Model-driven Configuration of Function Net Families in Automotive Software Engineering

Cem Mengi, Önder Babur, Holger Rendel, and Christian Berger

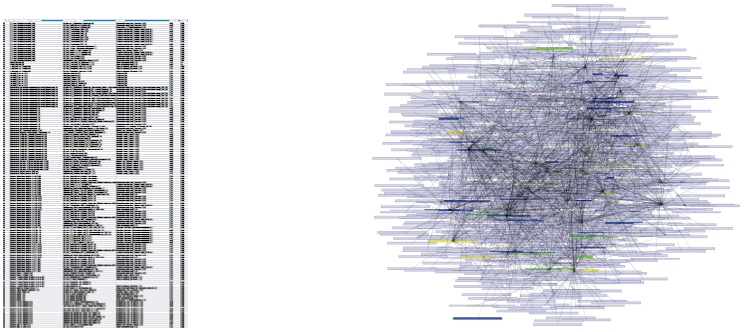
Software Engineering
RWTH Aachen University, Germany
http://www.se-rwth.de/

Abstract. Recent efforts in the automotive domain to initiate a paradigm-shift from a traditional hardware-driven to a function-driven development process create new challenges to tackle. A hardware-driven variant handling mechanism will get more and more inappropriate. Instead, new concepts and methods are necessary to model and configure concrete systems. A software document which is used in the early phase of development is the so called function net model. Variability in function nets is captured implicitly and is strongly dependent on the hardware infrastructure, constraints are collected with informal annotations, and variants are generated manually. This results in a situation where function nets get too complex, time consuming and unsuitable for future standards. In this paper, we will present a model-driven approach for function nets to capture variability explicitly, to express formal constraints, and to generate concrete variants with the support of an automated configuration process. By this, it is possible to use the generated variants as skeletons for virtual prototyping, so that requirement specifications can be verified efficiently.

1 Introduction and Motivation

Variability in automotive software engineering has achieved a degree of complexity which can no longer be handled with traditional hardware-driven methodologies [1–3]. *Function net models* are software documents which are used in an early phase of such a development process to describe the first virtual realization of the system structure [4]. Typically, it consists of *functional modules* that interact via *signals*.

Variation points and their *variants* are captured implicitly by modeling a so called *maximal function net model*. The idea behind that is to capture every functional module, e.g., a sensor, an actuator, a control algorithm etc. In this way, a function net corresponds to an Electronic Control Unit (ECU) together with all its physical connections to bus systems, sensors, and actuators. Figure 1 illustrates two forms of a maximal function net model. Figure 1(a) represents a function net model specified in an Excel sheet, while Figure 1(b) is a visualization of this Excel sheet as graph. In both forms, it is very difficult or nearly impossible to extract useful information manually out of it.



(a) Function net model as Excel sheet (cutout).

(b) Function net model as graph.

Fig. 1. A maximal function net model illustrated in two different forms.

Communication dependencies between function nets, i.e., communication dependencies between ECUs, which have an influence on variability are handled by using informal annotations. Variants are then generated manually by removing specific parts of the function net with the help of these annotations. In a hardware-driven process, this can be regarded equally to removing hardware components such as ECUs, sensors, or actuators and the appropriate software portions. Any implications to other hardware and software components have to be solved manually.

Regarding the current trend in automotive software engineering, we can observe a paradigm-shift from a hardware-driven process to a function-driven process [5]. Therefore, a hardware-driven variant handling mechanism will get more and more inappropriate, because it gets too complex and is not suitable for future standards: it completely *depends on the underlying hardware infrastructure*, consistency has to be ensured manually, i.e., there is *no formalization to express dependencies*, and there is also *no automated support to configure and validate concrete variants*.

