figure 2. Qualified names together with literals and operators create numerical, relational, and boolean expressions. Qualified names of numerical types (i.e., integer or a user defined enumeration) form one type of numerical factors and are used in creating relational expressions. Qualified names of type boolean form one type of boolean factors. Qualified names representing collections of items can be combined with set quantifiers (i.e., for all and exists) to form another group of boolean factors. In addition to these, variables (i.e., var) may be used as integer or boolean factors. Variables are used in combination with quantifiers.

3.1.3 A configuration example

Figure 3 is a class diagram showing an excerpt of a simplified reference architecture for a family of subsea oil production systems. It is part of a larger case study, which is presented in [5, 4]. Each class in Figure 3 represents a component type, and each attribute in a class represents a configurable parameter. In addition, two OCL constraints are defined in the context of class ElectronicConnections.

To make a product, one has to create and configure an instance of a XmasTree. To do so, engineers have to specify the number of electronic boards on each of the Subsea Electronic Modules[3] (SEMs) by initializing the eBoards array in each of the two SEMs (each XmasTree instance has two SEM instances) and assign a value to each item in those arrays, create a number of electronic connections by setting the size of array myConnections, and assigning values to relatedSEM, pinIndex and bIndex attributes of each ElectronicConnection instance.

[3] A Subsea Electronic Module is an electronic unit, with software deployed on it. It is the main component in a subsea control system.




```
context ElectronicConnection
bIndex   <= relatedSEM.eBoards->size()
pinIndex <= relatedSEM.eBoards[bIndex]
```

Fig. 3. An excerpt of the reference architecture model of a subsea oil production system.

Example 1. Suppose that, at some point in the configuration of a product, a user configures an ElectronicConnection by first setting its bIndex to 5, then setting its pinIndex to 20, and finally setting its relatedSEM to one of the SEM instances. At this point, if the chosen SEM instance has less than 5 electronic boards, or its eBoards[5] is less than 20, an inconsistency happens. In this case, the user has to backtrack to fix the inconsistency, for example by changing the value of pinIndex or bIndex. Alternatively, to eliminate backtracking, the user can first configure relatedSEM, then assign a value to bIndex, and finally configure pinIndex. ■

In this paper, based on a static analysis of the constraints, we propose an approach for identifying configuration orderings that eliminate backtracking.

3.2 The configuration tree

A product is usually represented by a *configuration tree*. We denote a configuration tree by a tuple (N, E), where N is the set of nodes, and E is the set of edges of the tree. In our approach, each node in a configuration tree has a type and a value, and each edge has a label. The type of a node belongs to the language of types in Figure 1. Based on this, we identify four types of nodes: *primitive nodes*, *component nodes* (if the node is typed by a user defined type), *reference nodes*, and *array nodes*. Figure 4 shows two example configuration trees.

Primitive nodes and reference nodes represent configurable parameters, and are always leaf nodes in the tree. The value of a leaf node must conform to its type. A missing value for a leaf node means that the corresponding configurable parameter is not yet configured. Nodes m13 and m15 in Figure 4-(b) are primitive and reference nodes, respectively. Both nodes are unconfigured.

Each array node has a child node of type 'int', which is connected to it by an edge labeled 'size()'. We refer to this node as the array's size node. An array node is called *uninitialized* if its size node does not have a value, and is called *initialized* otherwise. An initialized array node of size n, and type '$single\_type$[]' has n additional child nodes. Each of these child nodes is typed by '$single\_type$', and is connected to the array node via an edge labeled '$at(i)$', where i is an integer in $[1..n]$. Node m2 in Figure 4-(a) is an initialized array node of size two.

