

Technical Report 2012-05

**Guided Interactive Configuration of Embedded Software Systems Using
Constraint Satisfaction over Finite Domains**

Authored by:

Razieh Behjati, Shiva Nejati, Arnaud Gotlieb, Tao Yue, and Lionel Briand

Date: June 8, 2012

[**simula** . research laboratory]
- *by thinking constantly about it*

Executive Summary

Modern society is increasingly dependent on highly-configurable software systems, in particular, architecturally configurable software such as that embedded in Integrated Control Systems (ICSs). Configuring ICSs is expensive, time-consuming and error-prone. This is due, in large part, to the fact that the hardware and software configuration processes are, typically, rather isolated from one another, resulting in many configuration errors to be detected only after the integration of software and hardware. In this work, we propose a model-based configuration approach that allows us to configure software in a stepwise manner, to automate some configuration decisions, and to iteratively validate software and hardware configuration decisions. Our approach has two major steps. In the first step (modeling), a generic model describing an ICS family is built. In the second step (configuration), we interactively guide a user to derive a particular product specification complying with the generic model of its ICS family. We use a constraint solver, SICStus Prolog, to evaluate user decisions at each round, to automatically infer configuration decisions and to ensure that software and hardware configurations are consistent. We evaluated our approach by applying it to a real subsea production system. Specifically, we rebuilt a number of existing verified product configurations of our industry partner. Our experience shows that our approach successfully enforces consistency of configurations, can automatically infer up to 50% of the configuration decisions, and reduces the complexity of making configuration decisions by guiding the user during the configuration process.

Contents

1	Introduction	6
2	Configuration of ICSs: Practice and Problem Definition	6
3	Overview of our approach	8
4	Product-line modeling	8
4.1	A subsea product-line model	9
4.2	Configuration activities in a model-based context	10
5	Interactive model-based guided configuration	10
5.1	Guided and automated configuration	11
5.2	Constraint satisfaction to provide guidance and automation	12
6	Prototype tool	13
6.1	The <code>clpfd</code> library of SICStus Prolog	13
6.2	Mapping to <code>clpfd</code>	14
7	Evaluation	14
7.1	Inference percentage	15
7.2	Reduction of valid domains	16
7.3	Constraint propagation efficiency	16
7.4	Discussion	17
8	Related Work	18
9	Conclusion	18

List of Figures

1	A fragment of a simplified subsea production system.	7
2	An overview of our configuration approach.	8
3	A fragment of the SimPL model for the subsea production system.	9
4	Architecture of the configuration tool.	13
5	Constraint propagation time grows quadratically with the number of variables (with a coefficient of determination of 0.9994).	17

List of Tables

1	Characteristics of the rebuilt configurations.	15
2	Inference rates.	15
3	Average shrinking of the domains.	16

1. Introduction

Modern society is increasingly dependent on highly-configurable software systems, and in particular, architecturally configurable software such as that embedded in Integrated Control Systems (ICSs). Examples of ICSs include industrial robots, process plants, and oil and gas production platforms. To control and monitor physical devices and processes, the embedded software in ICSs needs to be configured for hardware architectures such as various field layouts ranging from single satellite wells to large multiple sites, for data communication protocols such as standard protocols like Modbus [3] or some proprietary protocol, and for individual devices such as specific sensor resolution and scale levels. As a result, embedded software systems in ICSs have very large configuration spaces, and their configuration requires precise knowledge about hardware design and specification.

Current design and implementation strategies for embedded systems tend towards an early split between hardware and software development, with separate languages and development tools for different domains. This leads to the hardware and software configuration processes that are isolated from one another, resulting in many configuration mismatches and errors that are often detected very late and only after the integration of software and hardware.

Architecturally configurable software systems have been previously studied in the area of software product lines [18], where support for configuration largely concentrates on resolving high-level variabilities in feature models [16] and their extensions [14], e.g., the variabilities specified for end-users at the requirements-level. Feature models, however, are not easily amenable to capturing complex architectural variabilities as well as the hardware-software dependencies in embedded systems. Furthermore, existing configuration tools do not particularly focus on guiding engineers to resolve variabilities, while maintaining consistency between the software and hardware configuration decisions.

Contributions. We develop a model-based automated approach for configuring software embedded in ICSs. Our approach can (1) capture complex variabilities and dependencies that exist in embedded systems, (2) interactively guide users to make configuration decisions and automate some of the decisions, and (3) iteratively validate software and hardware configuration consistency. We evaluated our approach by applying it to a subsea production system. Our experiments show that our approach can automatically infer up to 50% of configuration decisions for the subjects in our experiment, therefore helping save significant configuration effort and avoid configuration errors. It reduces the valid domain of configuration parameters by 40% by dynamically recomputing them, and does this in an efficient manner, thus simplifying configuration decisions.

Organization. In Section 2 we first motivate the work and formulate the problem by explaining the current practice in configuring ICSs. We give an overview of our model-based solution in Section 3. Our SimPL methodology [6, 7] for modeling families of ICSs is briefly presented in Section 4. In Section 5, we present our model-based approach to solve configuration problems by automating and interactively guiding the configuration. An implementation of our approach as a prototype tool is presented in Section 6. An evaluation of the approach using our prototype tool is given in Section 7. In Section 8, we analyze the related work. Finally we conclude the paper in Section 9.

