Backtrack-Free Consistent Configuration of Cyber-Physical 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
raziehb@simula.no, shiva.nejati@uni.lu

Abstract. Configuration is a recurring problem in many domains. In this paper, we focus on architecture-level configuration of Cyber-Physical Systems (CPS). In this context, engineers configure products by instantiating a given reference architecture model. The elements in each product instance 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 satisfying the constraints. Backtracking configuration decisions makes the configuration process considerably slow. In this paper, we propose a backtrack-free configuration mechanism. Specifically, given a generic reference architecture, we provide an ordering over configuration parameters. Utilizing this ordering over parameters, we then propose a configuration algorithm and prove that our algorithm produces consistent products without ever requiring backtracking.

1 Introduction

Cyber-Physical Systems (CPS) are large-scale heterogeneous systems that are used in many industry sectors where hardware and software systems are integrated to control production processes and to ensure safety aspects. Examples include the energy, automotive, and avionics industries. In most situations, product-line engineering approaches [25, 29, 23] are applied to develop integrated control systems. Software development, in this context, is done through configuring a *reference architecture*, which provides a common, high-level, and customizable structure for all members of the product family [25].

Briefly a reference architecture specifies component types and their configurable parameters, as well as, constraints capturing relationships between these parameters. Through configuration, engineers specify each product by creating a number of component instances related to their desirable selected component types. In addition, they have to assign values to configurable parameters for each component instance such that the constraints over parameters are satisfied. In most approaches, value assignment is done iteratively 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 engineer 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.

Many existing configuration approaches (e.g., [5, 9, 11, 20, 24]) are not interactive and produce final configured products without requiring intermediate input from users. In contrast, some configuration approaches (e.g., [16, 21, 28, 30]) interact with users, providing them with guidance to achieve consistent configurations and warn them when backtracking is required. Backtracking complicates the configuration process both computationally and conceptually. Some more recent approaches [16, 28] have eliminated backtracking by computing the space of all consistent solutions a priori to the configuration. These approaches fail to scale to most CPSs where the complexity of constraints and the size of the configuration space is so large that makes it impossible to compute the set of all possible configurations.

In this paper, we propose a new approach and eliminate backtracking during configuration by configuring parameters in a certain order. Specifically, given a cycle-free reference architecture, we propose an ordering over configurable parameters that prevents possibility of any backtracking during the configuration process. We show how an ordering is extracted from a cycle-free reference architecture, and prove 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.

In the rest of the paper, we first present the related work and position our work in the literature. Sections 3 and 4 provide an overview of the main concepts in product family modeling, and the configuration process. In Section 5, we present the required formalism for explaining the ordering approach, which is presented in Section 6. Finally, we present our backtrack free configuration algorithm in Section 7, and conclude the work in Section 8.

2 Related Work

Existing configuration approaches fall into two general categories, *non-interactive* and *interactive*. Most configuration approaches belong to the first category, 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., [11, 20]) or find all configuration solutions (e.g., [5, 9, 24]). The non-interactive approaches may either rely on meta-heuristic search approaches [12, 15, 22], or on systematic search techniques used in constraint solvers [7, 8, 18], or on symbolic decision procedures [6]. 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., [16, 21, 28, 30]) 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 [10], identifying no-goods constraints [1, 2], and ordering the search [13, 14]. 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 [16, 28] 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 CPSs, the complexity of constraints and the size of the configuration space is so large, 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 configuration. Our approach applies to architecture-level configuration of CPSs with architectural dependencies and constraints specified in First-Order Logic (FOL) [26]. Computation of ordering in our work is fast and performed based on static analysis of architectural models and the constraints syntax.

3 Motivating example

A product family can be described by a *reference architecture*, which provides a common and high-level structure for all members of the family [25]. A reference architecture specifies different types of reusable components that may exist in some members of the product family. Each component may have relationships with other components, and has a number of *configurable elements* and *configurable parameters* through which it defines a number of variability points.

Several approaches (e.g., [17, 9, 27]) exist for modeling the reference architecture of a product family. In our earlier work [4], we proposed a UML-based product-line modeling methodology called SimPL. In this approach, a reference architecture mainly consists of a class diagram, which describes all component types and their relationships, and a set of OCL constraints. Figure 1 is a class diagram showing an excerpt of a simplified reference architecture for a family of subsea oil production systems.

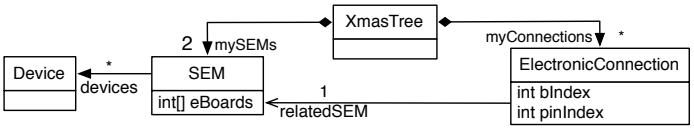

Fig. 1. An excerpt of a subsea oil production system reference architecture model.

In our approach, the class diagram has a topmost element (e.g., class XmasTree in Figure 1). Each product contains one instance of the topmost class, and

all other components are direct or indirect sub-components of that instance. This way, we can consider a product to simply be a component that can be created and configured like any other component.

In our approach, we have identified four categories of configurable parameters: *configurable attributes* (e.g., pinIndex), *configurable cardinalities* (e.g., size of array myConnections), *configurable topologies* (e.g., relatedSEM), and *configurable types* (an example for this does not exist in Figure 1). Each type and its configuration is explained in detail in [4, 3].