In a research project at our department, we are providing new concepts, methods, and tools for handling complexity and variability in software documents for automotive software engineering [6–9]. We have developed a *formal language to model function nets*, and a *formal language to model variation points and variants*. Furthermore, we have *integrated* these two modeling languages in order to reduce synchronization overhead. *Modeling guidelines*, some of which can be checked on syntactical level and others which completely rely on behalf of the function net architect, are also provided. Moreover, *methodology rules* for the usage of such an integrated modeling language are also specified. What is still missing is the possibility to express *constraints* between functional modules or its signals inside one or many function net models. Furthermore, there is also a need to have an *automated support to configure concrete variants* out

of the family which also ensures the consistency. Because we are not only dealing with function net models but with several software documents, we need a *configuration engine which is independent from the input language.*

Including the missing concepts will enhance the development process in a way, that out of the family of function net models, we can generate skeleton variants for virtual prototypes (i.e., behavioral simulation on a PC). This allows an early and efficient verification of requirement specifications.

In this paper, we will introduce an approach to express constraints in and between function net models. Furthermore, a configuration process is integrated which is coupled with an independent back-end inference engine. The output of the configuration process is a concrete variant of a function net.

The paper is structured as follows: Section 2 describes and motivates the problems in more detail. Here, we also illustrate an example which is used continuously in the whole paper. Section 3 and 4 include the main contribution of this paper. Here, we explain the language to express constraints and the whole configuration process. Section 5 describes related work and finally Section 6 concludes this paper.

2 Problem Description by Example

This section describes the initial situation of our approach that is relevant to explain the problems we are going to tackle and the contribution of this paper. Nevertheless, we will not describe the conceptual background but illustrate an example and refer to them appropriately.

Figure 2 shows an example of a function net model integrated with a variability model [6, 8]. It illustrates a part of a `Central Locking System (CLS)`, with a `Vehicle Access Controller (VAC)`, a `Data Transceiver (DT)`, and a `Central Locking Master (CLM)`. A `VAC` can `lock (l)` or `unlock (ul)` the vehicle. For this, it has to authorize itself with some `authentication data (ad)`. To communicate between functional modules we use *ports*. Thereby, we distinguish between *data ports*, i.e., signals with a data storage characteristic, and *control ports*, i.e., signals initiating a functionality. In Figure 2 data ports are visualized with a rectangular shape, while control ports are visualized with a circle shape. Furthermore, these two types of ports are distinguished between *provided* and *required* ports. In Figure 2 provided ports are marked black, while required ports are marked white.

Variation points and their variants are captured in the variability model [6, 8]. For example, a `VAC` is a variation point, which has two variants, i.e., a `Remote Control (RC)` and an `Identification Transmitter (IDT)` for a more comfortable access. In the same way, a `DT` is also a point of variation which has for example `Antennas (At)` as variants. A further variant could be a `key slot` (not shown in Figure 2). Variation points have *group cardinalities* and variants have *variant cardinalities* in the same way as *Czarnecki et al.* have introduced [10]. For example, in Figure 2 the group cardinality `[1..2]` of `VAC` expresses that out of the captured variants at least one but at most two variants are needed for

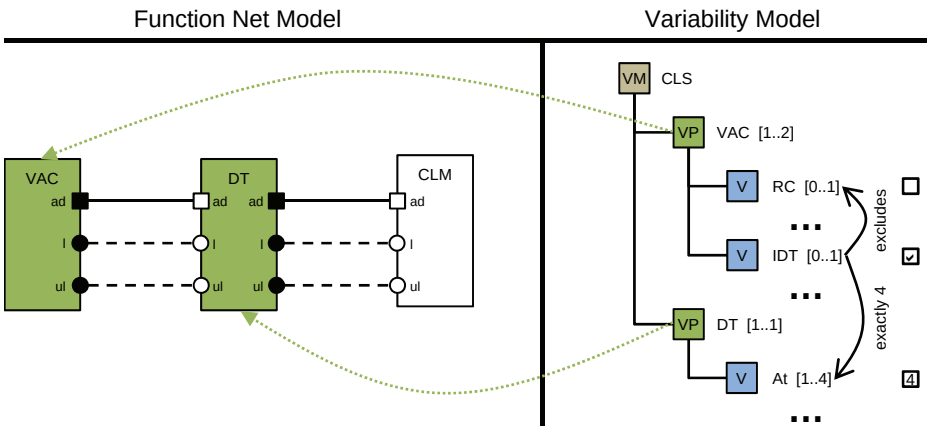


Fig. 2. An example showing the integration of a function net and variability model.