2. Configuration of ICSs: Practice and Problem Definition

We motivate our work using configuration scenarios from our industry partner which is a maritime and energy company. Figure 1, shows a simplified model of a fragment of a subsea production system produced by

our industry partner. As shown in the figure, products are composed of mechanical, electrical, and software components. Our industry partner, similar to most companies producing ICSs, has a generic product that is configured to meet the needs of different customers. For example, different customers may require products with different number of subsea Xmas trees.

Product configuration is an essential activity in ICS development. It involves configuration of both software and hardware components. Currently, software and hardware configuration is performed separately in two different departments within our industry partner. In the rest of this paper, whenever clear from the context, we use *configuration* to refer either to the configuration process or to the description of a configured artifact.

The software configuration is performed in a top-down manner where the configuration engineer starts from the higher-level components and determines the type and the number of their constituent (sub)components. Some components are invariant across different products, and some have parameters whose values differ from one product to another. The latter group, known as *configurable components*, may need to be, further, decomposed and configured. The configuration is completed once the type and the number of all the components and the values of their configurable parameters are determined.

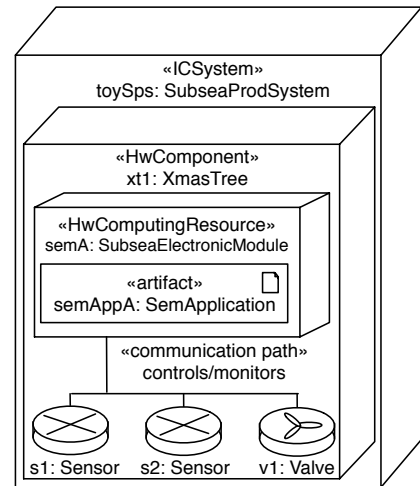


Figure 1: A fragment of a simplified subsea production system.

Currently, the configuration engineer uses an in-house developed tool to add and edit the configuration data. However, all the configuration decisions have to be made manually. Note that this can be time-consuming since ICSs have very large configuration spaces. To configure software, the engineer is typically provided with a hardware configuration plan, but she has to almost entirely rely on her own domain knowledge and experiences to ensure that the resulting software configuration conforms to the given hardware plan, and that it respects all the software consistency rules as well. The software-hardware dependencies and most of the internal software consistency rules are not checked by the existing tool. As our initial investigation showed, configuration is a complicated and large-scale process in our industry partner, and can be error-prone without appropriate automation support [7].

In short, the existing configuration support at our industry partner faces the following challenges: (1) Consistency between hardware and software is not verified and enforced by the configuration tool. Some of these consistencies are, however, tested and verified at unit level according to standard test procedures before the integration testing phase, in which software, configuration, and network are tested all together. (2) Partially-specified configurations cannot be verified. Only complete configurations can be checked against a subset of consistency rules. (3) Engineers are not provided with sufficient interactive guidance and automation from the tool, though recently our industrial partner has tried to using configuration file templates to guide configuration engineers throughout its complicated configuration process, which has been proven effective to certain extent. In our previous work [7, 6], we proposed a modeling methodology to properly capture and document, among other things, the software-hardware dependencies and consistency rules. In this paper, we build on our previous work to develop an automated guided configuration tool that addresses all the three above-mentioned challenges.

3. Overview of our approach

Figure 2 shows an overview of our automated model-based configuration approach. In the first step, we build a configurable and generic model for an ICS family (the Product-line modeling step). In the second step, the Guided configuration step, we interactively guide users to generate the specification of particular products complying with the generic model built in the first step.

During the product-line modeling step, we provide domain experts with a UML/MARTE-based methodology, called SimPL [7, 6], to manually create a product-line model describing an ICS family. The SimPL methodology enables engineers to create product line models from textual specifications and the scattered domain experts knowledge. These models can then be utilized to automate the configuration process. They include both software and hardware aspects as well as the dependencies among them. The dependencies are critical to effective configuration. Currently, most of these dependencies exist as tacit knowledge shared by a small number of domain experts, and only a fraction of them, mostly those related to software, have been implemented in the existing tool used by our industrial partner. Our domain analysis [7, 6], however, showed that failure to capturing all the dependencies have led to critical configuration errors. We briefly describe and illustrate the SimPL methodology in Section 4.

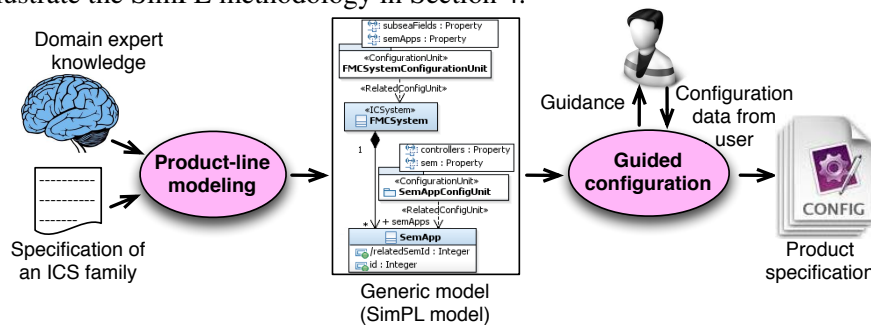


Figure 2: An overview of our configuration approach.