A component node represents a component. Such a node is typed by a user defined type, and its value is the identifier of the corresponding component. Let

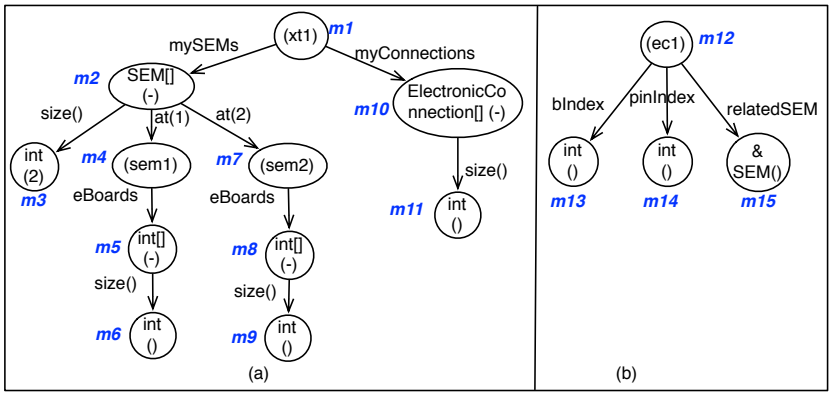


Fig. 4. Configuration subtrees representing two components. Text inside a circle represents the type and value (in parenthesis) of the node. The text next to a node is a unique name to refer to the node in our explanation of the approach. (a) an instance of XmasTree, and (b) an instance of ElectronicConnection.

m be the component node representing the component $c = (id, \mathcal{V})$. For each $(id_e, t_e) \in \mathcal{V}$, there is a child node for m typed by t_e and connected to m via an edge labeled id_e. Node m1 in Figure 4-(a) is a component node of type XmasTree, representing a component with identifier xt1. To avoid cluttering, we have not shown the type of the component (i.e., XmasTree) in the text inside node m1. The subtree beneath m1 shows the configurable elements of xt1. None of these elements are configured in Figure 4-(a). One possible partial configuration is given in Figure 5.

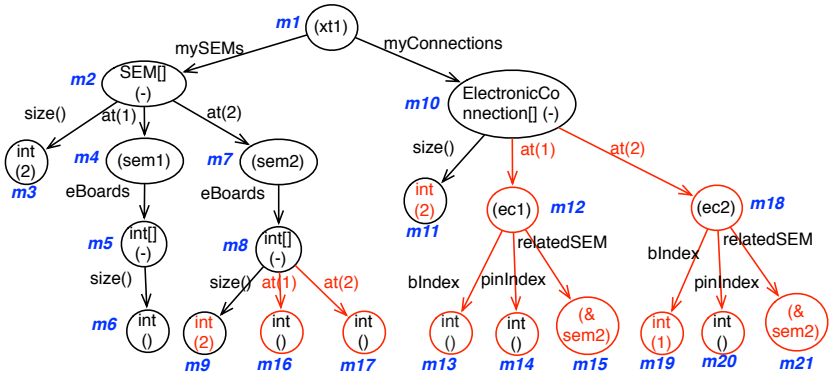


Fig. 5. One possible partial configuration of node *m1* in Figure 4-(a).

4 The configuration process

Our model-based configuration approach presented in [4] validates configuration decisions automatically and interactively. For this purpose, we use the config-

uration tree and the constraints defined in the reference architecture model to create a constraint network [24]. A constraint network is a finite set of variables, each associated with a finite domain of discrete values, and a number of constraints over those variables. The problem of finding a consistent configuration maps to a constraint satisfaction problem, where the objective is to find a consistent assignment of values to all the variables in the constraint network. Each configurable parameter (a leaf node in the configuration tree) maps to a variable in the constraint network. The domain of the variable corresponding to configurable parameter p is a finite set of literals that can be assigned to p. Each constraint in the reference architecture model is rewritten in terms of the variables in the constraint network, and is added to the constraint network. During configuration, new variables or constraints may be added to the constraint network. We call this the *dynamic growth* of the constraint network.

To ensure the consistency of configuration decisions, we use constraint propagation over finite domains [28]. Constraint propagation provides a sound approximation of consistency: it does not eliminate any consistent solution, but it may fail to identify all inconsistent value-assignments. In other words, constraint propagation prunes the search space, but it does not enumerate all possible solutions. The benefit of using constraint propagation is that it is fast, and therefore applicable in an interactive context. Its drawback is that it does not eliminate all inconsistent value-assignments, and therefore, backtracking may be needed to ensure consistency. This can be avoided by imposing some restrictions on the reference architecture model of the product family. In particular, the model of the product family should not contain any cyclic constraints.

