Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 93 (2020) 556–561

www.elsevier.com/locate/procedia

53rd CIRP Conference on Manufacturing Systems

Using Component-based Software Synthesis and Constraint Solving to generate Sets of Manufacturing Simulation Models

Fadil Kallat[a,*], Carina Mieth[b], Jakob Rehof[a], Anne Meyer[c]

[a]TU Dortmund University, Chair for Software Engineering. Otto-Hahn-Str. 12, Dortmund 44227, Germany
[b]TRUMPF Werkzeugmaschinen GmbH & Co. KG, Johann-Maus-Str.2, 71254 Ditzingen, Germany
[c]TU Dortmund University, Chair of Enterprise Logistics, Leonhard-Euler-Str. 5, Dortmund 44227, Germany

Abstract

There is a high degree of flexibility in the design of production systems when it comes to the selection and configuration of machines. Simulation supports this complex decision process. However, modeling various configurations in a simulation environment is very time-consuming. We present a framework that includes component-based software synthesis to generate the set of all possible simulation models for the respective planning case. From the set, feasible solutions for a simulation study are then selected using constraint solving methods. We evaluate our approach using a practical example from sheet metal production.

Keywords: simulation model generation; manufacturing; factory planning; sheet metal production

1. Introduction

Smaller batch sizes due to individualized products increase the complexity of planning. Suitable production systems reliably produce a great variety of products. Furthermore, due to volatile demands, adaptation planning is required in ever shorter intervals [1]. As decisions in factory planning are the basis for economic success, it is particularly important to develop new technologies that allow for thorough planning with little effort [2]. Manufacturing simulation is an established method to support and facilitate the decision making process during the planning of complex production systems [3]. It allows to compare different system configurations with regard to their logistical goal achievement. Although manufacturing processes are already automated to the extent that it is economically feasible, planning processes are still largely carried out manually and individually by experts [4]. Thus, the creation of different simulation models is still very time-consuming. There is a need for adaptation of parameters, structures and control strategies in each simulation model, even if the variants are similar [5].

* Corresponding author. Tel.: +49-231-755-8496.
E-mail address: fadil.kallat@tu-dortmund.de (Fadil Kallat).

1.1. Methodological Structure

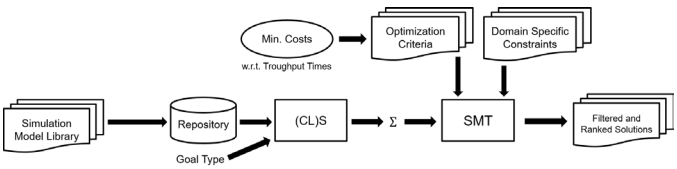

Fig. 1. Method to generate sets of feasible simulation models (SMT = satisfiability modulo theory; (CL)S = Combinatory Logic Synthesizer).

Our approach, as depicted in Fig. 1, uses component-based software synthesis and constraint solving methods. In component-based software synthesis programs are not built from scratch but are composed from a repository of software components [6, 7, 8]. The Combinatory Logic Synthesizer (CL)S is one such framework based on a type inhabitation algorithm for combinatory logic with intersection types [7]. The simulation model of the as-is analysis is migrated into a repository of typed components. Then the (CL)S framework is used for the automatic composition of all possible variants according to the user-intent (goal). The synthesis result of (CL)S is provided as a tree grammar Σ that is used to enumerate all possible variants [9]. We translate the tree grammar into adequate SMT formulas and introduce optimization constraints. In this

2212-8271
10.1016/j.procir.2020.03.018

work, the goal of the optimization criteria is to minimize the acquisition costs of the machines considering a defined threshold for the allowed throughput times. Domain-specific constraints are also expressible in the SMT-Solver such as incompatibilities between machines and automation components. SMT formulas are first-order logic formulas with respect to background theories. A background-theory prescribes the interpretation of certain predicates and function symbols [10]. We use the SMT solver Z3 [11] with the optimization functionality *νZ* [12] for solving those formulas. The SMT solver determines the satisfiability of the formulas and returns a tree model, which represents a solution.