During the configuration step, engineers create full or partial product specifications by resolving variabilities in a product-line model. In our work, configuration is carried out iteratively, allowing engineers to create and validate partial product specifications, and interactively, guiding engineers to make decisions at each iteration. Therefore, our approach alleviates two shortcomings of the existing tool discussed in Section 2. Our configuration mechanism enables engineers to resolve variabilities in such a way that all the constraints and dependencies are preserved. At each iteration, the engineer resolves some of the variabilities by assigning values to selected configurable parameters. Our configuration engine, which is implemented using a constraint solver, automatically evaluates the engineer’s decisions and informs her about the impacts of her decision on the yet-to-be-resolved variabilities, hence, guiding her to proceed with another round of configuration. In Sections 5 and 6, we describe in details how the configuration step is designed and implemented, respectively.

4. Product-line modeling

In the first step of our approach in Figure 2, we use the SimPL modeling methodology [7, 6] to create a generic model of an ICS family. The SimPL methodology enables engineers to create architecture models of ICS families that encompass, among other things, information about variability points in ICS families.

The SimPL methodology organizes a product-line model into two main views: the *system design view*, and the *variability view*. The system design view presents both hardware and software entities of the system and

their relationships using the UML class diagram notation [1]. Classes, in this view, represent hardware or software entities. To distinguish them, we use MARTE stereotypes [2]. The dependencies and constraints that cannot be expressed in class diagrams are captured by OCL constraints [4]. The variability view, on the other hand, captures the set of system variabilities using a collection of *template* packages. Each template package represents a *configuration unit* and is related to exactly one class in the system design view. Template parameters of each template package in the variability view are related to the configurable properties of the class related to that package. Template packages and template parameters are inherent features in UML and are intended to be used for the specification of generic structures. In the remainder of this section, we first describe a small fragment of a subsea product-line model, which is used as our running example. Then, using our running example, we provide a model-based view on the essential configuration activities mentioned in Section 2.

4.1. A subsea product-line model

Figure 3 shows a fragment of the SimPL product line model for a subsea production system¹, represented by the class `SubseaProdSystem`. In a subsea production system, the main computation resources are the Subsea Electronic Modules (SEMs), which provide electronics, execution platforms, and the software required for controlling subsea devices. SEMs and Devices are contained by XmasTrees. Devices controlled by each SEM are connected to the electronic boards of that SEM. The electronic boards are categorized into four different types based on their number of pins. Software deployed on a SEM, referred to as SemAPP, is responsible for controlling and monitoring the devices connected to that SEM. SemAPP is composed of a number of DeviceControllers, which is a software class responsible for communicating with, and controlling or monitoring a particular device. The system design view in Figure 3 represents the elements and the relationships we discussed above.

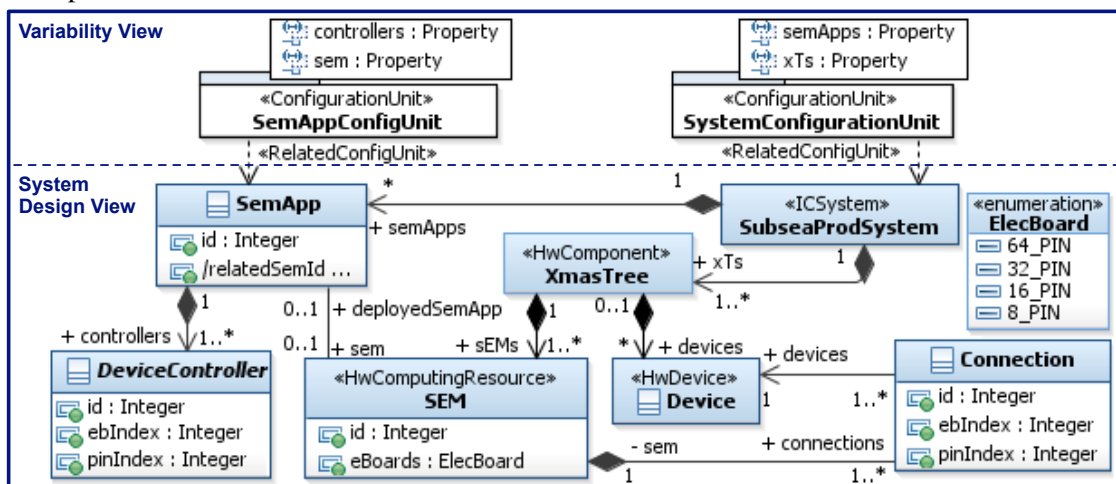


Figure 3: A fragment of the SimPL model for the subsea production system.

The variability view in the SimPL methodology is a collection of template packages. The upper part in Figure 3 shows a fragment of the variability view for the subsea production system. Due to the lack of space we have shown only two template packages in the figure. As shown in the figure, the package `SystemConfigurationUnit` represents the configuration unit related to the class `SubseaProdSystem` in the

¹The example product line is a sanitized fragment of a subsea production case study. The complete model is available at [7].

system design view. Template parameters of this package specify the configuration parameters of the subsea production system, which are: the number of XmasTrees, and SEM applications (`semApps`). Some of the other configurable parameters in Figure 3 are: the number and type of device controllers in a `SemAPP` as shown in the template package `SemAppConfigUnit` using the template parameter `controllers`, the number of SEMs and devices in a `XmasTree`, etc.

As mentioned earlier, the SimPL model may include OCL constraints as well. Two example OCL constraints related to the model in Figure 3 are given below.

```
context Connection inv PinRange
self.pinIndex >= 0 and self.sem.eBoards->asSequence()->
  at(self.ebIndex+1).numOfPins > self.pinIndex

context Connection inv BoardIndRange
self.ebIndex >= 0 and self.ebIndex < self.sem.eBoards->size()
```

The first constraint states that the value of the `pinIndex` of each device-to-SEM connection must be valid, i.e., the `pinIndex` of a connection between a device and a SEM cannot exceed the number of pins of the electronic board through which the device is connected to its SEM. The second constraint specifies the valid range for the `ebIndex` of each device-to-SEM connection, i.e., the `ebIndex` of a connection between a device and a SEM cannot exceed the number of the electronic boards on its SEM.