Another reason for requiring backtracking in our original configuration approach is the dynamic growth of the constraint network. New constraints that are added to the constraint network may be inconsistent with some of the previously made decisions. To avoid this, we configure parameters in a particular order. In the following, before presenting our approach for ordering configuration decisions, we first present the notion of qualified names. Then, based on the proposed ordering approach, we present a backtrack-free configuration algorithm.

4.1 Qualified names

Figure 6 shows a grammar for *qualified names*. A qualified name (e.g., int_qName) represents a typed variable (e.g., a configurable parameter) and may represent an individual item (e.g., int_qName) or a collection of items (i.e., array_qName). The last rule in Figure 6 is added to explicitly define int_qName and bool_qName as *primitive qualified names*. Primitive qualified names represent configurable parameters, and together with array qualified names are used in the grammar of boolean expressions in Figure 2.

A qualified name can be created by traversing a configuration tree. Let CT be a configuration tree, and n be a node representing a component $c = (id, \mathcal{V})$ in the configuration tree. Each node n' in the subtree rooted at n can be uniquely identified by a string created using id and edge labels. To do so, we start with

string $str =$ "*id*", and follow the edges that bring us to n'. After traversing each edge, we concatenate str with "$.l$", where l is the label of the last traversed edge[4]. Using this approach each node in the tree may be represented by more than one string, depending on the starting node. A string should always start with the label of a component node.

```
1  int_qName                ::= element_qName '.' int_prop_name |
2                               array_qName '.' 'size()' |
3                               int_array_qName '[' int_factor ']';
4  int_factor               ::= int_literal | int_qName;
5  int_array_qName          ::= element_qName '.' int_array_prop_name;
6  element_qName            ::= component_id |
7                               element_qName '.' element_prop_name |
8                               element_array_qName '[' int_factor ']';
9  element_array_qName      ::= element_qName '.' element_array_prop_name;
10 bool_qName               ::= element_qName '.' bool_prop_name |
11                              bool_array_qName '[' int_factor ']';
12 bool_array_qName         ::= element_qName '.' bool_array_prop_name;
13 array_qName              ::= int_array_qName | bool_array_qName |
14                              element_array_qName;
15 primitive_qName          ::= int_qName | bool_qName;
```

Fig. 6. The grammar of qualified names.

4.1.1 Semantically valid qualified names

Let CT be a configuration tree representing a possibly partially-configured product derived from a given reference architecture. A subset of the qualified names created using the grammar in Figure 6 are semantically valid with respect to the configuration tree CT. We use $Q(CT)$ to denote this subset. A qualified name q belongs to $Q(CT)$ iff one of the following holds:

- q is the label of a component node in CT,
- $q = q1.t$, where $q1 \in Q(CT)$, and $q1$ represents a component $c = (id, \mathcal{V})$, such that t is the name of an element in $\mathcal{V}$,
- $q = q1[q2]$, where $q1 \in Q(CT)$, $q1$ represents an arrayed element, and $q2$ is either an integer literal or a semantically valid qualified name representing an integer parameter,
- $q = q1.size()$, where $q1 \in Q(CT)$, and $q1$ is an arrayed element.

4.1.2 Mapped and unmapped qualified names

Let CT be a configuration tree, and q be a semantically valid qualified name in $Q(CT)$. If q corresponds to a node in CT, then we call q a *mapped* qualified name, otherwise, it is called an *unmapped* qualified name. A qualified name q is unmapped if any of the following conditions holds:

[4] In the rest of this paper, for the sake of conciseness, we use $a[i]$ to denote $a.at(i)$, where a represents an array node in the configuration tree, and i is an integer literal.

– a prefix of q maps to an unconfigured reference node,
– q is of the form $q1[q2]$, where $q2$ represents an unconfigured parameter,
– q is of the form $q1[q2]$, where $q1$ is an uninitialized array ($q1.size()$ is not configured).

In each case, a parameter is unconfigured. For a semantically valid unmapped qualified name q, we use $\mathcal{U}(q)$ to denote the set of all such unconfigured parameters. For a qualified name q mapped to a leaf node, we use $\mathcal{M}(q)$ to denote the corresponding configurable parameter.

Let CT be the configuration tree in Figure 5. Then xt1.mySEMs[1] is mapped, and sem1.eBoards[1] is an unmapped semantically valid qualified name in $Q(CT)$. An unmapped qualified name can become mapped as parameters are configured and the tree is expanded. For example, sem1.eBoards[1] becomes mapped after configuring the size of sem1.eBoards.