4 Configuration process

To explain the configuration process, we introduce the notion of *configuration tree*. Each product is represented by a configuration tree. Each node in a configuration tree has a name and a label. Edges are labeled as well. In Section 5.2, we explain how nodes and edges of a configuration tree are created and labeled.

Each leaf node in a configuration tree represents a configurable parameter. In addition to the name and label, such a node has a domain. The domain of a leaf node is a finite set of literals that can be assigned to the corresponding configurable parameter when configuring it.

Configuration is the process of assigning values to the leaf nodes (i.e., configurable parameters). Value assignment is done by a configuration engineer. When configuring a leaf node, the engineer has to select a value from the domain of that node. As a result of configuring a node, modifications may be needed in the configuration tree. In Section 5.2, we explain how the configuration tree changes after each step of configuration.

The domain of a leaf node depends on the type of the corresponding configurable parameter, and the values and the domains of other configurable parameters that are related to it. Relations among configurable parameters are specified using constraints (i.e., OCL constraints). After each configuration step, the value assigned to a node may remove some values from the domains of other leaf nodes. In Section 7, we explain how the domains are computed in each configuration step using the constraints and the values assigned to configurable parameters.

5 Formal definitions

In this section, we provide the formal definitions that are needed for explaining the ordering approach, and proving that using our ordering solution we can guarantee the consistency of configured products without requiring backtracking.

5.1 Component

Here we formally describe components, which are the main building blocks of products. Recall from Section 3 that each product is itself a component.

Definition 1 (Component). *A component c is defined as a tuple $(id, \mathcal{V})$, where id is a unique identifier, and $\mathcal{V}$ is a set of configurable elements.*

Let $c = (id, \mathcal{V})$ be a component. 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 2 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. 2. A simplified grammar for types.

In Figure 2, 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.

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). Configurable elements of a primitive type or a referenced type represent configurable parameters. In this paper, we focus on three categories of configurable parameters, namely, configurable attributes, configurable topologies, and configurable cardinalities. Configurable types can be handled in a similar manner as the other categories. However, for the sake of succinctness we have excluded them from this report.

5.2 Configuration tree

As mentioned in Section 4, each product can be represented by a *configuration tree*. Each component in the product maps to a subtree of the configuration tree. The configuration tree grows and expands as configurable parameters are configured throughout the configuration process. In the following, we explain the creation and expansion of configuration trees.

5.2.1 Configuration tree for an un-configured component

Here we explain how to create a configuration subtree for a given unconfigured component $c = (id, \mathcal{V})$. Figure 3-(a) shows such a subtree for an instance of XmasTree (from Figure 1). The identifier of this component is xt1.

The root of the configuration subtree of $c = (id, \mathcal{V})$ is labeled by id. For each element $e = (id_e, t_e) \in \mathcal{V}$, we add a node directly under the root. An edge labeled by id_e connects the root to this added node. The following explains the details of each node based on t_e, the type of the configurable element e.

t_e is a single primitive type: In this case, the added node is a leaf node labeled by t_e. This node represents a configurable attribute. We call such nodes *primitive nodes*. Nodes m13 and m14 in Figure 3-(b) are primitive nodes.

t_e **is a single user defined type:** As mentioned above, element e in this case represents a sub-component $c' = (id', \mathcal{V}')$ of c. We add a non-leaf node labeled by id'. Such a node is called a *component node*, and is the root of a subtree that is created for c' in the same manner as c. At this stage, c' is un-configured.

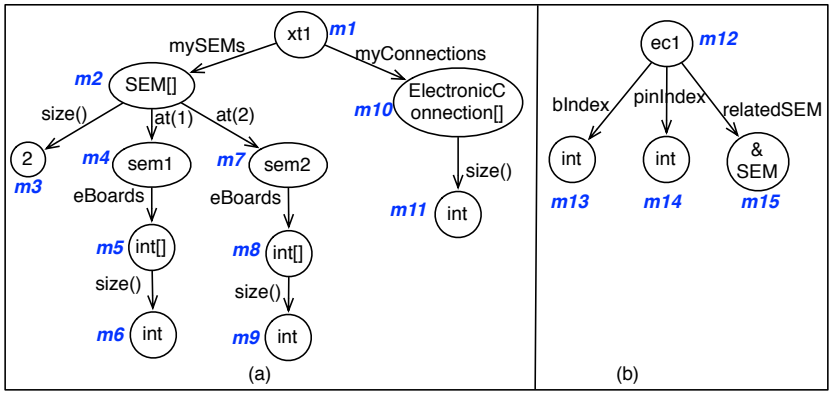

Fig. 3. Configuration subtrees representing two components. Text inside a circle represents the node's label, and the text next to a node represents its name. (a) is an instance of XmasTree, and (b) is an instance of ElectronicConnection.

t_e **is a referenced user defined type:** The added node, in this case, is a leaf node labeled by t_e. This node represents a *configurable topology*. We refer to such nodes as *reference nodes*. Node m15 in Figure 3-(b) is a reference node.

