Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

COMPONENT-BASED SYNTHESIS OF STRUCTURAL VARIANTS OF SIMULATION MODELS FOR CHANGEABLE MATERIAL FLOW SYSTEMS

Jan Winkels[1], Felix Özkul[2], Robin Sutherland[2], Jannik Löhn[1], Sigrid Wenzel[2], and Jakob Rehof[1,3]

[1]Department of Computer Science, TU Dortmund University, GERMANY
[2]Dept. Organization of Production and Factory Planning (pfp), University of Kassel, GERMANY
[3]Lamarr Institute for Machine Learning and Artificial Intelligence, Dortmund, GERMANY

ABSTRACT

Despite relevant research endeavors, modeling efforts related to the building of discrete-event simulation models for planning changeable material flow systems still limit their practical application. This is because simulation experts have to model many possible structural variants and compare them based on key performance indicators such as throughput, workload or investment costs, while also ensuring sufficient system changeability. This article presents a methodology for reducing efforts for structural variation during the experimental phase of a simulation study. Starting from a valid initial simulation model, structural variants of this simulation model are automatically generated by applying component-based software synthesis which uses combinatorial logic; thereby, a range of simulation models is provided for the user. This paper presents the outlined methodology using a case study and places it in the research context of reducing efforts associated with the design and execution of simulation experiments.

1 INTRODUCTION

Nowadays, manufacturing companies operate in a dynamic market environment that is increasingly characterized by disruptions, which requires continuous change within these organizations to survive in global competition; thus, changeability is becoming a decisive competitive factor for production and logistics (Wiendahl and Heger 2004). Production and logistics systems should no longer only be planned in a solution-neutral way (Cisek et al. 2002) but must also allow *in vivo* possibilities for low-cost structural changes over a system operating period; this deliberate change to the system structure is referred to below as structural variance (Sutherland et al. 2024). Structural variance can be concretized as a set of operations through which changes to system structures are made. These include adding or removing elements, changing the connection structure (element sequence), merging several elements and substituting these elements (Sutherland et al. 2024). In this paper, we only consider the structure and material flow view in relation to production and logistics systems, which is why the term material flow system (MFS) is used below as an umbrella term for the systems to be planned.

Discrete-event simulation (DES) has proven its suitability for the planning of MFSs in a wide range of industries and for a variety of planning reasons (e.g., see Bicalho-Hoch et al. 2022). It can be used to simulate and experimentally investigate the stochastic dynamic behavior of MFSs. However, conducting simulation studies (see Law 2019; Rabe et al. 2009) is still primarily time-consuming knowledge work. The scientific literature mentions significant efforts especially in connection with the building of simulation models, the collection of input data and information, verification and validation as well as the design and execution of experiments (see section 2). In order for MFSs to be planned in a changeable manner, simulation users and system experts have to model and evaluate a large number of structural variants. In practice, however, those involved in planning projects often only have time to simulatively examine a few candidate variants of systems; systematic and comprehensive modeling of all variants is usually uneconomical due to the effort involved.

Therefore, the methodology discussed in this paper aims to reduce the effort associated with building model structural variants during the experimental phase. Decisive for our approach is the use of the

Combinatory Logic Synthesizer (CLS) framework (Bessai et al. 2014), which creates a product line based on a repository (i.e., a set of typed combinators). The product line is the set of all possible solutions synthesized by CLS under the specification of composition rules (i.e., the target specification; see section 3). Starting from a valid initial simulation model, CLS is used to synthesize structural variants of the model. For this purpose, model elements are first wrapped as components and represented in CLS by typed combinators, from which structural variants (i.e., component-based representations of MFS simulation models) can be synthesized. The synthesized structural variants meet certain requirements that are previously specified and taken into account by CLS during synthesis. For example, the synthesis with CLS is carried out by applying the above-mentioned set of operations (e.g., by adding or removing components). Synthesized structural variants are then returned to the simulation tool and are available to the simulation user (see section 4) for further experimentation. As the synthesis takes place automatically and in the background from the user's point of view, the time required to build structural model variants is significantly reduced. The remainder of this paper focuses on detailing the outlined approach using a case study.