a valid `CLS`. The variant cardinality `[1..4]` of `At` expresses that a valid `CLS` can have at least one but at most four instances of the variant. By using group and variant cardinalities it is possible to restrict valid systems. In the subsequent sections, we will see that this is not enough to express constraints.

Finally, variants can be specified more fine-grained (depicted with the dots in Figure 2). For example, ports and their connections can differ depending on the variant. For our explanations in this paper, we will not need them and therefore, they are neglected here.

As explained before, currently we cannot express more complex constraints between functional modules or its signals inside one or many function nets. Furthermore, an automated support to configure concrete variants is also not possible. The next two subsections will describe the mentioned problems in more detail.

2.1 Expressing Constraints

In Figure 2, we have denoted two constraints for `IDT` (solid black line with arrow), i.e., the exclusion of `RC` if it is selected and the necessity of exactly four `Ats`. Please note that the exclusion can also be modeled with the expression of group cardinalities. This kind of constraints and obviously more complex constraints are often needed when modeling a family of function nets. As mentioned before, in our current prototype it is not possible to express more complex constraints which is a strong limitation of our integrated concept. Therefore, there is a need for an appropriate constraint language in order to express dependencies between variants.

An important requirement is that the language has *sufficient expressiveness*. It should be possible to combine multiple variants and to formulate nested statements. It should also be possible to define constraints between different variability models. Another important requirement is that constraints should be

reusable. Having gained experience in often used constraints, it should be possible to predefine standard constraints and reuse them. Furthermore, the constraint language should *not cause that much time-consuming training effort* for the configuration process. This minimizes error-proneness and enhance usability.

2.2 Configuration of Concrete Variants

Consider again Figure 2. We have depicted check boxes for the variability model to illustrate the configuration process. In the example, `IDT` is selected which requires exactly 4 `Ats`. Obviously, this is a valid configuration from which a concrete variant can be derived. The variant can then be used as skeleton for virtual prototypes to verify the requirements specification. Such a virtual prototype can for example be modeled with tools such as Matlab® Simulink® and Stateflow® [11]. In our current implementation such a configuration process is not supported.

There is a need for back-end configuration engine which infers valid variants. The configuration engine should be *independent from the input language*. This means, that it should be possible to handle different kind of models, such as function net models, Simulink models, and even source code. Furthermore, it should be possible to *combine different models* and process them as input. For this, the configuration process should be easily *extendable* and *highly performant*.

3 Design of Function Net Families with Constraints

Because we do not plan to design a new constraint language, we have investigated literature to find a suitable one that fulfills our requirements and is adaptable to our prototype. As one important requirement, we have identified that the constraint language should not cause additional expenses. Therefore, the selection of our language depends on the input language of the underlying configuration engine. As we will see in Section 4, we will use *smodels* as configuration engine which requires *Weight Constraint Rule Language* (*WCRL*) as input language [12]. *Soininen et. al.* have shown that WCRL has sufficient expressiveness. It is also possible to reuse rules. Therefore, WCRL is a language that is suitable for our purposes. In this paper, we do not show the complete definition of WCRL but limit it to a portion that is enough for the variability and function net model. For the complete definition, please refer to the authors of WCRL [12].

A *weight constraint rule* is of the form

$$C_0 \leftarrow C_1, \ldots, C_n, \tag{1}$$

where C_i is a *weight constraint* which in turn is of the form

$$C_i = L \leq \{a_1 = w_1, \ldots, a_n = w_n\} \leq U, \tag{2}$$

where $L, U \in \mathbb{Z}$ are *lower* and *upper bounds*, a_i is a *literal* (atomic formula or predicate), and $w_i \in \mathbb{N}_0$ is a *weight*.