t_e **is an arrayed type:** The added node is a non-leaf node n, labeled by t_e. Such a node is called an *array node*. Additional nodes should be added as children of n. Two following cases exist:

- *Arrayed element e has an unfixed (configurable) size:* An additional node n' is added as n's child. The edge connecting n to n' is labeled by $size()$. Node n' represents a configurable cardinality, and is labeled by 'int' to indicate that an integer value is needed to configure it. An example of this case are nodes m5 (as n) and m6 (as n') in Figure 3-(a).
- *Arrayed element e has a fixed size k:* Similar to the previous case, an additional node $n1$ is added, and connected to n via an edge labeled $size()$. Furthermore, we add k new nodes, $n_1 ... n_k$, as n's children. Each node n_i is connected to n via an edge labeled $at(i)$. Let t_e be $t[]$, meaning that items in the arrayed element e are of type t. We create nodes $n_1 ... n_k$ as follows:
  - If t is a primitive type, then each node n_i is a primitive node labeled by t (meaning that n_i represents a configurable attribute that should be configured in later steps of configuration).
  - If t is a referenced type, then n_i is a reference node labeled by t representing a configurable topology.
  - If t is a user defined type, then each item in the array represents a sub-component $c_i = (id_i, \mathcal{V}_i)$ of c. Each n_i is a component node, which is the root of a subtree corresponding to one of these sub-components. We

label n_i by id_i, and create the subtree beneath it in the same manner as c. All n_is are unconfigured at this stage. In Figure 3-(a), $m2$ is an array node with a fixed size two. Nodes m4 and m7 are the component nodes that represent the items in the array.

5.2.2 Configuration and its impact on the configuration tree

In the configuration tree, each leaf node is labeled by a primitive type or a referenced type and represents a configurable parameter. In each step of configuration, a value is assigned to a configurable parameter. In the case of configurable attributes, the value is either a boolean literal, or an integer literal. In the case of configurable cardinalities, the value is an integer literal, and in the case of configurable topologies, the value is $\&id'$, where id' is the identifier of a component.

As a result of configuring a configurable parameter, the configuration tree changes. In particular, the label of a node may change, or new nodes may be added. Figure 4 shows an example configuration, where the subtree beneath component node *m1* in part (a) of Figure 3 is configured.

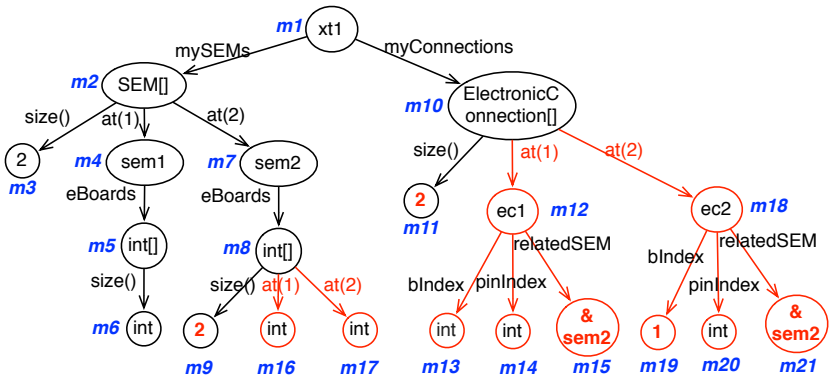

Fig. 4. One possible partial configuration of node *m1* in Figure 3-(a).

In the following, we explain in details how the configuration tree changes as a result of configuring a configurable parameter. Let n be a leaf node representing a configurable parameter, k be the value assigned to the parameter, and n' be the immediate parent of n in the configuration tree.

Node n represents a configurable attribute: We replace the label of n with k. Node m19 in Figure 4 shows an example of this configuration case. In this example, value 1 is assigned to the configurable parameter represented by m19.
Node n represents a configurable topology: We replace the label of n with k. In this case k is of the form $\&id'$, where id' represents the unique identifier of a component. This implies that there should be another node, in the configuration tree, labeled id'. Node m15 in Figure 4 shows an example of this configuration

case. In this example, the value &sem2 is assigned to the configurable parameter. Note that sem2 is the label of the component node m7.

Node n represents a configurable cardinality: In this case, we add k new nodes $n_1 ... n_k$ as children of n'. We connect each node n_i to n' with an edge labeled $at(i)$. Each node n_i is created according to its type, which is deducible from the label of n', by following the same rules presented in Section 5.2.1. Nodes m12 and m18 in Figure 4 are added as a result of assigning value two to the configurable cardinality represented by node m11.

5.3 Qualified names for accessing nodes

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. 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.

Let a be a string representing an array node in the configuration tree, and i be a string representing an integer (e.g., a node of type 'int' in the configuration tree). For the sake of conciseness, we use $a[i]$ instead of $a.at(i)$.