The paper is structured as follows: First, we present the state of the art in reducing the efforts associated with the design and execution of simulation experiments and place our contribution within it. Then, we delve into the theoretical foundations of combinatorial logic synthesis with intersection types and the CLS framework to establish a sufficient understanding for the methodology outlined in section 4. Section 5 describes the use case to which the methodology is applied – a laboratory scale MFS (further details in Özkul et al. 2023). Subsequently, we present a proof-of-concept implementation, demonstrating how the CLS approach can effectively be utilized for the automatic generation of model structural variants during the experimental phase. The paper concludes in section 6 with an outlook on future research that will build on this work, aiming to further develop the prototype and explore its practical application.

2 RELATED WORK

There is a significant amount of scientific literature that deals with reducing the effort of conducting a simulation study, with a particularly pronounced field of application in production and logistics (Reinhardt et al. 2019). Those approaches often focus on the automatic generation of simulation models, which is confirmed by numerous literature reviews (see Reinhardt et al. 2019; Wenzel et al. 2019; Schlecht et al. 2023). However, not only the implementation of a simulation model is a time-consuming task, but so are the preparation and execution of experiments to empirically investigate the model behavior (Ruscheinski et al. 2018). In particular, the high number of different simulation experiments – which are necessary to ensure statistically valid results – is a significant burden for users. For this reason, some simulation tools include experiment management systems with features for automated experimental design, parameter variation and results analysis. In recent years, some researchers have also focused on the specification, automatic generation, execution, and reuse of simulation experiments (Wilsdorf et al. 2023). The various approaches can be distinguished according to the objective and extent of automation, the methodology used, the types of experiments, and the simulation tools supported (Wilsdorf et al. 2023).

Simulation experiments must be specified in advance for automated execution (Wilsdorf et al. 2023). Therefore, some research work focusses on this (see Ruscheinski et al. 2018; Teran-Somohano 2015; Wilsdorf et al. 2019). For example, Ruscheinski et al. (2018) use templates containing information about the various experiments and methods to generate the simulation experiments. Wilsdorf et al. (2019) also use this information and guide the user through the specification process independently of the simulation method (e.g., discrete or continuous simulation methods). In general, explicit specification is relevant for the accessibility and reusability of simulation results (Wilsdorf et al. 2023). Accordingly, there are also approaches that address the reuse of the specification of simulation experiments (see Peng et al. 2017; Feng and Jiang 2020). For example, Peng et al. (2017) pursue the idea that the specification from individual simulation models can be reused for extended or composite simulation models. To achieve this, the experiment specification of individual models (i.e., the specified behavioral properties) is optionally enhanced with information on experiment adaptation (e.g., renaming or reassigning model parameters) and

reused or automatically adapted for the composed model. Finally, the specified behavioral properties are checked for validity by conducting experiments with the composed model.

In general, the applicability of some approaches is limited to one application domain (e.g., biology, engineering or healthcare), one specific experiment type (e.g., sensitivity analysis or steady-state analysis) or one simulation tool (Wilsdorf et al. 2022). Therefore, Wilsdorf et al. (2021) describe best practices for automatically selecting, parametrizing, generating, and executing simulation experiments. For that, they link the contextual information of a simulation model with an ontology defining the general properties of different analysis methods (e.g., sensitivity analysis). This approach is being further developed by the authors so that experiments can also be identified automatically (Wilsdorf et al. 2023). For this purpose, information on previous modeling and experimental activities is collected, compared with defined patterns, and then interpreted (Wilsdorf et al. 2023).