A weight constraint C_i is satisfied for a set S iff

$$L \leq \sum_{a_i \in S} w_i \leq U, \tag{3}$$

where S is a *set of literals*.

To illustrate WCRL consider again Figure 2. To express the exclusion of `RC` if `IDT` is selected, we can define a weight constraint rule as follows:

$$\texttt{ExcludeRC} \leftarrow \texttt{IsSelectedIDT},$$

where

$$\texttt{ExcludeRC} = -\infty \leq \{\texttt{InstOfType(X, RC) : InstDomain(X)}\} \leq 0,$$

and

$$\texttt{IsSelectedIDT} = 1 \leq \{\texttt{InstOfType(X, IDT) : InstDomain(X)}\} \leq \infty.$$

The expression `InstOfType(X, RC) : InstDomain(X)` returns, out of the whole set of instances, i.e., the selected variants, a subset of a literals `X` which are of type `RC` (and a have default weight of 1). The same holds for `IDT`.

To express that exactly 4 `At`s are needed if `IDT` is selected, we can define another weight constraint rule:

$$\texttt{Exactly4Ats} \leftarrow \texttt{IsSelectedIDT},$$

where `IsSelectedIDT` is defined as above, and

$$\texttt{Exactly4Ats} = 4 \leq \{\texttt{InstOfType(Y, At) : InstDomain(Y)}\} \leq 4.$$

In this way, it is possible to express formal constraints which are also processable by the configuration process which is presented next.

4 Configuration of Function Net Families

A constraint language allows the restriction or precision of the configuration domain. We are now able to model a family of function nets and to express constraints across variants. What is still missing is a methodology and appropriate concepts to configure a concrete variant out of the family. In this section, we will provide such a configuration process and explain the steps relevant for it.

Figure 3 gives an overview of the whole configuration process. The input document will be a function net family as shown in Figure 2 and the output document after configuration will be a comletely configured function net. The configuration steps are (1) the selection of the variants in the variability model. This is done with the help of the integrated prototype [8]. The result of this step will be a function net family plus selected variants. Because WCRL is the input

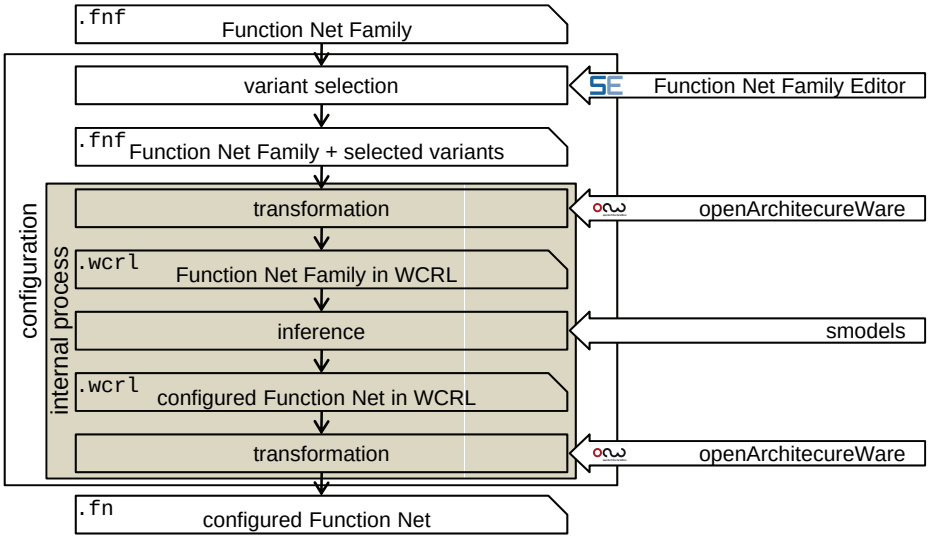


Fig. 3. Overview of the configuration process.

language for the back-end inference engine, we need (2) a transformation from the language of function net families to an equivalent model in WCRL which is the result of this step. The transformation rules are defined with support of openArchitectureWare (oAW) [13]. The function net model in WCRL is the input for the inference engine smodels which (3) infers a configured function net in WCRL. In order to visualize the result we (4) transform it back to the function net model. Note that steps (2)-(4) are internally processed so that a user does not notice them.