1.2. Outline of this Paper

This paper is structured as follows: In Section 2, we introduce related work on automated simulation model generation in the context of factory planning. Additionally, the fundamentals of CL(S) are explained and some previous applications of this framework are presented. In Section 3, the industrial use case is depicted and we show the degree of freedom which is inherent to this planning problem. In Section 4, we set up the respective feature model and the repository for the synthesis of all possible simulation models. In Section 5, the runtimes of the CL(S) framework as well as the top three system configurations with respect to costs are given as the main results. We conclude this paper in Section 6 with a short summary and an outlook to future work.

2. Related Work

In this section, the current approach to factory planning is presented and compared with our approach and a brief overview of related work from the field of automated simulation model generation (ASMG) for factory planning is given. Then, the formal foundation of (CL)S is shortly outlined to provide a basic understanding of the theoretical background.

2.1. Further Development of existing approaches

The current approach to factory planning in practice is compared qualitatively with our approach in Fig. 1. During the as-is analysis, the state of the production system is examined to gain an understanding of the underlying problem. In the next step, experts collect and develop planning variants based on discussions and experience. This leads to selecting a few variants for detailed planning, since the experts involved still have to oversee the solution space. Subsequently, often only the favorite parameter setting is used for the simulation to assess the system performance [13].

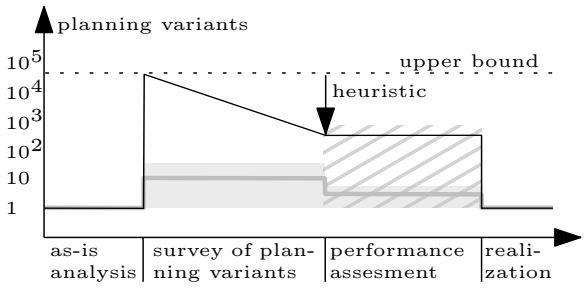

Fig. 2. Example of considered planning variants during the factory planning process (gray line = current procedure in practice; black line = our approach).

Our approach differs in that, after the as-is analysis, all combinatorially possible solution alternatives are generated automatically. From these, the most promising ones are selected by means of an optimization heuristic and simulatively compared using cloud computing, which is the basis to evaluate a huge number of different planning alternatives within a reasonable time [14]. Our approach avoids the rejection of potentially better plant concepts by experts in an early planning phase by considering a larger solution space. In this way, well-trodden paths are left, because scenarios are also considered that would at first glance appear to be unacceptable to experts.

2.2. ASMG for Factory Planning

There has been some research in the field of automated simulation model generation (ASMG) for discrete manufacturing over the last years that aims at reducing the efforts for simulation model generation for tasks in production planning and control, bottleneck detection, remanufacturing and layout planning [15].

Völker et al. [16] couple a factory planning tool with a simulation environment to automatically generate the matching simulation model. This approach reduces the effort in creating simulation models of planning alternatives, but it does not solve the problem of how to create different planning variants. This is still done manually by experts in the factory planning tool.

As far as we know, the existing approaches of ASMG have focused on generating a specific simulation model that satisfactorily represents the real system. In [5], an ASMG approach for the management and exploitation of the planning variety in factory planning projects was developed.

Previous work used the CL(S) framework for workflow management in factory planning. This paper focuses on the selection and configuration of machines and equipment. Winkels [1] used (CL)S within the scope of the precedence diagram method with a repository of taxonomically ordered building blocks for the creation of planning workflows. The planning blocks contain methods and processes of factory planning and are described in detail in [17].

In [5], a first theoretical example of a production line was synthesized with CL(S). There were only twelve system variants. This paper uses a real-world example with considerably larger solution space to prove the scalability and the applicability of our approach.

2.3. (Combinatory Logic) Synthesizer

Use cases of the Combinatory Logic Synthesizer (CL)S include the automatic composition of software and also for the synthesis of data structures such as BPMN 2.0 processes [8] and

planning processes [4]. The framework is fully integrated into the Scala programming language and is publicly available [18].

The formal foundation of (CL)S lies in the type-theoretical problem of inhabitation $\Gamma \vdash ? : \sigma$, which denotes the question if a well-typed applicative term exists, that can be formed from the user-specified repository Γ of typed combinators to satisfy the target type σ. Each combinator $(c : \tau)$ in Γ represents a type assumption τ for a combinator named c. The inhabitation algorithm [7] uses the combinator types to determine which combinators can be applied to each other in order to satisfy the target type.