A string created using this tree traversal approach represents a *qualified name*. Figure 5 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 5 is added to explicitly define int_qName and bool_qName as *primitive qualified names*. Primitive qualified names represent configurable parameters that can appear in constraint specifications.

```
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. 5. The grammar of qualified names.

5.3.1 Length of a qualified name

The first 12 rules in Figure 5 can be divided into three categories:

Cat.1 Rules of the form *non-term1 ::= term*, or
Cat.2 Rules of the form *non-term1 ::= non-term2 '.' term*, or
Cat.3 Rules of the form *non-term1 ::= non-term2 '[' non-term3']'*

The first category (i.e., rule 6) creates a qualified name of length one.

Let qualified name $q1$ be created from qualified name $q2$ by applying a rule of the second category (i.e., Cat.2), and let n be the length of $q2$. Then, length of $q1$ is $n + 1$.

Let qualified name $q1$ be $q2[q3]$, created from $q2$ and $q3$ by applying a rule of the third category. Let n be the length of $q2$. Then, length of $q1$ is $n + 1$. Note that in this case, the length of $q1$ is independent from the length of $q3$.

5.3.2 Prefixes of a qualified name

Considering the last two categories of rules mentioned above, a qualified name q is either of the form $q1.t$ or $q1[q2]$. In either case, we refer to $q1$ as a prefix of q. Note that the prefix of a qualified name is itself a qualified name.

Let q be a qualified name, q_1 a prefix of q, and q_2 a prefix of q_1. Then q_2 is, as well, a prefix of q. This allows recursively identifying shorter prefixes of qualified names. A qualified name of length one (i.e., created using rule 6) does not have any prefixes.

5.3.3 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 5 are semantically valid with respect to the configuration tree CT. We use $Q(CT)$ to denote this subset. The following rules specify $Q(CT)$:

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

5.3.4 Mapped and unmapped qualified names

Let CT be a configuration tree, and q be a 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.

Considering the description of semantically valid qualified names above, there are two cases where an unmapped qualified name q can be created from a mapped qualified name $q1$:

1. $q = q1.t$, and $q1$ maps to an unconfigured reference node (which will eventually be configured to refer to a component node representing a component $c = (id, \mathcal{V})$). Note that if $q1$ maps to a component node, then $q1.t$ cannot be unmapped.
2. $q = q1[q2]$, and $q1$ maps to an array node, and $q2$ is an unconfigured semantically valid qualified name representing an integer parameter. Note that in this case, if $q2$ is an integer literal, or if it is configured, then $q1[q2]$ cannot be unmapped.

Furthermore, the description of semantically valid qualified names implies that mapped qualified names cannot be created from unmapped qualified names. Therefore, prefixes of a semantically valid mapped qualified name are all mapped. In addition, each semantically valid unmapped qualified name has at least one mapped qualified name. Let $P = \{p_1, ..., p_n\}$ be the set of all mapped prefixes of a semantically valid unmapped qualified name q. The definition of prefixes implies that each p_i of length i in the set P is a prefix of all p_js of length $j > i$. This way, we can define a total order over elements in P based on their lengths.

Based on the above definitions, in the following, we present a lemma that is used in the definition of the ordering approach, and the backtrack-free configuration.

Lemma 1. *Let CT be a configuration tree, q be a semantically valid unmapped qualified name, and $q1$ be the longest mapped prefix of q. Then, either $q1$ is mapped to a reference node, or there exists a qualified name $q2$, such that either $q = q1[q2]$, or $q1[q2]$ is a prefix of q.*

Proof. We prove by induction over the size of the qualified name q.

Base case. For a given configuration tree CT, every qualified name of length one or two that is semantically valid w.r.t. CT, has a mapping node in the tree. According to Figure 5, a qualified name of length one represents a component identifier, and has a mapping node in the tree. Any qualified name of length two is of the form $id.t$, where id is the identifier of a component $c = (id, \mathcal{V})$, and t is the name of one of the elements in $\mathcal{V}$. According to configuration tree creation approach in Section 5.2.1, all elements in $\mathcal{V}$ have a corresponding node in CT. Therefore, a qualified name of length two cannot be unmapped.
Let q be an unmapped qualified name of length three. There are two groups of rules in Figure 5 that may result in such a node. The first group are rules 1, 5, 7, 9, 10 and 12, when the element qualified name represents an element of a referenced type. In this case, q is of the form $id.referencedElement.prop$. As discussed above, $id.referencedElement$ has to be a mapped qualified name. It maps to a reference node and is the longest mapped qualified name of q. Therefore, the lemma holds for this base case.
The second group includes rules 3, 8 and 11. All these rules result in a qualified name q of the form $id.array\_prop[int\_q]$ (i.e., $q1[q2]$), where id is the identifier of a component, $array\_prop$ identifies an arrayed element of the component, and int_q is some integer factor. In this case, $id.array\_prop$ is the longest mapped

prefix of q. Therefore, the lemma holds for the base case.

Inductive step. Suppose that Lemma 1 holds for every unmapped qualified name of size n. Let q be an unmapped qualified name of size $n+1$, and $q1$ be a prefix of q of size n. The following two cases exist according to the mapping state of $q1$.

- $q1$ is a mapped qualified name. In this case, $q1$ is the longest mapped prefix of q. As discussed above, there are two cases where an unmapped qualified name (e.g., q) can be created from a mapped qualified name (e.g., $q1$):
  - $q = q1.t$, and $q1$ maps to an unconfigured reference node.
  - $q = q1[q2]$, and $q1$ maps to an array node, and $q2$ is an unconfigured semantically valid qualified name representing an integer parameter.

  Therefore, the lemma holds for q and $q1$.
  Note that the other two rules for creating semantically valid qualified names in Section 5.3.3 do not result in unmapped qualified names.
- $q1$ is an unmapped qualified name. Therefore, according to the assumption of the inductive step, $q1$ has a longest mapped prefix p for which the condition in the lemma holds. Since $q1$ is unmapped, then p has to be the longest mapped prefix of q. Therefore, the lemma holds for q.

□

5.4 Constraints

Let $c = (id, \mathcal{V})$ be a component, and Φ be a set of constraints defined in the context of c. Each member of the set Φ is a boolean expression denoting a constraint ϕ. A simplified grammar for the language of boolean expressions is given in Figure 6. This grammar is defined based on the basic OCL operators that we use in specifying constraints in the SimPL methodology. These operators include, for all, exists, arithmetic, relational and logical operators. In Figure 6, 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. 6. A simplified grammar of boolean formulas.

Three types of qualified names defined in Figure 5 (i.e., bool, int, and array qualified names) are used in the production rules of the grammar given in Figure 6. 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.

We use constraints to compute domains of the leaf nodes. We use constraint propagation over finite domains [18] to ensure the consistency of the domains. Suppose that we are given a configuration tree and a set of constraints defined in the context of some components in the tree. Constraint propagation considers a constraint only if all of its qualified names map to some nodes in the configuration tree. We refer to these constraints as the *ready-to-evaluate* constraints.

6 The ordering approach

Backtracking is needed when some nodes are assigned inconsistent values. Such inconsistent values may exist in the domain of a node if some constraints are not considered during constraint propagation. This happens if a node is configured before all its related constraints are ready-to-evaluate. In order to avoid inconsistent value assignment and therefore backtracking, we delay the configuration of each node until all its constraints are ready-to-evaluate. This is done through defining a partial ordering among the nodes in the configuration tree.

In the rest of this section, we present two ordering rules. Then, we propose the notion of *ordering graph*, which specifies the partial ordering among nodes, based on the ordering rules. Finally, we present some restrictions that should be applied on the reference architecture model to prevent any possibility of deadlock in the ordering graph. The first ordering rule is called *constraint-based ordering*, as it derives the ordering based on constraint specifications. The second ordering rule is called *topology-based ordering*, as it derives the ordering based on the associations among classes in the reference architecture.

6.1 Constraint-based ordering

Let $CT = (N, E)$ be a given configuration tree, and Φ be the set of all constraints defined in the context of some components in CT. Let Q be the set of all qualified names appeared in one or more constraints in Φ. For each q in Q, let $X(q)$ denote the set of all primitive qualified names in Q that are directly or indirectly related to q through some constraints in Φ.

Example 1. Let CT be the configuration tree in Figure 4, and Φ be $\{\phi_1, \phi_2\}$, where ϕ_1 and ϕ_2 are the following two constraint, both defined in the context of the component identified by $ec1$.