In the following subsections, we will explain the above outlined steps in more detail by illustrating them with the example of Section 2. We will start with step (2), i.e., a function net family plus selected variants which are transformed to a model in WCRL.

4.1 Transformation to a WCRL Model

Consider Figure 4. We have a family of function nets where WCRL constraints can be expressed (see Section 3) and in addition variants can be selected. In the example, we want to configure an `IDT` and four `Ats`. As explained above, we first have to transform the function net family to a model in WCRL.

The bottom part of the figure shows the result after transformation, i.e., a model in WCRL. For example, the functional module `VAC` is transformed to `Function(VAC) <-`. This is a short notation for a weight constraint rule where the right part is always true (and therefore empty). Such a rule is also called a *fact*. The information that `VAC` is also a variation point is captured in the next line. Furthermore, we need the association information between `VAC` as a

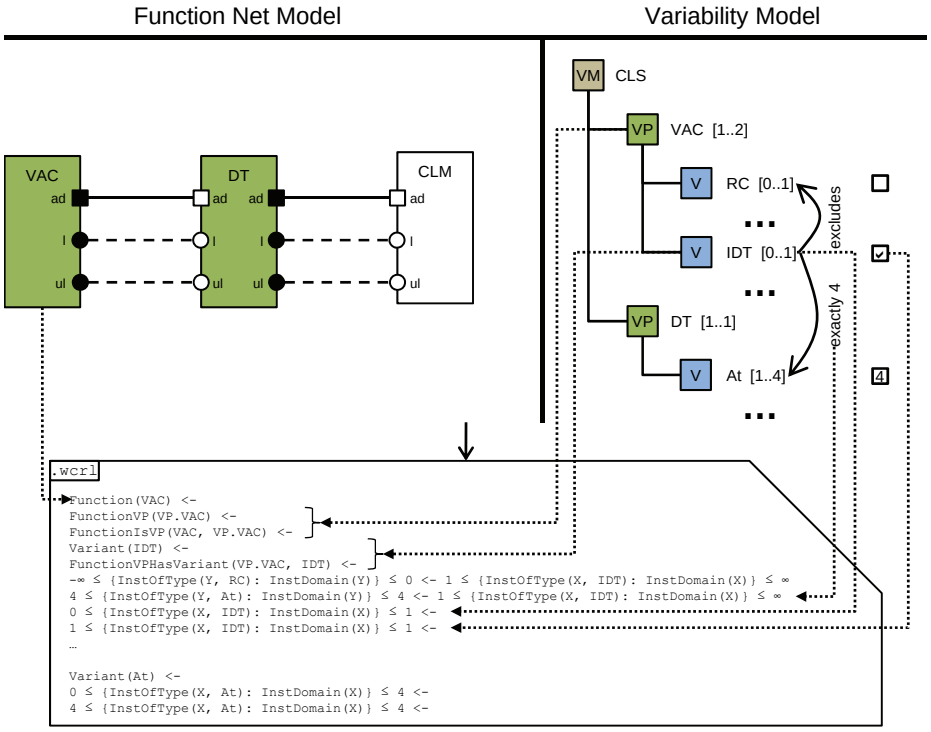


Fig. 4. Transformation of function net family model to WCRL.

functional module and variation point. This is expressed in the third line. In the same way, variants of variation points, constraints, cardinalities, and selections are all transformed to WCRL.

To implement transformation rules, we have used the oAW framework, more precisely the component Xpand. Xpand is a template-based language to generate code based on EMF (Eclipse Modeling Framework) models [14]. Figure 5 illustrates an Xpand template that processes all variation points and variants of our integrated models (which are also based on EMF) and generates the appropriate WCRL code. Having now the model in WCRL, we can use the inference engine smodels to generate a valid configuration.