If a combinatory expression M exists, so that $\Gamma \vdash M : \sigma$, then we call M inhabitant of σ. Terms are constructed by a named component or combinator c or an application of M to N, (MN), where M and N are terms. The type system of (CL)S is based on intersection types [7]. Types are constants which can be native or semantic types. Native types correspond to data types in programming languages, whereas semantic types can be used to consider additional domain-specific knowledge. Types can also be type variables that can be substituted by type constants according to a substitution map. Moreover, function types $(\sigma \to \tau)$ or intersections $(\sigma \cap \tau)$ can be used to construct a type.

Although the underlying type system of (CL)S is well-suited to express feature vectors of programs and software components, the expression of domain-specific knowledge is limited since expressing the logical connectives conjunction, disjunction and negation is difficult when using intersection types [19]. In addition, expressing constraints on the global structure of results is challenging, since in the combinatory approach the typing information is specified for a local combinator.

Therefore, in [19] the authors evaluated the joint usage of (complementary) specification formalisms that yield a synthesis approach, which benefits of the particular strengths of the underlying techniques. The authors identified SMT as a proper counterpart to the (CL)S specification formalism and used SMT for filtering the enumeration of inhabitants. The approach is implemented in a tool called CLS-SMT and was evaluated considering an example for sort programs and a labyrinth example. The additional domain-specific constraints referred to the structure of the inhabitants. In contrast, we mainly consider numerical optimization in this paper.

Heineman et al. [20] migrate an existing object-oriented framework into a software product line. They show how to design a repository of modular units, which are formalized using combinatory logic. In this paper, we follow a similiar approach by taking an existing simulation model and migrating it into a simulation model product line.

3. Industrial Use Case

In the context of this paper, a discrete event simulation application is implemented using the simulation framework *AnyLogic 8*. The model of a sheet metal box production is a real-world example. As shown in Fig. 3, it consists of an inbound warehouse for the sheets, an intermediate storage area for empty pallets, two cutting machines and two bending machines. One cutting machine is directly connected to the inbound warehouse, the other is supplied with sheets by forklifts. Cut blanks are also transported to the bending machines by forklifts.

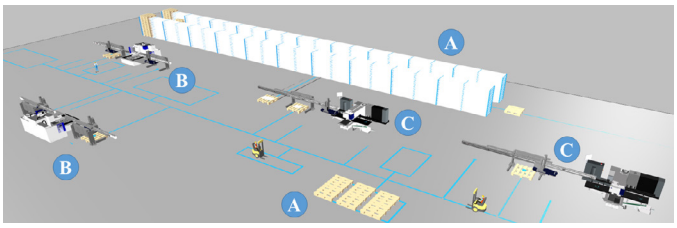

Fig. 3. Simulation model of sheet metal production system in AnyLogic with storages (A), two bending machines (B) and two cutting machines (C).

The variability in planning this production system results from the choice of two cutting and two bending machines. Different cutting machines are available, which differ in their cutting speeds (low/mid/high), the assembly space (processing of small, medium or large sheets possible) and the type of loading and unloading (automated - yes/no). We group continuous parameters into intervals, because otherwise the solution space would consist of an uncountable number of combinations. It should be emphasized here that this is fundamental to automating the factory planning process.

Considering the introduced intervals, this results in $2*3*3 = 18$ machine configurations for one cutting machine. For the bending machines, two types (mid- and high-end) are available, which bend at mid or high speed. Both in turn, can process different sheet sizes (small-medium-large) resulting in $2 * 3 = 6$ machine configurations for one bending machine. Fig. 4 summarizes all possible configurations for each machine. In total, there are $18 * 18 * 6 * 6 = 11664$ different system variants for the production system consisting of two cutting and two bending machines.

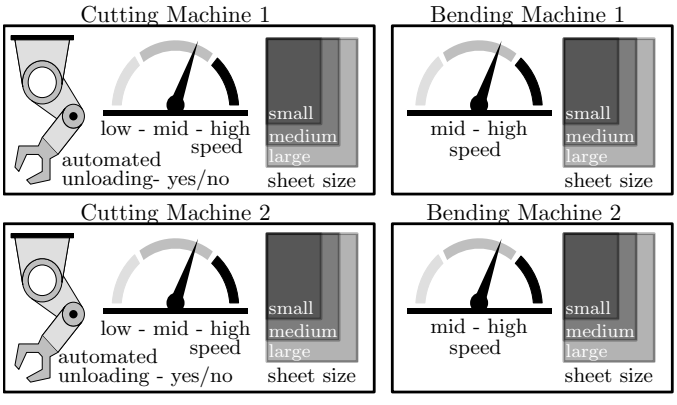

Fig. 4. Overview of the decision-making scope for the industrial use case.