4.2. Configuration activities in a model-based context

As mentioned in Section 2, configuration involves a sequence of two basic activities: (1) specifying the type and the number of (sub)components, and (2) determining the values for the configurable parameters of each component. We ground our configuration approach on the SimPL methodology and state these activities in modelling terms as follows: (1) creating instances for classes that correspond to configurable components, and (2) assigning values to the configurable parameters of those instances. For example, to configure the subsea system in Figure 3, we need to first create instances of `XmasTree`, `SEM`, `Device`, and `SemApp`, and then assign appropriate values to the configurable variables of these instances. Note that value assignment may imply instance creation as well. Specifically, a configurable parameter can represent the cardinality of an association. Assigning a value to such a parameter automatically implies creation of a number of instances to reach the specified cardinality.

5. Interactive model-based guided configuration

The outcome of the configuration step in Figure 2 is a (possibly partial) model of a product that is *consistent* with the SimPL model describing the product family to which that product belongs. In our approach, SimPL models are described using class-based models, while the product models are object-based. A product model is consistent with its related SimPL model when:

- Each object in the product model is an instance of a class in the SimPL model.
- Two objects of types C_1 and C_2 are connected only if there is an association between classes C_1 and C_2 in the SimPL model.

- The object model satisfies the OCL constraints of the SimPL model.

The above *consistency rules* are invariant throughout our configuration process, i.e., they hold at each configuration iteration even when the product model is defined partially. In this section, we first describe how our approach guides the user at each configuration iteration while ensuring that the above rules are not violated. We then demonstrate how a constraint solver can be used to maintain the consistency rules throughout the entire configuration process, and to automatically perform some of the configuration iterations.

5.1. Guided and automated configuration

The product configuration process is a sequence of *value-assignment* steps. At each step, a value is assigned to one *configurable parameter*. A configurable parameter can represent (1) a property in an instance of a class, (2) the size of a collection of objects in an instance of a class, or (3) the concrete type of an instance.

A *configuration* is a collection of value-assignments, from which a full or partial product model can be generated. A configuration is complete when all the configurable parameters are assigned a specific value, and is partial otherwise. Each configurable parameter has a *valid domain* that identifies the set of all values that can be assigned to that configurable parameter without violating any consistency rule. Below, we describe the guidance information that our tool provides to the user at each iteration of the configuration process.

Valid domains. At each iteration, the tool provides the user with the valid domains for all the configurable parameters. Such domains are dynamically recomputed given previous iterations. The values that the user provides should be within these valid domains, or otherwise, the user's decision is rejected and he receives an error message. For example, the valid domain for the configurable parameter `pinIndex` is initially 0..63. Therefore, if a user assigns to this parameter a value outside 0..63 his decision will be rejected.

Decision impacts. If the user's decision is correct, the decision is propagated through the configuration to identify its impacts on the valid domains of other configurable parameters. This may result in pruning some values from the valid domains of some configurable parameters. For example, the valid domain for the type of an `eBoard` in a `SEM` is initially `{8_PIN, 16_PIN, 32_PIN, 64_PIN}` (the set of all literals in the enumeration `ElecBoard`). If a user configures a `Connection` in a `SEM` by assigning 2 to `eblIndex`, and 13 to `pinIndex`, then according to the OCL invariant `PinRange` (defined above), the third `eBoard` in that `SEM` must at least have 14 pins. Therefore, such a value-assignment removes `8_PIN` from the valid domain of the type of the third `eBoard`, resulting in the pruned valid domain `{16_PIN, 32_PIN, 64_PIN}`.

The impacts of the decisions are then reported to the user, in terms of reduced valid domains. In some cases, the impacts may result in automatic value-assignments that are also reported to the user.

After value-assignment propagation and pruning, the tool checks if the size of any valid domains is reduced to one. The configurable parameters with singleton valid domains are set to their only possible value. This enables automatic inferences of values for some configurable parameters, therefore, saving a number of value-assignment steps from the user. For example, in Figure 3 there is a one-to-one deployment relationship between `SEM` and `SemApp`. As a result, whenever the user creates a new instance of `SEM` the tool automatically creates a new instance of `SemApp` and correctly configures in it the cross-reference to the `SEM`. Inferring a value for a configurable parameter that represents the size of an object collection, is followed by automatically creating and adding to that collection the required number of objects.

5.2. Constraint satisfaction to provide guidance and automation

The main computation required for providing the aforementioned guidance and automation is the calculation of valid domains through pruning the domains of all the yet-to-be-configured parameters after each configuration iteration using the user's configuration decision.

In our approach, we use a constraint solver over finite domains to calculate the valid domains. In this approach, the configuration space of a product family forms a *constraint system* composed of a set of variables, x_1, \dots, x_n , and a set of constraints, \mathcal{C} , over those variables. Variables represent the configurable parameters, and get their values from the *finite domains* $\mathcal{D}_1, \dots, \mathcal{D}_n$. A finite domain is a finite collection of tags, that can be mapped to unique integers. We extract the finite domains of variables from the types of the configurable parameters, enumerations, multiplicities, and OCL constraints in the SimPL model. The constraint set \mathcal{C} includes both the OCL constraints and the information, e.g., multiplicities, extracted from the class diagrams in the SimPL model. A configuration in this scheme corresponds to a (possibly partial) evaluation of the variables x_1, \dots, x_n . Using a constraint solver the consistency of a configuration w.r.t the constraint set \mathcal{C} is checked, and the valid domains, D^*_1, \dots, D^*_n , for all the variables are calculated.