When planning changeable systems, a systematical variation of the structure of a simulation model can also be useful. However, determining the appropriate system structure variants is a challenge (Wenzel et al. 2019). Compared to automatic model generation, in this case a valid executable model already exists. The generation of variants through structural adaptations considering structural variance (Sutherland et al. 2024) is only considered in a few research contributions (see Lattner et al. 2010; Wenzel et al. 2019; Kallat 2023). The automatic generation of system variants not only saves time and money, but also enables the systematic identification of structural variants (Lattner et al. 2010). This approach is followed by Lattner et al. (2010), who automatically apply knowledge-based structural changes to a simulation model. The knowledge base contains information about existing and potential components and how they are connected. This information is then used to generate variants of an existing simulation model. These approaches are often tailored to a specific use case and can only be transferred with significant efforts (Reinhardt et al. 2019). In addition, Wenzel et al. (2019) discuss the necessity of automatic structure variant generation for simulation models and test an approach in a simplified use case using combinatorial logic. Based on the work of Wenzel et al. (2019), further synthesis experiments have also been carried out using the CLS framework (see Kallat et al. 2020; Kallat et al. 2021; Mages et al. 2022). Kallat et al. (2020) successfully use the CLS framework with constraint-solving techniques to synthesize sophisticated simulation models for factory configurations. These models represent specific factory setups differing in their machine settings. By employing constraint-solving techniques, the authors were able to filter the synthesized configurations, taking into account numerical constraints such as total costs or processing times. Kallat et al. (2021) utilize the CLS framework to generate simulation models for block-stacking warehouses, with the resulting models differing structurally in their process logic. To make this approach accessible even without programming knowledge, the authors developed a method to automatically extract components from existing simulation models. Users can modify semantic types and synthesis goals in this method through a user-friendly web application to generate new variants. In contrast to previous works where variants of process logic were synthesized within a model, our proof-of-concept generates new models and compositions based on the same building blocks (see section 5). Both approaches are inspired by the work described in Heineman et al. (2015), where an existing software library was migrated into a product line. However, our method presents a novel approach to the gradual migration of an existing application into a product line.

An overlap with our work lies with Mages et al. (2022), who also employ CLS for automatic composition of simulation models in AnyLogic. Analogous to our approach in sections 3 and 4 of this paper, semantically typed components are used in their work as well, which are then automatically assembled into various variants matching a target specification by the synthesis framework. However, unlike in our approach, a fixed layout of machines and the intermediate transport paths are used. Within this layout, components are substituted in the variants. In contrast, we go a step further and demonstrate an approach to also synthesize the arrangement of machines, thus accommodating different layouts.

3 SYNTHESIS WITH COMBINATORY LOGIC

CLS (Bessai et al. 2014) is a framework for composing software components or data structures based on type signatures. It utilizes combinatory logic (Hindley 2008) to generate solutions for planning by

establishing its formal foundation and employing intersection types. Type expressions include constants (native or semantic), variables, function types, and intersections. CLS addresses software synthesis by solving the type-theoretic problem of inhabitation (Barendregt et al. 2013), connecting type theory and programs (the inhabitation problem is: Given a type, does there exist a program having the type?). The synthesis process combines existing software components in a combinatory way from a repository, ensuring completeness. Implementation details can be provided for combinators, including variability points and type taxonomies. Algorithms for deciding inhabitation problems are available, supporting component-based synthesis (Rehof et al. 2014). Type expressions, denoted as σ and τ, are defined as follows:

$$\sigma, \tau :: = a \mid \omega \mid \alpha \mid \sigma \rightarrow \tau \mid \sigma \cap \tau \tag{1}$$