4. Synthesis of simulation models

First, the scenario is converted into a feature model. A feature model represents all occurrences of a software product line

(SPL) [21]. It allows to map the variability and the different configurations of the simulation model in the given use case.

Converting the scenario into a feature model was feasible with little effort in consequence of the modular structure of the simulation model. Fig. 5 shows the feature model, where each leaf in the model represents a variation possibility. For instance, the model shows that a cutting machine varies in cutting speed, type of unloading and allowed sheet size.

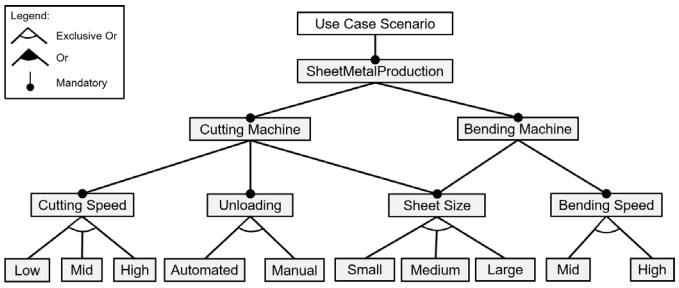

Fig. 5. Feature model.

Next, the feature model is transformed into combinators for synthesis, which is a fundamental step in our work. As a result of transforming, we achieve the repository Γ, which is shown in Fig. 6. Each combinator in the repository Γ is classified by a native type (e.g. integer or string) and a semantic type. Additionally, the combinators may have input parameters that are classified in the same way. Given a target type, (CL)S tries to find a set of combinators, that meet the target type and cover all input parameters with appropriate combinators.

The combinators *cuttingMachine* and *bendingMachine* represent configurations of the particular machines. For instance, the combinator *bendingMachine* requires a combinator of the type $E \cap BendingTime(\alpha)$ and also a combinator of the type $E \cap SheetSize(\gamma)$. The native type E is in place for the data type scala.xml.Elem, which represents XML documents in the programming language Scala. The variable α expresses that the bending time is arbitrary and can be substituted by *Low*, *Mid* or *End* according to the substitution map *WF* in Fig. 6. In the same way, the variables β and γ can also be substituted. Thus, the $BendingTime(\alpha)$ in the type specification of the bending machine can be filled in by the combinators *bendingMidEnd* or *bendingHighEnd* since they provide the required type. The total use case is described by the combinator *sheetProduction* in that manner. The combinator expresses the idea that the sheet production requires two cutting machines and two bending machines. By using different variables (α_1, α_2, ...), we achieve different configurations of the particular machines. However, we can allocate variables to constant values by restricting the substitution map *WF*. For instance, we can demand that the first cutting machine processes large-sized sheets with automated unloading.

Besides a combinator name and type assumption, combinators can have implementation details such as programs, data, data fragments or functions in the Scala implementation of (CL)S. In our case, they contain functions manipulating XML fragments, which represent parts of the simulation model in a AnyLogic project file. The combinators of the variation possibilities *cuttingLowEnd*, *cuttingMidEnd*,..., *largeSheetSizes* contain the XML encoded values according to the corresponding parameter. The implementations of the *cuttingMachine* and *bendingMachine* combinators produce XML code, which represent the machine in its chosen configuration. XML files are produced that are simulation models directly executable by *AnyLogic*. Listing 1 shows the call for executing the inhabitation algorithm. The algorithm is asked if it is possible to generate a solution that meets the requirement of using a combinatory expression (that is, a composition of combinators) of the type *SheetProduction*. In our use case, we obtain 11644 different valid and ready simulation models.