At each value-assignment step during the configuration, a value v_i is assigned to a variable x_i . This value assignment forms a new constraint $c : x_i = v_i$, which is added to the constraint set \mathcal{C} . The added constraint is then propagated throughout the constraint system to identify the impacts of the assigned value on other variables, and to prune and update the valid domains of those variables. This process is realized through a simple and efficient Constraint Programming technique called *constraint propagation* [15]. Constraint propagation is a monotonic and iterative process. During constraint propagation, constraints are used to filter the domains of variables by removing inconsistent values. The algorithm iterates until no more pruning is possible. Algorithm 1 shows the constraint propagation algorithm.

Algorithm 1 Constraint Propagation Algorithm (AC-3)

Input: σ a set of constraints, D a set of finite domains

Output: D' partially-consistent domains

```

queue  $\leftarrow \sigma$ 
while queue  $\neq \emptyset$  do
  c  $\leftarrow$  pop(queue)
   $D' \leftarrow$  narrow(c, D) {Filtering}
  if  $D' \neq D$  then
    queue  $\leftarrow$  queue  $\cup \{c' \in \sigma \mid \text{vars}(c') \cap \text{vars}(c) \neq \emptyset\}$ 
     $D \leftarrow D'$ 
  end if
end while
return  $D'$ 

```

Assigning a value to a variable representing the size of a collection relates to adding items to, or removing items from the collection. Adding an item to a collection implies introducing new variables to the constraint system. Similarly, removing items from a collection implies removing variables from the constraint system. As a result, to identify the impacts of changing the size of a collection, new variables have to be added or removed during constraint propagation. This is possible as constraint propagation does not require the set of initial variables to be known a priori. However, the process is no longer monotonic in that case and may iterate forever. Hopefully, in our application, the number of added variables is always bounded, avoiding any non-termination problems.

In the following section, we further elaborate on the design of a tool implementing the configuration process presented above.

6. Prototype tool

Figure 4 shows the architecture of the configuration engine that provides the guidance and automation mentioned in Section 5. Inputs to the engine are the generic model of the product family, and the user-provided configuration data. The configuration process starts by loading the generic model. From the loaded model, the configuration engine extracts the first set of the configurable parameters. These configurable parameters are presented to the user via the interactive user interface for collecting configuration decisions. In addition, the configuration engine generates a constraint model from the input model of the product family. This constraint model is implemented in `clpfd`, a library of the *SICStus Prolog* environment [10, 5]. In `clpfd`, each configurable parameter is represented by a logic variable, to which is associated a finite set of possible values, called a finite domain. After the generic model is loaded, the configuration engineer starts an interactive configuration session for entering configuration decisions.

The configuration engine iteratively and interactively collects configuration decisions from the user. At each iteration, the user enters the values for one or more configurable parameters. Using the domains of the configurable parameters, the consistency of the configuration decisions is checked. If the entered values are all consistent, the *Query generator* is invoked to create a new Prolog query representing a constraint system that contains all the constraints created from the collected configuration decisions. This Prolog query is then used to invoke constraint propagation in order to prune the domains. The new domains serve as inputs to the *Inference engine* and the *Guidance provider*. The inference engine infers values for configuration parameters while the guidance provider reports the impacts of configuration choices (e.g., updated domains).

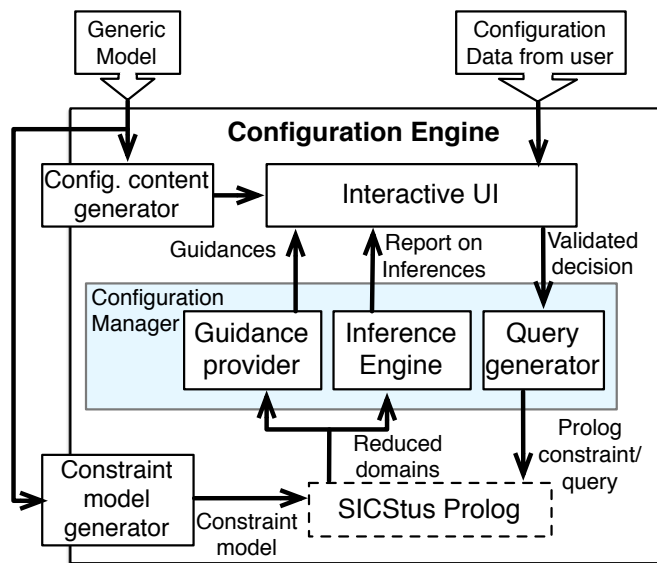


Figure 4: Architecture of the configuration tool.

6.1. The `clpfd` library of *SICStus Prolog*

Choosing Prolog as a host language for developing our configuration engine has several advantages. First, Prolog is a well-established declarative and high-level programming language, allowing fast prototyping for building a proof-of-concept tool, and containing all the necessary interfaces to widely-used programming languages such as Java or C++. In our tool development, we have used the `jasper` library that allows invoking the *SICStus Prolog* engine from a Java program. Second, as it embeds a finite domains constraint solver through the `clpfd` library, this allows us to benefit from a very efficient implementation of constraint propagation [11], and all the available constructs (e.g., combinatorial constraints) that have been proposed for handling other applications.