4.2 Ordering configuration decisions

Example 1 in Section 3.1.3 shows an example of inconsistencies that arise due to the dynamic growth. In this example, the two constraints in Figure 3 cannot be evaluated until relatedSEM is configured. This is because relatedSEM.eBoards, appeared in both constraints, is unmapped as it does not correspond to a unique node in the tree. By configuring relatedSEM both constraints become *ready-to-evaluate*, can be added to the constraint network, and can be used in constraint propagation to validate the values assigned to pinIndex and bIndex or to eliminate inconsistent values for them if they were not configured. We call a constraint that is not yet ready-to-evaluate, a *pending* constraint. Such a constraint contains one or more unmapped qualified names and is pending on one or more parameters to be configured. These parameters should be configured to make the unmapped qualified names mapped. For example, the constraints in *Example 1* are pending on relatedSEM to be configured. In each configuration iteration, each constraint is either pending or ready-to-evaluate. Figure 7 shows the state transition diagram of a constraint. As a consequence of configuring parameters, a pending constraint may become ready-to-evaluate. Only ready-to-evaluate constraints can be included in the constraint network.

Consider the ith step of configuration and let c be a pending binary constraint, containing two qualified names q_1 and q_2. Suppose that q_1 is unmapped, and q_2 is mapped to the configurable parameter p (i.e., $\mathcal{M}(q_2) = p$). This parameter cannot be configured until c becomes ready-to-evaluate (i.e., until q_1 becomes a mapped qualified name). We refer to such a parameter as a *pending* parameter. Let parameters $p_1, ..., p_n$ be the parameters that should be configured to make q_1 a mapped qualified name (i.e., $\mathcal{U}(q_1) = \{p_1, ..., p_n\}$). To eliminate backtracking, parameter p should be configured after all p_is are configured. This is shown in Figure 8. Before a parameter reaches the state ready, the set X, which is the set of all p_is as described above, should be empty. As shown in Figure 8, a parameter can be configured only when it is in state ready. Note that, as suggested by the formulation of X, in general p may be involved in more than one constraint.

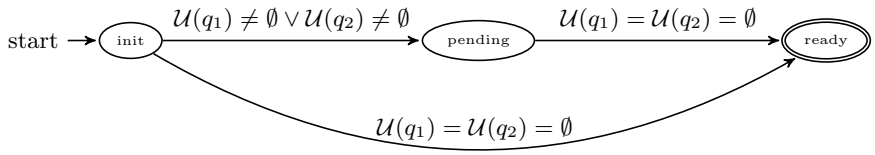


Fig. 7. States of constraint $c = \phi(q_1, q_2)$.

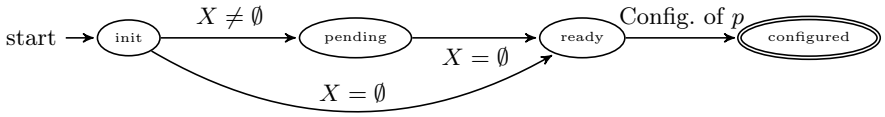


Fig. 8. States of parameter p. X is $\{p' | \exists c = \phi(q_1, q_2).p' \in \mathcal{U}(q_1) \wedge \mathcal{M}(q_2) = p\}$.

In other words, in each configuration iteration, the set of all unconfigured parameters is partitioned into two sets: pending and ready-to-configure parameters. This partitioning of parameters, together with the stepwise configuration, which in each iteration may add new unconfigured parameters to the system, imposes an ordering on the configuration decisions. Note that, in each iteration, there is no ordering among the ready-to-configure parameters. The acyclic property of the reference architecture model guarantees that every parameter eventually reaches the state ready.

4.3 Backtrack-free configuration

Algorithm 1 is our backtrack free configuration algorithm, which implements the ordering approach explained above. Input to the algorithm is a cycle-free reference architecture, which contains a class diagram and a set of constraints. The output is a configuration tree CT. We maintain three sets of parameters: configured (C), ready-to-configure (R), and pending parameters (P); and two sets of constraints: ready-to-evaluate (Φ), and pending (Φ') constraints. Using the input reference architecture model, we initialize all these sets and the configuration tree in line 1 of the algorithm.