```
ec1.bIndex <= ec1.relatedSEM.eBoards.size()
ec1.pinIndex <= ec1.relatedSEM.eBoards[ec1.bIndex]
```

Then, Q is {ec1.relatedSEM.eBoards.size(), ec1.relatedSEM.eBoards[ec1.bIndex], ec1.bIndex, ec1.pinIndex}. We can create a set $X(q)$ for each of the qualified names in Q. For example, set X(ec1.relatedSEM.eBoards[ec1.bIndex]) contains only one element, which is ec1.pinIndex. ■

For each unmapped primitive qualified name $q \in Q$, the ordering rule identifies the set of all nodes $pre(q) \subset N$ that should be configured before any qualified name in $X(q)$ can be configured. Let p be the longest mapped prefix of q. According to Lemma 1, two cases are possible. In the following, we explain what constitutes $pre(q)$ in each case.

- If p is mapped to a reference node, then $pre(q)$ contains p.
- Otherwise, there exists a qualified name p', such that either $q = p[p']$, or $p[p']$ is a prefix of q. Using these, we define $pre(q)$ as follows:
  - $pre(q)$ contains $p.size()$, if it is un-configured.
  - $pre(q)$ contains p', if p' is an un-configured mapped qualified name.
  - $pre(q)$ contains all members of $pre(p')$, if p' is unmapped.

Example 2. Qualified name ec1.relatedSEM.eBoards[ec1.bIndex] is an unmapped qualified name in the configuration tree in Figure 4 (because ec1.bIndex is not assigned a value). The set pre(ec1.relatedSEM.eBoards[ec1.bIndex]) is therefore, {ec1.bIndex}. According to the ordering rule, this qualified name should be configured before ec1.pinIndex can be configured. ■

6.2 Topology-based ordering

The topology-based ordering restricts the configuration of configurable topologies. In particular, a parameter of a referenced type can only be set to an unconfigured element in the tree.

Consider classes ElectronicConnection and SEM in Figure 1. The association between these two classes results in a configurable topology in each instance of ElectronicConnection. The topology-based ordering states that these configurable topologies should be configured before any existing instance of SEM can be configured. For example, in Figure 4, none of the nodes beneath m4 and m7 could be configured before the configuration of m15 and m21.

6.3 The ordering graph

An ordering graph is a directed acyclic graph, where nodes are a subset of nodes in the configuration tree of the product under configuration, and each edge connecting a node m to a node n implies that m has to be configured before n. In the ordering graph, a node with a zero indegree is *ready-to-configure*.

Let $CT = (N, E)$ be a configuration tree, and Φ be the set of all constraints defined in the context of some component in CT. Algorithm 1 describes how an ordering graph is created from CT and Φ.

The first part of the algorithm (lines 8-14) iterates over all unmapped qualified names q in Q, and extracts a subset of leaf nodes of the configuration tree that correspond to the mapped qualified names in $X(q)$. This is done in line 9 of the algorithm. A set of nodes corresponding to elements in $pre(q)$ are created in line 10. The ordering graph is created by adding these nodes to the graph, and adding an edge for each pair (p', p) as in line 14.

Algorithm 1 MKORDERINGGRAPH

Input: a configuration tree $CT = (N, E)$, and a set of constraints Φ
Output: the ordering graph $Ord = (N', E')$