That means that a type expression can either be a type constant (a / ω), a type variable (α), a type function (σ → τ, where a certain input type is converted to a certain output type), or an intersection type (σ ∩ τ, where two or more types are intersected and mapped onto a type variable). Type constants, represented by a, b, c, ..., can be programming language types (native types like string or integer for example) or textual descriptions (semantic types). The special type constant ω serves as the top element of the subtyping relation. Type variables, indicated by α, β, γ, ..., are substituted with type constants according to a substitution map, which is part of the domain specification (not part of type expressions). This map is employed to resolve type variables before computing an inhabitant. Additionally, type expressions may contain function types (→) and intersections (∩) (Barendregt et al. 2013). In addition to the type signature, implementation details can be provided for combinators, including programs, data, data fragments, or functions. Variability points can be inserted and described with the type expression. The use of type taxonomies and type variables further supports the comprehensive specification of complex combinators. An algorithm for deciding the inhabitation problem for intersection types is detailed in Düdder et al. (2012). The component-based synthesis with intersection types can be categorized based on the dimensions in program synthesis (Gulwani 2010). Domain knowledge can be expressed through the semantic layer and corresponding combinator implementation. The search space is defined by well-formed applicative compositions of available combinators, and the inhabitation algorithm represents the search strategy. User intent must be supplied as a target type expression.

4 PROCEDURE MODEL

Our primary objective is the development of a systematic procedure model enabling users of simulation tools to automatically generate diverse structural variants of their models. To achieve this, we have conceived a comprehensive procedure model, exemplified in Figure 1 using AnyLogic. This simulation tool is chosen for the initial prototype development and test due to its dedicated material handling library for modeling production and logistics systems and its support for multiple modeling concepts (e.g., agent-based modeling). In the procedure model, the user specifies a model part or the entire model that they want new structural variants of. These parts will then be regenerated. Subsequently, the coordinates of the selected region and the relevant (partial) model boundaries are conveyed to our synthesis framework.

This synthesis framework (CLS) draws upon a repository of modular building blocks, such as machines or buffers. Its task is to explore and identify solutions that effectively cover the material flow from the system's input to its output. As elucidated in Bessai (2019), CLS generates *all* possible solutions that can be constructed from existing building blocks and fulfill the specified input and output conditions. The set of diverse solutions is then transmitted to a connector, responsible for generating an XML file for each solution. These XML files encapsulate the individual components, complete with appropriate coordinates, allowing them to be seamlessly processed by AnyLogic.

Consequently, the user is presented with an array of new simulation models, each representing a variant of their original model. These variants can be executed, facilitating a comprehensive assessment and evaluation of their performance.

In this paper, our emphasis is specifically directed towards elucidating our approach to generate structural variants. We elaborate on the process of assembling solutions and provide insights into the implementation details. To facilitate a better understanding, we introduce a benchmark system and subsequently guide the reader through a step-by-step explanation of how structural variants are systematically generated from this system.

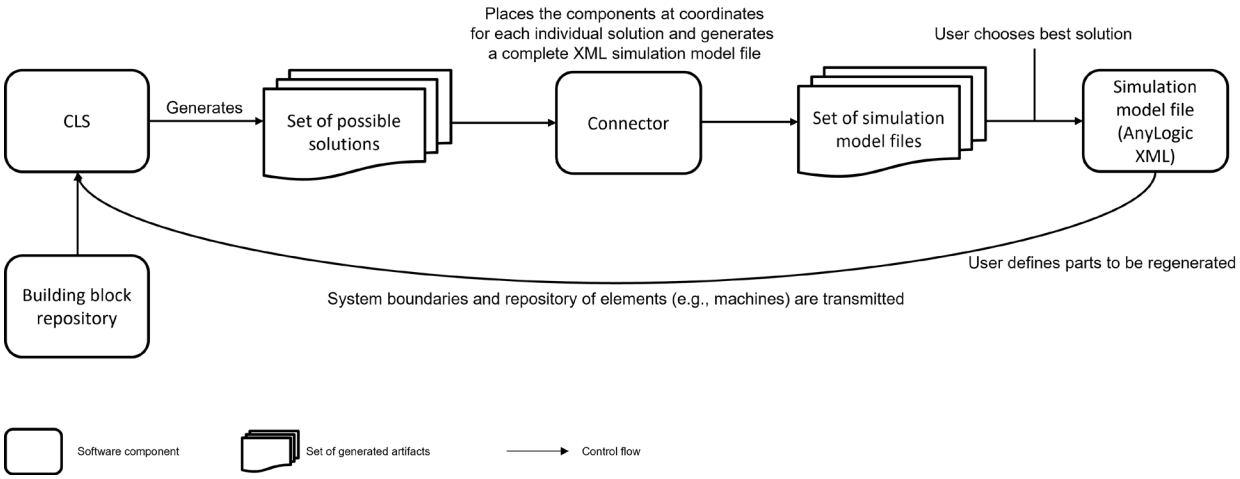

Figure 1: Procedure model for the generation of variants of a simulation model.