4.2 The Inference Engine: smodels

We have decided to use an independent inference engine, which is able to process combined models and has a reasonable performance. We will not present in this paper our classification and evaluation method for our decision, because it is still work in progress. But among all evaluated possibilities, smodels seems to be a good candidate [15]. In Section 5 we will give some arguments for our decision.

```
1.  «DEFINE processVariationPoint FOR FunctionVariationPoint»
2.
3.    FunctionVP(«this.getId()») <-
4.    FunctionIsVP(«this.variableFunction.getId()», «this.getId()») <-
5.
6.    «FOREACH vcard.variants AS vars»
7.
8.      Variant(«vars.getId()») <-
9.      FunctionVPHasVariant(«this.getId()», «vars.getId()») <-
10.     «vcard.getLowerBound()» {InstOfType(X, «vars.getId()») :
                  InstDomain(X)} «vcard.getUpperBound()» <-
11.
12.   «ENDFOREACH»
13.
14. «ENDDEFINE»
```

Fig. 5. An example of an Xpand template that generates WCRL code for variation points and variants.

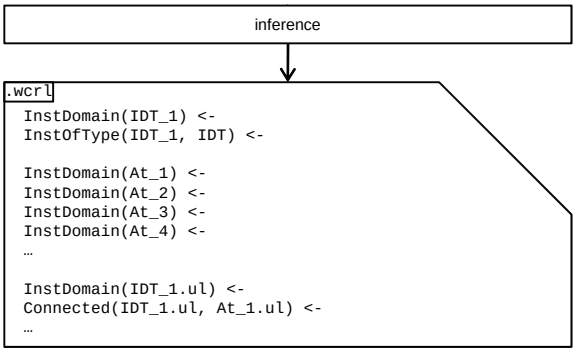


Fig. 6. The result of the inference engine smodels.

Using smodels as back-end inference engine, a WCRL model is taken as input which then infers a configured WCRL model as output (for the input from Figure 4). Figure 6 shows an example of the output. We have one instance `IDT_1` of type `IDT` (line 1 and 2), and four instances of type `At`. Furthermore, all instances for ports and their connections are inferred.

4.3 Transformation to a Function Net Model

Finally, the result of smodels has to be transformed back to the function net model. Therefore, we again specify our transformation rules using oAW. We do not describe this in detail, because it is analogous as described in Subsection 4.1. Figure 7 illustrates the result of this step. The function net has now one instance of `IDT` and four instances of `At`, which all are connected appropriately. Note that in this model the complete variability is bind (there is no green colored functional module) and therefore it is a variant that can be used as a skeleton for virtual prototyping.