```
 1 Q ← set of all qualified names in Φ
 2 L ← set of all leaf nodes in CT
 3 T ← set of all un-configured referenced nodes in CT
 4 C ← set of all component nodes in CT
 5 N' ← L ∪ C
 6 E' ← ∅
 7 ▷ Constraint-based ordering
 8 for each unmapped q ∈ Q do
 9    X_m ← set of all mapped qualified names in X(q)
10    pre ← pre(q)
11    N' = N' ∪ pre
12    for each p ∈ X_m do
13       for each p' ∈ pre do
14          E' = E' ∪ {(p', p)}
15 ▷ Topology-based ordering
16 for each t ∈ T do
17    for each c ∈ C do
18       if c ∈ domain(t) then
19          for each n ∈ subtree(c) do
20             E' = E' ∪ {(t, n)}
21 return (N', E')
```

The second part of the algorithm (lines 16-20) iterates over all unconfigured configurable topologies and connects each to all nodes that appear in the subtree beneath a component that can potentially be used to configure that topology. Note that the domain of a configurable topology of type &ClassX contains all unconfigured instances of class ClassX in the tree.

Time analysis. Here we present an amortized worst-case analysis of the time complexity of the ordering algorithm. Let Q denote the total number of configurable parameters that need to be configured to create a particular product. Let U be number of unmapped qualified names. We have U < Q. Since for an un-

mapped qualified name q, the set $pre(q)$ is created based on the prefixes of q and indexes used in it, the number of elements in $pre(q)$ cannot exceed a constant number K that can be extracted from the reference architecture. Let C be the maximum number of constraints in which a qualified name may appear. Then the time complexity of the first for-each loop is $O(U \times (C + K + Q \times K))$. For large-scale cyber-physical systems, we usually have C < Q. Assuming that K is constant for a reference architecture, the time complexity of applying constraint-based ordering is $O(Q^2)$.

Now, let T denote the total number of configurable topologies, and N denote the total number of nodes in the configuration tree. The for-each loop in lines 17-20 visits each node at most once. Therefore, the complexity of the for-each loop in lines 16-20 is $O(T \times N)$. However, we have T < Q, and N = O(Q). Therefore, the complexity of applying topology-based ordering is $O(Q^2)$.

In total, the time complexity of creating a full ordering graph is $O(Q^2)$.

6.4 Deadlock avoidance

In general, it is possible to have circles in the ordering graph. Such circles result in deadlock situations, where a group of nodes are waiting for each other to be configured. To avoid deadlocks, we impose two restrictions on the reference architecture, one on the class diagram, and the other on the OCL constraints. We call such a reference architecture *cycle-free*.

The first restriction specifies that there should not be any circular relationships in the class diagram. This guarantees that the topology-based ordering does not result in any cycles.

To explain the second restriction, we define the notion of *circular constraints*. To avoid deadlock when applying the constraint-based ordering, the set of constraints in the reference architecture must have no circular constraints.

Definition 2 (Circular constraints). *Let q_1 to q_n be some mapped qualified names, q'_1 to q'_n be some unmapped qualified names, and $\phi_1(q_1, q'_1)$ to $\phi_n(q_n, q'_n)$ be some constraints. These constraints form a set of circular constraints, if the following conditions hold:*

- *for each $i \in [2..n]$: $q_i \in pre(q'_{i-1})$, and*
- *for each $i \in [1..n-1]$: $q_i \in pre(q'_{i+1})$, and*
- *$q_1 \in pre(q'_n)$ and $q_n \in pre(q'_1)$*

Note that the mapping state of a qualified name changes over time. When identifying the possibility of circularity of constraints, we consider constraints right after their creation. OCL constraints defined in the context of a class are instantiated each time a new instance of a class is created. All constraints in a set of circular constraints must be defined in the context of the same component, and are created at the same time. This allows us to verify the existence of circular constraints at the class-level, and before beginning the configuration process.

7 Backtrack-free configuration

Algorithm 2 shows the backtrack free configuration algorithm. Input to the algorithm is a reference architecture, which contains a class diagram and a set of OCL constraints. An initial configuration tree, CT, is created in line 1. Figure 3-(a) shows the initial configuration tree for the reference architecture in Figure 1. An initial set of constraints Φ are then extracted from the reference architecture, particularly the OCL constraints. These constraints are used to compute the domains of the leaf nodes in CT.

Algorithm 2 BTFreeConfig

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