$$
\begin{aligned}
\Gamma = \{\ & sheetProduction : (E \to E \to E \to E \to E) \cap \\
& (CuttingMachine(\alpha_1, \beta_1, \gamma_1) \to \\
& CuttingMachine(\alpha_2, \beta_2, \gamma_2) \to \\
& BendingMachine(\alpha_3, \gamma_3) \to \\
& BendingMachine(\alpha_4, \gamma_4) \to \\
& SheetProduction), \\
& cuttingMachine : (E \to E \to E \to E) \cap \\
& (CuttingTime(\alpha) \to \\
& Unloading(\beta) \to \\
& SheetSize(\gamma) \to \\
& CuttingMachine(\alpha, \beta, \gamma)), \\
& bendingMachine : (E \to E \to E) \cap \\
& (BendingTime(\alpha) \to \\
& SheetSize(\gamma) \to \\
& BendingMachine(\alpha, \gamma)), \\
& cuttingLowEnd : E \cap CuttingTime(Low), \\
& cuttingMidEnd : E \cap CuttingTime(Mid), \\
& cuttingHighEnd : E \cap CuttingTime(High), \\
& bendingMidEnd : E \cap BendingTime(Mid), \\
& bendingHighEnd : E \cap BendingTime(High), \\
& manualUnloading : E \cap Unloading(Manual), \\
& automaticUnloading : E \cap Unloading(Automated) \\
& smallSheetSize : E \cap SheetSize(Small) \\
& mediumSheetSize : E \cap SheetSize(Medium) \\
& largeSheetSize : E \cap SheetSize(Large)\ \}
\end{aligned}
$$

$$
\begin{aligned}
WF = \{\ & (\alpha \to Low),\ (\alpha \to Mid),\ (\alpha \to High), \\
& (\beta \to Automated),\ (\beta \to Manual), \\
& (\gamma \to Small),\ (\gamma \to Medium),\ (\gamma \to Large)\ \}
\end{aligned}
$$

Fig. 6. The upper part shows the combinator repository Γ for simulation model synthesis, while the substitution map *WF* is shown in the lower part, which indicates how the variables α, β and γ can be replaced.

```
lazy val results =
  Gamma.inhabit[Elem]('SheetProduction)
```

Listing 1. Call for executing inhabitation

However, not all of the generated simulation models are useful since various configurations violate general project constraints. Therefore, we introduce SMT techniques for filtering solutions. We use the tool CLS-SMT to translate the synthesis result of the (CL)S framework, which is provided as a tree grammar, into adequate SMT formulas. Solving these formulas with an SMT solver yields a tree model, which is a valid inhabitant. The shape of solutions can be influenced by additional domain-specific constraints. In this paper, we consider constraints that limit the throughput time and minimize the total costs.

All possible cutting and bending machine configurations are compromised in the following sets:

$$CM = \{\overrightarrow{cm}^1, \overrightarrow{cm}^2, \ldots, \overrightarrow{cm}^{18}\} \quad \text{and} \quad BM = \{\overrightarrow{bm}^1, \overrightarrow{bm}^2, \ldots, \overrightarrow{cm}^6\}$$

We represent a cutting and bending machine configuration by the following vectors:

$$\overrightarrow{cm} = (cm_{cspeed}, cm_{size}, cm_{unloading}) \text{ and } \overrightarrow{bm} = (bm_{bspeed}, bm_{size})$$

We define the functions *t()* and *c()* that take a machine configuration and return the throughput time or costs, respectively. We assume that x sheets are processed by a cutting and bending machine that are configured to assemble the same sheet size. The same applies to the variable y. Therefore, we require $cm^i_{size} = bm^j_{size}$ and $cm^k_{size} = bm^l_{size}$ with $0 < i,k \leq 18$ and $0 < j,l \leq 6$.

Given the number of sheets x, y and the additional processing times e_1, e_2, which depend on the sheet size, we can calculate the total costs and throughput time for a concrete configuration as follows:

$$totalCosts = x * (c(\overrightarrow{cm}^i) + c(\overrightarrow{bm}^j)) + y * (c(\overrightarrow{cm}^k) + c(\overrightarrow{bm}^l)),$$

$$totalTime = x * (t(\overrightarrow{cm}^i) + t(\overrightarrow{bm}^j) + e_1) + y * (t(\overrightarrow{cm}^k) + t(\overrightarrow{bm}^l) + e_2)$$

Then, we can require that the total time should not exceed 400 minutes with minimal costs.

$$totalTime \leq 400 \wedge min(totalCosts)$$

We implemented the described constraints as SMT formulas and use the SMT Solver Z3 [11] with the optimization functionality νZ [12]. By solving the formulas, we receive a tree model that relates to the cheapest configuration. The solver returns the next best tree model, when the previous one is added as a negated formula. The solutions are generated and saved as AnyLogic Project Files, so that they can directly be executed and evaluated in the simulation environment *AnyLogic 8*.