6.2. Mapping to clpfd

Reasoning about embedded software systems using the finite domains constraint engine of SICStus Prolog requires translating their specifications into `clpfd`. The required translation is done in two steps. First, we create a Prolog/`clpfd` program from the input SimPL model that is composed of UML classes, their relationships, and OCL constraints constraining them. Each *rule* in the Prolog/`clpfd` program represents an OCL expression (or subexpression). Each rule evaluates the corresponding OCL expression for a given collection of objects. Each object is specified using a list in the Prolog/`clpfd` program, and each element in the list represents a single property of the object. If the property is not yet evaluated, it appears in the list as a logic variable. Object collections are, hence, mapped to lists of lists in Prolog. For example, the following Prolog code represents the rule associated with the second OCL constraint given in Section 4.

```

all_e_board_ind_in_range(_, _, []).
all_e_board_ind_in_range(AI, Ids, [Obj0|Rest]) :-
    one_e_board_in_range(AI, Ids, Obj0),
    all_e_board_ind_in_range(AI, Ids, Rest).
one_e_board_in_range(AI, _, [[5],_,[OnrId],_,_, [BInd],_,_,_]) :-
    BInd #>= 0, ...,
    get_object(AI, OnrId, [_,_,_,EBoards, _, _, _, _]),
    length(EBoards, L), ...,
    BInd #< L.

```

In the second stage of the transformation, Prolog queries are created from product specifications. A product specification is a collection of instances and links connecting them. A query to SICStus Prolog looks like `check_product(Als, Ids)`, where `Als` is the list representation of all instances, and `Ids` is the list of the identifiers of instances. The Query provider in our tool is responsible for generating these two lists from the instances created and configured by the user. Given the query `check_product(Als, Ids)`, the constraint engine checks whether the product specification specified by `Als` and `Ids` represents a valid instantiation of the input SimPL model, and if so, it provides the valid domains for all the variables in `Als`. Note that the calculation of the valid domains terminates because `Als` contains a finite number of variables (as the number of the instances in the product are finite), and all variables take their values from finite domains.

A detailed specification of our mapping is outside the scope of this paper. However, similar to our mapping, several mappings from UML/OCL to Prolog exist in the literature (e.g., [9, 19]), and we refer the interested reader to them.

7. Evaluation

To empirically evaluate our approach, we performed several experiments which are reported in this section. The experiments are designed to answer the following three main research questions:

1. What percentage of the value-assignment steps can be saved using our automated configuration approach?
2. How much do the valid domains shrink at each iteration of configuration?
3. How long does it take to propagate a user's decision and provide guidance?

Saving a number of value-assignment steps is expected to reduce the configuration effort, and reduction of the domains decreases the complexity of decision making. Therefore, answers to the first two research questions provide insights on how much configuration effort can be saved. Answering the third research question provides insights into the applicability and scalability of our technique.

To answer these questions we designed an experiment in which we rebuilt three verified configurations from our industry partner using our configuration tool. One configuration belongs to the environmental stress screening (ESS) test of the SEM hardware, which we refer to in this section as the ESS Test. The other two are the verified configurations of two complete products, which we refer to in this section as Product_1 and Product_2. Table 1 summarizes the characteristics of these configurations. We performed our experiments using the simplified generic model of the subsea product family given in Section 4. Number of objects and variables in Table 1 are calculated w.r.t that simplified model.

Table 1: Characteristics of the rebuilt configurations.

	# XmasTrees	# SEMs	# Devices	# Objects	# Variables
ESS Test	1	1	111	226	343
Product_1	9	18	453	1396	2830
Product_2	14	28	854	2619	5307

We report in Sections 7.1-7.3 the evaluation and analysis that we performed on the experiments to answer the above research questions. At the end of this section, we also discuss some limitations, directions for future work, and the generalizability of our approach.

7.1. Inference percentage

The configuration effort required for creating the configuration of a product is expected to be proportional to the number of configuration iterations and the number of value-assignment steps. Automating the latter is therefore expected to save configuration effort and minimize chances for errors. To measure the effectiveness of our approach in reducing the number of value-assignment steps, we have defined an *inference rate* which is equal to the number of inferred decisions divided by the total number of decisions:

$$inference\ rate = \frac{inferences}{manual_decisions + inferences} \tag{1}$$

Table 2 shows the inference rates in each case.

Table 2: Inference rates.

	# Manual decisions	# Inferred decisions	Inference rate (%)
ESS Test	373	16	4.11
Product_1	1459	1426	49.42
Product_2	2802	2783	49.82

Note that the inference rate for Product_1 and Product_2 is very close to 50 %. This is because of the *structural symmetry* that exists in the architecture of the system. Structural symmetry is achieved in a product when two or more components of the system have identical or similar configurations. We have modeled the structural symmetries using two OCL constraints. One specifies that each XmasTree has two SEMs (*twin SEMs*) with identical configurations (i.e., identical number and types of electronic boards and devices connected to them). The other specifies that all the XmasTrees in the system have similar configurations (e.g., all have the same number and types of devices). The first OCL constraint applies to both Product_1 and Product_2, while the second applies to Product_2 only. As a result, the inference rate for Product_2 is slightly higher than that for Product_1. Neither of the OCL constraints applies to the ESS Test, which contains only one XmasTree and one SEM. Therefore, it shows a very low inference rate. In general, the

Table 3: Average shrinking of the domains.