```
 1 CT ← mkInitialConfigTree(RA)
 2 Φ ← extractConstraints(RA)
 3 G ← mkOrderingGraph(CT, Φ)
 4 D ← computeValidDomains(CT, Φ)
 5 while readyToConfig(G) ≠ ∅ do
 6    read(i) ▷ index of a ready-to-configure node
 7    read(tmp) ▷ value to be assigned to the selected node
 8    ▷ tmp must be in D_i (the domain of the selected node)
 9    while not tmp ∈ D_i do
10       read(tmp)
11    (CT, Φ) ← applyConfiguration(CT, Φ, i, tmp)
12    G ← mkOrderingGraph(CT, Φ)
13    D ← computeValidDomains(CT, Φ)
14    if some domains in D are empty then
15       throwException()
16 return CT
```

Domains of the leaf nodes are computed in line 4 using the routine computeValidDomains. For this purpose a constraint program is created from the constraints in the set Φ. A constraint program is a triple $\mathcal{P} = (X, D, C)$, where X is a set of variables, D is a set of domains of those variables, and C is a set of constraints over those variables. A constraint program is arc-consistent [19, 18] if for each $v_i \in D_i$, there exist values $v_1, ..., v_{i-1}, c_{i+1}, ..., v_k$ in domains $D_1, ..., D_{i-1}, c_{i+1}, ..., D_k$ such that $v_1, ..., v_k$ satisfy all the constraints in C. Given an arbitrary constraint program $\mathcal{P} = (X, D, C)$, a constraint solver can produce an arc-consistent constraint program $\mathcal{P}' = (X, D', C)$ by pruning the domains in D, provided that $\mathcal{P}$ has a solution (i.e., at least one combination of the values in D satisfy all the constraints in C). In computeValidDomains, we use a constraint solver to prune the domains by removing all inconsistent values. The resulting domains are used as the domains of the leaf nodes in the configuration tree. Note that only the ready-to-evaluate constraints and their variables are considered when pruning the domains.

The while loop in lines 5-15 iterates over the set of ready-to-configure nodes and, in each iteration, configures one node. Both the node and its value are selected by a human user. Lines 9 and 10 guarantee that the selected value is within the domain of the node, and therefore, consistent. Finally, the configuration decision is applied and the configuration tree, the ordering graph, and the domains are updated in lines 11 to 13. The algorithm throws an exception, in line 15, if some domains become empty. Theorem 1 states that using the ordering approach presented in Section 6, the algorithm never reaches line 15. Note that without the ordering, some domains may become empty, therefore throwing an exception in line 15. In that case, consistency cannot be guaranteed without backtracking.

Theorem 1. *Given a consistent cycle-free reference architecture, Algorithm 2 terminates by producing a complete and consistent configuration, without ever throwing an exception.*

Proof. First, we prove that the algorithm terminates by showing that the while loop in lines 5-15 executes a finite number of times. Then, we prove that line 15 is never reached.

The while loop in lines 5-15 terminates when there are no ready-to-configure nodes in the ordering graph. The ordering graph always contains a subset of the un-configured nodes in the configuration tree. Since the reference architecture is cycle-free, there are no cycles in the ordering graph. Therefore, an empty set of ready-to-configure nodes implies that no unconfigured parameters are left. The total number of configurable parameters in a configuration tree is always finite, and in each iteration one ready-to-configure node is assigned a value, without ever retracting or changing this value. Therefore, the ordering graph will be empty after a finite number of executions of the while loop. Furthermore, after a finite number of configuration steps, we'll have a complete product without any unconfigured parameters. Since all values are selected from the valid domains of the configurable parameters, the product is guaranteed to be consistent.

The algorithm reaches line 15 only if the domains of some nodes become empty after a configuration iteration. This may happen if an inconsistent value is assigned to a node. Lines 9 and 10 of the algorithm guarantee that the value assigned to a node belongs to the domain of that node. Therefore, inconsistent configuration does not happen unless some domains contain inconsistent values. The subroutine computeValidDomains guarantees that before the beginning of each configuration step, the domains of all ready-to-configure nodes are arc-consistent. We prove that these domains remain arc-consistent at the end of each configuration step, by showing that they are not pruned by the constraints that may be added as a result of applying a configuration decision (line 11).

Let $CT^{(i)}$, $G^{(i)}$ and $D^{(i)}$ be the configuration tree, the ordering graph, and the domains of the ready-to-configure nodes before the ith step of configuration. Let $CT^{(i+1)}$, $G^{(i+1)}$ and $D^{(i+1)}$ be the configuration tree, the ordering graph, and the domains at the end of the ith configuration step, computed in lines 11-13, respectively. In general, as a result of applying the ith configuration decision, new

nodes may be added to the configuration tree, i.e., $CT^{(i+1)}$. There are two ways that these new nodes may result in new constraints to be used for calculating $D^{(i+1)}$ in line 13:

- **New primitive nodes.** Let n be such a node, and q be the corresponding qualified name. Suppose that q has appeared in a set of constraints Φ. None of these constraints could have been considered when calculating $D^{(i)}$. Having n in the tree changes the state of some constraints in Φ to ready-to-evaluate. Therefore, they can be used in updating the ordering graph and computing the valid domains. We prove by contradiction that none of these constraints prunes the domains of any of the ready-to-configure nodes in $G^{(i)}$. Let ϕ be an arbitrary constraint in Φ, and suppose that there exists a ready-to-configure node $n' \in G^{(i)}$ the domain of which is pruned by ϕ. Let q' be the qualified name represented by n'. Since ϕ prunes the domain of n', q' must be directly or indirectly related to q (from the definition of arc-consistency). However, since q did not exist in $CT^{(i)}$, the in-degree of n' must have been greater than zero in $G^{(i)}$. Therefore, the initial assumption that n' is a ready-to-configure node in $G^{(i)}$ is false. Therefore, expanding the configuration tree by adding new primitive nodes to it does not prune or empty any of the domains in $D^{(i)}$.
- **New component nodes.** Let n be a new component node, and Φ be the set of constraints defined in the context of n. Some constraints in Φ may be ready-to-evaluate and should be considered in line 13. However, since all these constraints are defined in the context of n they can only specify relationships among the nodes beneath n, and not any of the ones previously existing in the tree, unless n has a reference node beneath it. However, the topology-based ordering rule guarantees that none of the nodes that may be accessed through a reference node are among the ready-to-configure nodes in $G^{(i)}$. Therefore, none of the constraints in Φ can affect the domain of any ready-to-configure node in $G^{(i)}$. □

Algorithm 2 has a better performance compared to our earlier configuration algorithm in [3], which did not impose any ordering over the configurable parameters. The amortized worst-case analysis of the ordering algorithm presented in Section 6.3 shows that the total cost of computing the ordering for a complete product is $O(Q^2)$, where Q is the total number of configurable parameters in the product. In each configuration iteration, the subroutine mkOrderingGraph is invoked in line 12 to recompute the ordering graph. Assuming that this subroutine has a mechanism to update the existing ordering graph instead of creating a new one from scratch, the amortized time analysis implies that in each configuration iteration, the cost of updating the ordering graph is $O(Q)$. This cost is negligible compared to the constraint propagation cost, which according to [3] is $O(Q^2)$, in each iteration. Furthermore, the elimination of backtracking implies that, using Algorithm 2, the user has to assign at most one value to each parameter to make a consistent product. However, the configuration algorithm in [3] typically requires the user to try several values for each configurable parameter.

8 Conclusion and Future work

Configuration is an inseparable part of today's industries. Correctness of configuration results is a major concern in this context. Constraint solving is generally used to ensure consistency of configurations. 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 the reference architecture of the product family. We have proven that the ordering approach prevents the need for backtracking, while ensuring consistency. In future, we plan to evaluate the improvements that a backtrack-free algorithm can bring about when configuring realistic CPSs.

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, S. Nejati, and L. C. Briand. Architecture-level configuration of large-scale embedded software systems. *Accepted for publication in TOSEM*, 2014.
4. 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.
5. D. Benavides, S. Segura, P. Trinidad, and A. Ruiz Cortés. FAMA: tooling a framework for the automated analysis of feature models. In *VaMoS*, 2007.
6. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 1986.
7. M. Carlsson and P. Mildner. SICStus Prolog – the first 25 years. *CoRR*, 2010.
8. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP*, 1997.
9. K. Czarnecki and P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Workshop on Software Factories at OOPSLA*, 2005.
10. R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artif. Intell.*, 136(2), 2002.
11. 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.
12. C. M. Fonseca and P. J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 1995.
13. E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM (JACM)*, 1982.

14. E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM (JACM)*, 1985.
15. F. Glover and E. D. Taillard. A user's guide to tabu search. *Annals OR*, 1993.
16. 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.
17. Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *SPLC*, 2008.
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. P. Van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 1992.
20. Á. Horváth and D. Varró. Dynamic constraint satisfaction problems over models. *Software and Systems Modeling*, 2010.
21. M. Janota, G. Botterweck, R. Grigore, and J. Marques-Silva. How to complete an interactive configuration process? In *SOFSEM*, 2010.
22. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 1983.
23. 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.
24. 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.
25. K. Pohl, G. Böckle, and F. J. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.
26. R. M. Smullyan. *First-order logic*. Springer, 1968.
27. M. Stephan and M. Antkiewicz. Ecore.fmp: A tool for editing and instantiating class models as feature models. Technical report, University of Waterloo, 200 University Avenue West Waterloo, Ontario, Canada, August 2008.
28. E. R. van der Meer, A. Wasowski, and H. R. Andersen. Efficient interactive configuration of unbounded modular systems. In *SAC*, 2006.
29. D. M. Weiss and R. Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999.
30. Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012.