5. Experimental Results

In this experiment, we introduce an exemplary job that requires the processing of 40 small-sized and 60 medium-sized sheets. Therefore, the inhabitation algorithm is asked if it is possible to generate a solution that requires a cutting and bending machine configured to handle small-sized and medium-sized sheets. We assume that a machine, which is able to process large-sized sheets can also handle sheets of medium and small size. The experiments were performed with the parameters shown in Table 1. The table illustrates the maximum cutting and bending time for a single sheet depending on the chosen machine configuration. Furthermore, it lists additional processing times depending on the type of loading and unloading and the sheet size. The shown process times are exemplary, but close to reality. The costs were chosen in relation to the machine configuration and can easily be replaced by concrete prices.

The cheapest, but also the slowest configuration comprises cutting at low speed, bending at mid speed and manual unloading. By having this configuration, processing a small-sized sheet takes 72+120+180 = 372 seconds. Processing a medium-sized sheet extends the time by 10 seconds and a large-sized sheet by 20 seconds. In this way, the cheapest configuration takes 647 minutes and costs 2120 whereas the fastest takes 317 minutes and costs 4020. By using the (CL)S inhabitation without filtering, we obtain 5184 different configurations. For this experiment we assume a time limit of 400 minutes. After the filtering with SMT techniques 1332 solutions are remaining. Table 2 shows the three *best* solutions according to the lowest costs and shortest throughput time. We migrated the simulation model within the space of a few weeks in a small team. The

Table 1. Costs and time (in seconds) that are considered during optimization

	Cutting Speed			Bending Speed		Unloading		Sheet Size		
	Low	Mid	High	Mid	High	Auto	Manual	Small	Medium	Large
Time	72	60	48	120	96	30	180	0	10	20
Costs	5	10	15	10	15	5	1	2	4	6

Table 2. Three *best* machine configurations

		Cutting Machine 1			Cutting Machine 2			Bending Machine 1		Bending Machine 2	
Costs	Time	Speed	Sheet Size	Unloading	Speed	Sheet Size	Unloading	Speed	Sheet Size	Speed	Sheet Size
2240	380	L	S	A	L	M	A	M	S	M	M
2360	380	L	S	A	L	M	A	M	S	M	L
2400	380	L	M	A	L	M	A	M	S	M	M

synthesis and the generation of the 1332 variants took 29 minutes and 53 seconds. (CL)S provided the first executable solution after 8 seconds. The experiment was run on a workstation computer equipped with an Intel i7 processor and 64 GB RAM.

6. Conclusion and Outlook

In this paper we have used a real-world example of sheet metal production to show how component-based software synthesis and constraint solving with the CL(S) framework can be used to generate all feasible system concepts during factory planning and to propose the most economical option among them. Different degrees of freedom are coded in a tree grammar, whereby continuous variables are decomposed into suitable interval ranges.

Our experiments suggest that the runtime of the synthesis algorithm is challenging but not prohibitive in practically relevant scenarios. The experiment documented here is only the first in a series to be carried out in future work, with a view towards more comparative information on scalability and further research on engineering and tuning the synthesis algorithm for scalability.

We are currently in the process of extending the presented framework by an automatic adaptation of the layout if the number of machines is changed. Also, the number of required machines should be determined by the framework itself and no longer by humans. Therefore, a further detailing of the production spectrum is necessary to generate solutions that also include a feasible machine allocation. Furthermore, the approximation for throughput time estimation during constraint solving could be further improved. It is also planned to apply learning methods to the generated system configurations in order to create a recommendation system that can help factory planners.

Acknowledgements

The authors gratefully acknowledge the support by the German Research Foundation (Deutsche Forschungsgemeinschaft (DFG)) within the Research Training Group GRK 2193 (www.grk2193.tu-dortmund.de) located in Dortmund. This work is a joint project with TRUMPF Werkzeugmaschinen GmbH & Co. KG.

References

[1] L. Lenz, J. Graefenstein, J. Winkels, and M. Gralla, "Smart factory adaptation planning by means of bim in combination of constraint solving techniques," in *Word Building Congress*, vol. 17, p. 2019, 2019.

[2] D. Zuehlke, "Smartfactory—towards a factory-of-things," *Annual Reviews in Control*, vol. 34, no. 1, pp. 129–138, 2010.