Algorithm 1. BTFreeConfig

Input: A reference architecture RA
Output: a configuration tree CT

```
1  (CT, C, R, P, Φ, Φ') ← initializeConfigurationProblem(RA)
2  D ← computeValidDomains(C, R, Φ)
3  while R ≠ ∅ do
4     read(i) ▷ i: index of the selected unconfigured parameter
5     read(v) ▷ value to be assigned to the selected parameter
6     ▷ v must be in D[i] (the domain of the selected parameter)
7     while not v ∈ D[i] do
8        read(v)
9     applyConfiguration(CT, C, R, P, Φ, Φ', i, tmp)
10    D ← updateValidDomains(C, R, Φ)
11    if some domains in D are empty then
12       throwException()
13 return CT
```

In line 2, domains of the unconfigured parameters are computed using the routine computeValidDomains. In this routine, we use a constraint solver to prune the domains by removing values that are inconsistent with one or more constraints in Φ or some values in C. Note that only the ready-to-evaluate constraints and their variables are considered when pruning the domains (i.e., P and Q are not included in the computation).

The while loop in lines 3-12 repeats while there are some ready-to-configure parameters (i.e., $R \neq \emptyset$). In each iteration, one parameter is configured. Both the parameter and its value are selected by the human user in lines 4 and 5 of the algorithm. Lines 7 and 8 guarantee that the selected value is within the domain of the selected parameter and is, therefore, consistent. As a result of assigning a value v to a parameter $R[i]$, one or more of the following may happen:

- new nodes may be added to the configuration tree, therefore new elements may be added to R and P
- if a constraint is pending on $R[i]$, it may become ready-to-evaluate, and
- some of the parameters that are pending on $R[i]$ may become ready for configuration. We move them to the set of ready-to-configure parameters R.

These actions are performed in line 9 by calling the routine applyConfiguration. The constraint solver is again invoked in line 10 to update the valid domains. If some domains become empty, the algorithm throws an exception in line 12. Otherwise, it continues to the next iteration. Eventually, a completely and consistently configured configuration tree is returned in line 13.

In [4], we showed that our original configuration algorithm produces complete and consistent products. A product is complete if it does not contain any unconfigured parameters, and is consistent if it satisfies all the constraints in the reference architecture model. In our technical report [3], we have shown that Algorithm 1 produces complete and consistent products, but without requiring backtracking. In other words, for any given acyclic reference architecture model, Algorithm 1 terminates without ever reaching line 12.

5 Evaluation

To evaluate the efficiency brought by our ordering approach, we performed an experiment using the reference architecture model presented in Figure 3. For this purpose, 1600 random configuration scenarios were created (800 scenarios for each of our original and backtrack-free configuration approaches). In each case, we started by configuring three parameters that identify the size of the configuration problem, then randomly configured the rest of the parameters. The first three parameters were configured as listed in Table 1. This is done merely to control the number of parameters in each case to simplify the analysis of the results. For each case in Table 1, we randomly generated 100 sample configuration scenarios using our original configuration approach, and 100 sample configuration scenarios using the backtrack-free configuration approach. Figure 9 shows the average response time in each iteration for both cases.

# Parameters	# Elec. Connections	# eBoards on SEMs
25	5	3,4
50	14	2,3
75	21	4,5
100	27	6,10
125	32	12,14
150	35	17,25
175	49	11,14
200	49	25,25

Table 1. Configuration settings.

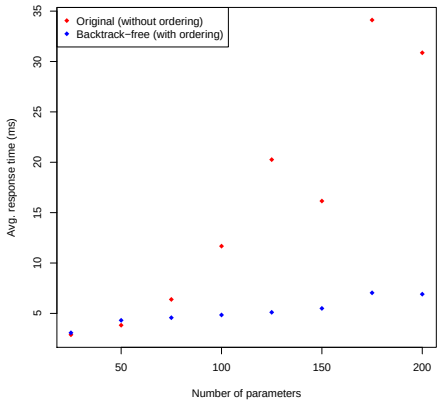


Fig. 9. A comparison of the average response time for our original and backtrack-free configuration approaches.