	Count*	Avg. initial domain size	Avg. reduction size	Avg. reduction rate (%)
ESS Test	732	30.557	13.803	45.17
Product_1	2564	62.125	21.367	34.39
Product_2	7557	35.97	14.205	39.49
Avg. over all cases:				37.98
* total number of domains that have been pruned or reduced.				
Avg.: the average over all reduced domains in the whole configuration.				

architecture of the product family, and characteristics of the product itself (e.g., structural symmetry) can largely affect the inference rate.

Our experiment shows that our approach can automatically infer a large number of consistent configuration decisions specially for products with some degree of structural symmetry. Assuming automated value-assignments have similar complexity to manual ones, our approach can save about 50% of the configuration effort of Product_1 and Product_2.

7.2. Reduction of valid domains

Pruned domains are the output of constraint propagation. Pruning of the domains decreases the complexity of decisions to be made. As part of our experiment, we measured how the domains shrink after each constraint propagation step. Such reduction of the domains is measured by comparing the size of each pruned domain before and after constraint propagation. This is possible and meaningful because all the domains are finite. Table 3 shows the average reduction of domains for each case. *Reduction rate* in the table is defined as the proportion of the *reduction size* (i.e., number of distinct values removed from a domain) to the initial size of the domain (i.e., the number of distinct values in a domain). In the calculations in Table 3 we have not considered domain reductions that resulted in inferences. This result shows that the domains of variables can be considerably reduced when a value is assigned to a dependent variable. Specifically, it shows that, on average, after each value-assignment step 37.98% of the values of the dependent variables are invalidated. Without such a dynamic recomputation of valid domains, there would be a higher risk for the user to make inconsistent configuration decisions. Moreover, comparing the inference rate from Table 2 and the reduction rate from Table 3 over the three cases suggests that while structural symmetry can highly affect the inference rate, it does not have a large impact on the reduction rate.

7.3. Constraint propagation efficiency

Providing automation and guidance as part of the interactive configuration process requires the underlying computation to be sufficiently efficient for our approach to be practical.

We define the efficiency of our approach as the amount of time needed for validating and propagating the user decision. For this purpose, we have measured at each constraint propagation step the execution time, and the number of variables in the constraint system. Figure 5 shows the average time required for propagating user decisions after each value-assignment step. As shown in this figure, for products with less than 1000 variables, it takes, on average, less than one second to validate and propagate the decision. However, this time grows polynomially with the number of variables, which itself is proportional to the number of instances.

Since in our experiment we have used a simplified model of the product family, we expect that for a complete

model of the system the number of instances and the number of variables be much higher than that in this experiment. However, our experiment shows that not all of these variables are dependent on each other. To provide an insight into the level of dependency between variables, for each case, we can compute the average number of reduced domains. The average number of reduced domains is 1.8 (2564 from Table 3 divided by 1459 from Table 2) for Product_1 and, 2.7 for Product_2. In other words, on average, each variable in Product_1 (Product_2) is dependent to less than two (three) other variables. The polynomial ($O(n^2)$) growth of the execution time is, however, due to our current implementation, in which, we compute the valid domains of all variables (not only the dependent variables) by creating a new constraint propagation session after each value-assignment step. Therefore, we expect that by optimizing our implementation and incrementally adding new constraints to an existing constraint propagation session we can significantly improve the efficiency of our approach. Such an optimization requires an additional preprocessing step before creating queries and invoking the constraint solver. This needs to be investigated in more depth and is left for future research.

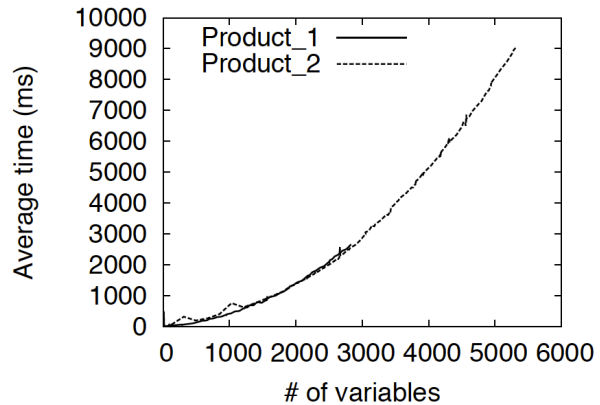


Figure 5: Constraint propagation time grows quadratically with the number of variables (with a coefficient of determination of 0.9994).

7.4. Discussion

Limitations and directions for future work. The inference rate and the reduction rate, in addition to be affected by the architecture of the product family, are affected by the order in which the decisions are made. An *optimal order* of applying configuration decisions can be defined as the order which can result in the maximum inference rate and reduction rate. The optimal order can be reported to the user as additional guidance. Our current implementation does not provide such a guidance and therefore the results reported in this paper are probably, a lower bound for potential configuration effort savings. It is therefore important that in the future we support the optimization of the ordering to maximize inferred decisions and the reduction of domains. Devising criteria and heuristics for finding such optimal order is one direction of our future work.

Another research question is "How useful is the guidance provided by our approach?". Answering this question requires conducting an experiment involving human subjects. This experiment is also part of future work.

Generalizability of our approach. Like any other model-based engineering approach, the effectiveness of our approach depends on the quality of the input generic models. Our configuration approach can be used to configure only the variabilities that are captured in the generic model of the product family. Similarly, the approach can validate the decisions and automatically infer decisions only based on the dependencies that are captured in the model. Our evaluation in this paper shows that the SimPL methodology and notations that we proposed in [7, 6] enables the creation of models of the required quality.