[3] A. Negahban and J. S. Smith, "Simulation for manufacturing system design and operation: Literature review and analysis," *Journal of Manufacturing Systems*, vol. 33, no. 2, pp. 241–261, 2014.

[4] J. Winkels, J. Graefenstein, T. Schäfer, D. Scholz, J. Rehof, and M. Henke, "Automatic Composition of Rough Solution Possibilities in the Target Planning of Factory Planning Projects by Means of Combinatory Logic," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice* (T. Margaria and B. Steffen, eds.), Lecture Notes in Computer Science, (Cham), pp. 487–503, Springer International Publishing, 2018.

[5] Sigrid Wenzel, Jakob Rehof, Jana Stolipin, and Jan Winkels, "Trends In Automatic Composition Of Structures For Simulation Models In Production And Logistics," in *Proceedings of the 2019 Winter Simulation Conference*, (Maryland, Washington, USA), 2019.

[6] J. Rehof and M. Y. Vardi, "Design and Synthesis from Components (Dagstuhl Seminar 14232)," *Dagstuhl Reports*, vol. 4, no. 6, pp. 29–47, 2014.

[7] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof, "Combinatory Logic Synthesizer," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change* (T. Margaria and B. Steffen, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 26–40, Springer, 2014.

[8] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof, "Combinatory Process Synthesis," in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques* (T. Margaria and B. Steffen, eds.), Lecture Notes in Computer Science, (Cham), pp. 266–281, Springer International Publishing, 2016.

[9] J. Bessai and A. Vasileva, "User support for the combinator logic synthesizer framework," *Electronic Proceedings in Theoretical Computer Science*, vol. 284, p. 16–25, Nov 2018.

[10] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, p. 69, Sept. 2011.

[11] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 337–340, Springer, 2008.

[12] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "*v*Z - An Optimizing SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. Baier and C. Tinelli, eds.), vol. 9035, pp. 194–199, Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[13] F. Auris, F. Gaisbauer, and T. Bär, "Exploring process variance in assembly planning with non-fixed simulation parameters," *Procedia CIRP*, vol. 81, pp. 1213–1218, 2019.

[14] S. J. Taylor, T. Kiss, G. Terstyanszky, P. Kacsuk, and N. Fantini, "Cloud computing for simulation in manufacturing and engineering: introducing the cloudsme simulation platform," in *Proceedings of the 2014 Annual Simulation Symposium*, p. 12, Society for Computer Simulation International, 2014.

[15] H. Reinhardt, M. Weber, and M. Putz, "A survey on automatic model generation for material flow simulation in discrete manufacturing," *Procedia CIRP*, vol. 81, pp. 121–126, 2019.

[16] S. Völker, M. Bacher, P.-M. Schmidt, and G. Gross, "Automatische Generierung logistischer Simulationsmodelle auf Basis von Planungswerkzeugen der Digitalen Fabrik," *Frontiers in Simulation*, vol. 15, pp. 518–523, 2005.

[17] J. Graefenstein, D. Scholz, O. Seifert, J. Winkels, M. Henke, and J. Rehof, "Automated processing of planning modules in factory planning by means of constraint solving using the example of production segmentation," in *Customization 4.0*, pp. 157–172, Springer, 2018.

[18] J. Bessai, B. Düdder, G. T. Heineman, *et al.*, "(CL)S Framework," 2019. Available at https://github.com/combinators/cls-scala. Accessed: 2019-11-01.

[19] F. Kallat, T. Schäfer, and A. Vasileva, "CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories," in *Electronic Proceedings in Theoretical Computer Science*, vol. 301, pp. 51–65, Aug. 2019. arXiv: 1908.09481.

[20] G. Heineman, A. Hoxha, B. Düdder, and J. Rehof, "Towards migrating object-oriented frameworks to enable synthesis of product line members," in *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, (Nashville, Tennessee), pp. 56–60, Association for Computing Machinery, July 2015.

[21] J. Bessai, B. Düdder, G. T. Heineman, and J. Rehof, "Combinatory synthesis of classes using feature grammars," in *Revised Selected Papers of the 12th International Conference on Formal Aspects of Component Software - Volume 9539*, FACS 2015, (Berlin, Heidelberg), p. 123–140, Springer-Verlag, 2015.