In our original configuration approach, a complete configuration iteration, in addition to validating the decision and propagating it, may involve several decision roll-backs, and is therefore time consuming. On the other hand, a complete configuration iteration in the backtrack-free configuration approach involves validating and propagating the decision, and updating the ordering (i.e., updating lists of pending constraints and parameters). By using the ordering, we eliminate all the roll-backs and their costs. This explains why for most cases in Figure 9, an average iteration in the original configuration approach takes much longer than that in the backtrack-free configuration approach. This experiment shows that for configuration scenarios with more than 50 parameters, the time overhead of computing the ordering is negligible compared to the time that should otherwise be spent on rolling-back the decisions.

In our original configuration approach, for fixing an inconsistency, in addition to rolling back some of the decisions, new values must be assigned to the parameters that might have caused that inconsistency. In our experiment, on average 23.6 different values were tried per parameter to achieve a consistent configuration. This high number is a result of our current naive implementation of backtracking. By exploiting heuristics such as back jumping [12], this number can be reduced significantly. Table 2 shows the number of decisions that were needed to achieve a consistent configuration.

# Parameters	25	50	75	100	125	150	175	200
Avg. # Decisions	174.85	949.60	1773.49	2678.15	3426.29	2112.34	6301.47	3842.10
Avg. Ratio	6.99	18.99	23.65	26.78	27.41	14.08	36.00	19.21
Total Avg. Ratio (average number of decisions per parameter)							23.62	

Table 2. Overhead of backtracking.

To provide a better insight into the time complexity of our configuration approaches, we performed another experiment. The result of this experiment is shown in Figures 10 and 11. In each case, we measured the average response time for randomly generated configuration scenarios. Figure 10 shows that for our original approach the response time (i.e., the time that it takes to complete one configuration iteration) grows quadratically with the size of the configuration problem (i.e., the number of configurable parameters). On the other hand, as shown in Figure 11, in the case of our backtrack-free configuration approach, this growth is linear with the size of the configuration problem.

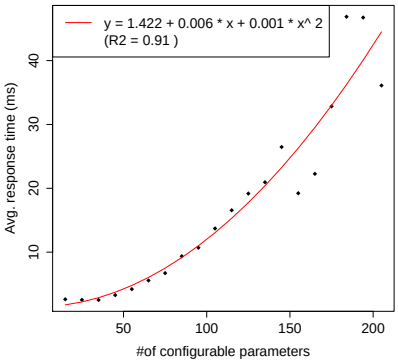


Fig. 10. Quadratic growth of the average response time in our original configuration approach.

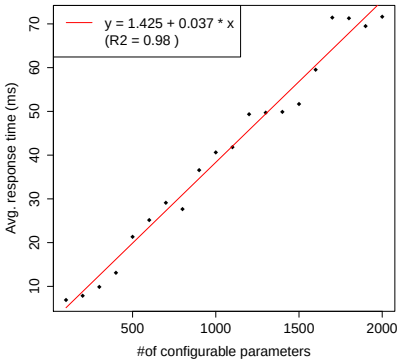


Fig. 11. Linear growth of the average response time in our backtrack-free configuration approach.

Furthermore, Figure 11 gives an insight into the usability of our backtrack-free configuration approach. According to a study reported in [25], 0.1 second is about the limit for having the user feel that the system is reacting instantaneously. Figure 11 shows that our backtrack-free configuration approach can respond instantly even for configuration problems with up to 2000 parameters.

6 Discussion

Normally, backtracking is used to explore the search space, in order to resolve inconsistencies, or to find all solutions. In our configuration approach [4], we use backtracking to resolve inconsistencies that may arise while configuring a single product. Analyzing configuration scenarios shows that, in our approach, most of these inconsistencies are due to early configuration of parameters that are involved in some pending constraints. By delaying the configuration of such parameters, using our ordering approach, we can prevent inconsistent configurations. One should note that our approach cannot generally eliminate backtracking for every use-case, such as enumerating all configurations, or resolving inconsistencies that may arise due to cyclic constraints.