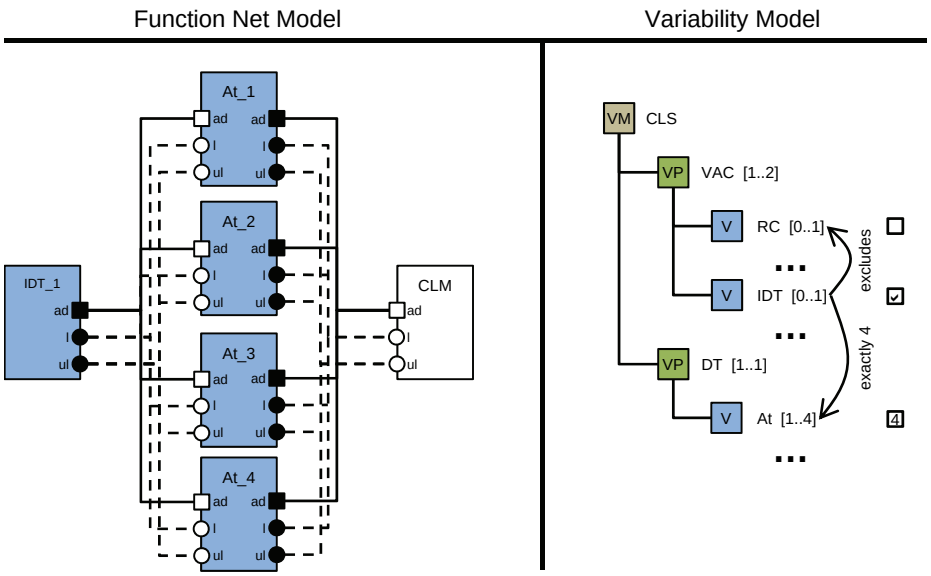


Fig. 7. The result of the overall configuration process as function net model.

5 Related Work

In terms of expressing constraints, two different approaches exist: structural/ graphical definition and textual definition. The former is usually adopted by configuration systems involving feature trees [16], and the possible relations include simple 'requires' and 'excludes' relations and cardinality constraints [10], and relations like 'recommends' or 'discourages' in some cases [17].The latter requires the use of a constraint language/logic, including OCL [18], WCRL [12], or Prolog [17] to express arbitrary logic formulas constraining the model, allowing less user-friendly yet much more expressive constraints. Among different languages, OCL has the advantage as the most well-known language since defining complex constraints requires a considerable amount of proficiency in the relevant language.

In [19] and [20], different methods for product configuration are summarized. Some are rule-based, logic-based and constraint-based configuration; the strengths and weaknesses of tools adopting different approaches are discussed in [21]. Answer Set Programming (ASP) [22] emerges to be an important and widely accepted means for product configuration, and is suitable for the representation of configuration knowledge [23]. Since the configuration task is NP-complete, performance of the inference engine seems to be important for medium to large scale models. There exist several solvers including pure ASP solvers such as smodels [15], DLV [24], and hybrid approaches integrated with SAT solvers such as ASSAT [25]. A theoretical comparison can be found in [26] and comparison of their performance in relevant sections of each paper and in [27].

Regarding the different ASP solvers, there exist considerable amount of data on using smodels, which has a long history of development and wide acceptance. It allows a direct integration into our configuration process with the use of WCRL as input language. In contrast, OCL requires an extra model transformation. However, we plan to integrate OCL into our configuration process in order to compare both in more detail. The disadvantage in performance and lack of advanced concepts (such as non-monotonic reasoning, learning, powerful heuristics) may be neglected considering the current scope of our work; and it is easy to migrate later to different engines with the same interface (WCRL I/O) for better performance.

6 Conclusion

In this paper, we have illustrated a model-driven approach to configure an integrated variability and function net model. First, we have explained the need for a constraint language in order to restrict the configuration domain. Therefore, we have included WCRL as an appropriate language. Second, we have motivated the necessity for an independent back-end inference engine. Here, smodels seems to be a tool which is adaptable, extendable, and with high performance. We plan to start a benchmark where this can be evaluated more precisely.

The possibility to configure concrete variants of function nets allow the generation of skeleton variants for virtual prototyping. For example, in this way, variants of Simulink models can be simulated in order to verify efficiently requirement specifications in an early phase of a model-driven development process.

References

1. M. Broy, I. H. Krüger, A. Pretschner, and C. Salzmann: Engineering Automotive Software. *In Proceedings of the IEEE*, pages 356–373, 2007.
2. K. Czarnecki and U.W. Eisenecker: Generative programming: methods, tools, and applications. Addison-Wesley, NY, USA, 2000.
3. K. Pohl, G. Böckle, and F. J. van der Linden: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, 2005.
4. H. Wallentowitz and K. Reif (eds.): *Handbuch Kraftfahrzeugelektronik: Grundlagen, Komponenten, Systeme, Anwendungen.* Vieweg, Wiesbaden, 2006.
5. AUTOSAR (AUTomotive Open System ARchitecture) Development Partnership. *http://www.autosar.org.* Last request: March 30. 2010.
6. C. Mengi and I. Armac: Functional Variant Modeling for Adaptable Functional Networks. *In VaMoS 2009: Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems*, pages 83–92, Seville, Spain, 2009.
7. C. Mengi and I. Armac: Ein Klassifikationsansatz zur Variabilitätsmodellierung in E/E-Entwicklungsprozessen. *In Software Engineering 2009 Workshop: Produkt-Variabilität im gesamten Lebenszyklus (PVLZ 2009)*, KL, Germany, 2009.
8. C. Mengi, A. Navarro Perez, and C. Fuß: Modellierung variantenreicher Funktionsnetze im Automotive Software Engineering. *In Proceedings of the 7th Workshop Automotive Software Engineering (ASE 2009)*, Lübeck, Germany, 2009.