The use of a constraint solver over finite domains limits our approach to the constraints that capture restrictions on variables with finite domains. Constraint solvers over continuous domains are available to overcome this limitation but their integration with an efficient finite domains solver is still an open research problem [12]. Moreover, as we have not encountered this type of constraint with our industry partner, we don't expect this to be a restriction in our context.

8. Related Work

Most of the existing work on constraint-based automated configuration focuses on resolving variabilities specified by feature models [16] and their extensions [14]. Basic feature models cannot express complex variabilities or dependencies required for configuring embedded systems [7, 6]. However, extended feature models that allow attributes, cardinalities, references to other features, and cloning of features are, as mentioned in [13], as expressive as UML class diagrams and can be augmented by OCL or XPath queries to describe complicated feature relationships as well.

We compare our work with the existing automated configuration and verification tools proposed for extended feature models since these are the closest to our SimPL models. FMP [13] is an Eclipse plug-in that enables creation and configuration of extended feature models. FMP can verify full or partial configurations for a subset of extended feature models, specifically those with boolean variables and without clonable features. FAMA [8] drops this limitation and can verify extended feature models with variables over finite domains. However, FAMA is more targeted towards the verification and analysis of feature models. Therefore, it does not handle validating partial configurations or help build full configurations iteratively. Finally, Mazo et. al. [17] use constraint solvers over finite domains to analyze extended feature models. This approach is the closest to ours as it can handle all the advanced constructs in extended feature models, and further enables verification of full and partial configurations.

The main limitation of all of the above approaches is that none of them supports verification and analysis of complex constraints such as those in Section 4.1. These constraints express complex relationships between individual elements or collections of elements and are instrumental in describing software/hardware dependencies and consistency rules in embedded systems. Our tool, in addition to verifying these constraints, provides interactive guidance to help engineers effectively build configurations satisfying these constraints. Finally, to the best of our knowledge, none of the above approaches have been applied to nor evaluated on real case studies.

9. Conclusion

In this paper, we presented an automated model-based configuration approach for embedded software systems. Our approach builds on generic models created in our earlier work, i.e., the SimPL models, and uses constraint solvers to interactively guide engineers in building and verifying full or partial configurations. We evaluated our approach by applying it to a real subsea production system where we rebuilt three verified configurations of this system to evaluate three important practical factors: (1) reducing configuration effort, (2) reducing possibility of human errors, and (3) scalability. Our evaluation showed that, in our three example configurations, our approach (1) can automatically infer up to 50% of the configuration decisions, (2) can reduce the size of the valid domains of the configurable parameters by 40%, and (3) can evaluate each configuration decision in less than 9 seconds.

While our preliminary evaluations demonstrate the effectiveness of our approach, the value of our tool is likely to depend on its scalability to very large and complex configurable systems. In particular, being an interactive tool, its usability and adoption will very much depend on how fast it can provide the guidance information at each iteration. Our current analysis shows that the propagation time grows polynomially with the size of the product. But we noticed in our work that after each iteration only a very small subset of variables are affected. Therefore, if we could reuse the analysis results from the previous iterations, we could possibly improve the time it takes to analyze each round significantly.

Acknowledgments

This work was mainly supported by a grant from Det Norske Veritas (DNV). We are grateful to FMC Technologies Inc. for their support and help on performing the industrial case study.

References

- [1] UML Superstructure Specification, v2.3, May 2010.
- [2] Marte. <http://www.omgarte.org/>, 2012.
- [3] Modbus. <http://en.wikipedia.org/wiki/Modbus>, 2012.
- [4] Object Constraint Language. <http://www.omg.org/spec/OCL/2.2/>, 2012.
- [5] Sicstus Prolog Homepage. www.sics.se/sicstus/, January 2012.
- [6] R. Behjati, T. Yue, L. Briand, and B. Selic. SimPL a product-line modeling methodology for families of integrated control systems. Submitted to Information and Software Technology Journal, 2011.
- [7] R. Behjati, T. Yue, L. Briand, and B. Selic. SimPL a product-line modeling methodology for families of integrated control systems, Tech. Report 2011-14, Simula Research Lab. <http://simula.no/publications/Simula.simula.746>, 2011.
- [8] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz Cortés. Fama: Tooling a framework for the automated analysis of feature models. In *VaMoS*, 2007.
- [9] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of uml/ocl class diagrams using constraint programming. pages 73–80, Washington, DC, USA, 2008. IEEE Computer Society.
- [10] Mats Carlsson and Per Mildner. Sicstus prolog – the first 25 years. *CoRR*, abs/1011.5640, 2010.
- [11] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. PLILP '97, 1997.
- [12] H el ene Collavizza, Michel Rueher, and Pascal Van Hentenryck. A constraint-programming framework for bounded program verification. *Constraints Journal*, 2010.
- [13] K. Czarnecki and P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Proceedings of the International Workshop on Software Factories at OOPSLA*, 2005.
- [14] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. In *Software Process: Improvement and Practice*, 2005.
- [15] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). In *Selected Papers from Constraint Programming: Basics and Trends*, 1995.
- [16] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, 1990.
- [17] Ra ul Mazo, Camille Salinesi, Daniel Diaz, and Alberto Lora-Michiels. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In *ENASE*, 2011.

- [18] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [19] Harald Störle. A prolog-based approach to representing and querying software engineering models. In *Proceedings of the VLL*, 2007.