5 USE CASE

Our benchmark system is a section of a conveyor network with machines on a laboratory scale, which is shown schematically in Figure 2 below. Starting from a source (K_{in}), object carriers loaded with objects are loaded onto a conveyor belt into the system. The four object types {a, b, c, d} are conveyed via the main conveyor to side conveyors in order to be processed at the machines M1, M2, and M3 in different sequences and to different extents. The outfeed from or infeed to the main conveyor take place at the conveyor switches w2, w4 and w6. If the capacity utilization on a side conveyor is too high to receive further object carriers, no further object carriers will be discharged via the conveyor switches. Instead, the object carriers are buffered on the main conveyor and conveyed cyclically. In this paper, however, this is ignored for reasons of simplification and we assume that every material can always travel the intended route through the system. After processing on one of the machines M1, M2 or M3, the material is conveyed back towards

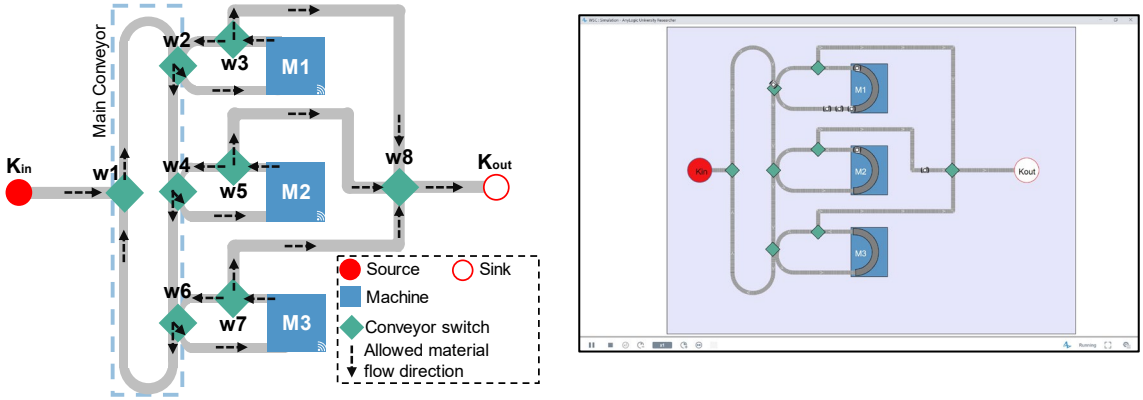

Figure 2: Benchmark system (l.) and corresponding simulation model (r.).

the main conveyor. If an object has passed through all the required processing steps, the object carrier will be ejected immediately after the last processing step via one of the conveyor switches w3, w5 or w7 in the direction of w8 and conveyed to the sink (K_{out}).

To successfully synthesize simulation models of production systems using the approach described above, CLS requires a target specification that reflects the structures of these models. For this purpose, we have chosen a graph representation. In order to provide the synthesis framework with simple structures and components, different graphs have to be mapped onto simple and generic structures. Based on our analysis, we distinguish three node types:

- Sources and sinks: Represent the system boundaries and introduce objects into the system or remove the desired output objects.
- Machines: Process and transform objects.
- Structural nodes: Represent branching or synchronization points of transport links of objects within the system.