For our backtrack-free configuration approach to be able to produce consistent and complete products, the input reference architecture model must be

cycle-free. In particular, to guarantee the termination of the configuration algorithm, the component hierarchy must be acyclic. Achieving this property for embedded software systems, where the software architecture follows, to a great extent, the architecture of hardware, is straightforward. To guarantee consistency, without requiring backtracking, the model should contain no cyclic constraints. Whether this restriction can negatively affect the applicability of our approach is a question that requires further investigation. Identifying the likelihood of embedded systems with cyclic constraints and proposing heuristics for ensuring their consistency with a minimum number of backtracks is left for future work. Finally, our proposed ordering approach may introduce some rigidity. Whether this rigidity affects usability negatively or not is an open questions that should be studied in future.

7 Conclusion

Constraint solving is generally used to ensure consistency of configurations, which are an essential part of software development in today's industries. A drawback of these techniques is the need for backtracking, which in the case of interactive configuration drastically hampers usability. In this paper, we proposed a partial ordering over the configurable parameters. The ordering is derived from a static analysis of the constraints between the parameters. Using the ordering approach, we have implemented a backtrack-free configuration tool. We performed a number of experiments using a case study from an industry partner. Results of our experiments show that our backtrack-free configuration tool ensures consistency, while preventing the need for backtracking. Furthermore, our approach significantly reduces the overall configuration time.

Acknowledgements

The first author acknowledges the Research Council of Norway (the ModelFusion Project - NFR 205606). The second author is funded by the National Research Fund - Luxembourg (FNR/P10/03 - Verification and Validation Laboratory).

References

1. A. A. Armstrong and E. H. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *AAAI/IAAI*, 1997.
2. R. J. Bayardo and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI*, 1996.
3. R. Behjati and S. Nejati. Backtrack-free consistent configuration of cyber-physical systems. `http://simula.no/publications/Simula.simula.2608`, 2014.
4. R. Behjati, S. Nejati, and L. C. Briand. Architecture-level configuration of large-scale embedded software systems. *Accepted for publication in TOSEM*, 2014.
5. R. Behjati, T. Yue, L. C. Briand, and B. Selic. SimPL: a product-line modeling methodology for families of integrated control systems. *Information and Software Technology*, 2013. Special Issue on Software Reuse and Product Lines.
6. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 2010.

7. D. Benavides, S. Segura, P. Trinidad, and A. Ruiz Cortés. FAMA: tooling a framework for the automated analysis of feature models. In *VaMoS*, 2007.
8. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 1986.
9. M. Carlsson and P. Mildner. SICStus Prolog – the first 25 years. *CoRR*, 2010.
10. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP*, 1997.
11. K. Czarnecki and P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Workshop on Software Factories at OOPSLA*, 2005.
12. R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artif. Intell.*, 136(2), 2002.
13. B. K. Eames, S. Neema, and R. Saraswat. DesertFD: a finite-domain constraint based tool for design space exploration. *Design Autom. for Emb. Sys.*, 14(2), 2010.
14. C. M. Fonseca and P. J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 1995.
15. E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM (JACM)*, 1982.
16. F. Glover and E. D. Taillard. A user's guide to tabu search. *Annals OR*, 1993.
17. T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *PETO*, 2004.
18. P. V. Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). In *Selected Papers from Constraint Programming: Basics and Trends*, 1995.
19. Á. Horváth and D. Varró. Dynamic constraint satisfaction problems over models. *Software and Systems Modeling*, 2010.
20. M. Janota, G. Botterweck, R. Grigore, and J. Marques-Silva. How to complete an interactive configuration process? In *SOFSEM*, 2010.
21. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 1983.
22. F. J. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer-Verlag New York, Inc., 2007.
23. R. Mazo, C. Salinesi, D. Diaz, and A. Lora-Michiels. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In *ENASE*, 2011.
24. Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.
25. Jakob Nielsen. *Usability Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
26. OMG. OMG Object Constraint Language (OMG OCL), Version 2.3.1, 2012.
27. K. Pohl, G. Böckle, and F. J. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., 2005.
28. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier Science Inc., New York, NY, USA, 2006.
29. R. M. Smullyan. *First-order logic.* Springer, 1968.
30. E. R. van der Meer, A. Wasowski, and H. R. Andersen. Efficient interactive configuration of unbounded modular systems. In *SAC*, 2006.
31. Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *ICSE'12*, 2012.