9. C. Mengi, C. Fuß, R. Zimmermann, and I. Aktas: Model-driven Support for Source Code Variability in Automotive Software Engineering. *In Proceedings of the 1st International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2009)*, San Francisco, California, USA, 2009.
10. K. Czarnecki, S. Helsen, and U. W. Eisenecker: Formalizing cardinality-based feature models and their specialization. *In Software Process: Improvement and Practice*, pages 7–29, Vol. 10, 2005.
11. The Mathworks. *http://www.mathworks.com/*. Last request: March 30. 2010.
12. T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen: Representing Configuration Knowledge with Weight Constraint Rules. *AAAI Technical Report SS-01-01*, 2001.
13. openArchitectureware. *http://www.openArchitectureware.org/*. Last request: March 30. 2010.
14. Eclipse Modeling Framework. *http://www.eclipse.org/modeling/emf/*. Last request: March 30. 2010.
15. I. Niemelä and P. Simons: Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. *In LPNMR '97: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 421–430, London, UK, Springer-Verlag, 1997.
16. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson: Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Technical Report: Carnegie-Mellon University Software Engineering Institute*, 1990.
17. pure::systems. *http://www.pure-systems.com/*. Last request: 30th March 2010.
18. Object Constraint Language Specification Version 2.0. *http://www.omg.org/ technology/documents/formal/ocl.htm*. Last request: March 30. 2010.
19. D. Sabin and R. Weigel: Product Configuration Frameworks - A Survey. *IEEE Intelligent Systems, Vol. 13*, pages 42–49, IEEE Educational Activities Department, NJ, USA, 1998.
20. L. Hotz and T. Krebs: Configuration - State of the Art and New Challenges. *In Proceedings of the 17th Workshop Planen, Scheduling und Konfigurieren, Entwerfen (PuK 2003), KI 2003 Workshop*, pages 145–157, 2003.
21. M. Sinnema, and S. Deelstra: Classifying variability modeling techniques. *Inf. Softw. Technol.*, pages 717–739, 7, Newton, MA, USA, 2006.
22. C. Anger, K. Konczak, T. Linke, and T. Schaub: A Glimpse of Answer Set Programming. *Künstliche Intelligenz, Vol. 19*, pages 12–17, 1, 2005.
23. T. Soininen and I. Niemelä: Developing a Declarative Rule Language for Applications in Product Configuration. *In PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 305–319, London, UK, Springer-Verlag, 1998.
24. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello: The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic Vol. 7*, pages 499–562, 3, New York, NY, USA, ACM, 2006.
25. F. Lin and Y. Zhao: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artif. Intell., Vol. 157, Nonmonotonic Reasoning*, pages 115–137, 2002.
26. E. Giunchiglia, N. Leone, and M. Maratea: On the relation among answer set solvers. *Annals of Mathematics and Artificial Intelligence Vol. 53.* pages 169–204, 1-4, Hingham, MA, USA, Kluwer Academic Publishers, 2008.
27. T. Mancini, D. Micaletto, F. Patrizi, and M. Cadoli: Evaluating ASP and Commercial Solvers on the CSPLib. *Constraints, Vol. 13*, pages 407–436, 4, Hingham, MA, USA, Kluwer Academic Publishers, 2008.