Edges in the graph represent (transport) connections between two elements (nodes) of the MFS. These edges can be labeled to restrict which objects can be transported on them. Based on these considerations, we create a graph model. The model contains the elements mentioned above and is subject to some general restrictions:

1. Each graph must have exactly one input (K_{in}) and output node (K_{out}) specifying the system boundaries.
2. Edges must always be labeled to specify which objects may be transported on them.
3. Any object appearing as part of an edge label in the graph must either be introduced into the system by K_{in} or transformed by a machine.
4. Any object appearing as an edge label in the graph must either be removed from the system by K_{out} or transformed by a machine.
5. The graph must be connected.

Figure 3 illustrated our benchmark system mapped in this kind of graph model. In this model, the round nodes represent sources and sinks. The diamonds are the structural nodes, while the squares are machines.

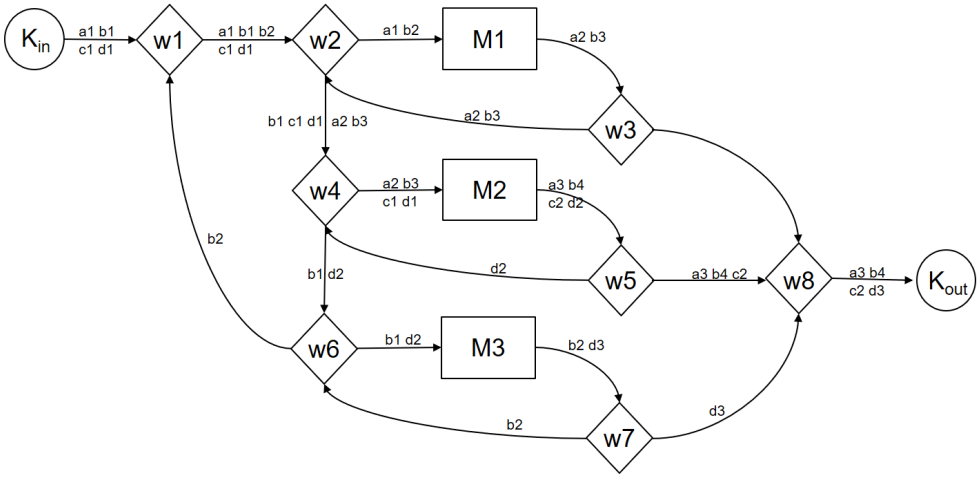

Figure 3: Benchmark system as a modeled graph.

The edges represent the material flow between the elements, whereby the edge labels restrict which objects can be transported on the respective edge.

Our synthesis software produces tree structures representing combinatory solutions. However, the graph depicted does not conform to a tree structure as it contains parallel branches that subsequently merge, as well as cycles. Recognizing the need to handle such complexities, we have introduced additional specialized nodes. These nodes serve the purpose of abstracting cycles and parallelization within the graph, facilitating the transformation of the original non-tree structure into a form compatible with our synthesis software. This allows for a more effective and accurate synthesis process.

In the context of this specific use case, we have introduced a specialized node called a "parallelization node". This node is visually represented by a parallelogram and serves the purpose of consolidating parallel paths within the graph into a single entity. The primary objective is to simplify the overall structure of the original graph to enable CLS to synthesize these structures. Each parallelization node encompasses a comprehensive list of all paths originating from the starting point and converging at the end of the respective parallel branch. These paths can be categorized as either forward paths, illustrating the progression of material flow, or backward paths, signifying the return of material flow to the initial branching point. This inclusive approach enables the representation of cyclic structures within the graph. Furthermore, it is worth noting that a path within a parallelization node can itself involve additional parallelization nodes. This hierarchical arrangement allows for a flexible and expressive representation of complex relationships and parallel processes within the synthesized graph. In our representation, these parallelization nodes therefore replace structures such as cycles without changing the semantics of the graph.

An illustrative depiction of the graph from Figure 3, taking into account the presence of the parallelization node, is featured in Figure 4. In this representation, the three distinct cycles associated with each machine are meticulously captured through dedicated parallelization nodes, with the overarching parallelization node encapsulating the broader main cycle. This structural adjustment ensures that, on a fundamental node-based level, the graph is now devoid of cyclic dependencies, rendering it amenable to the synthesis process.

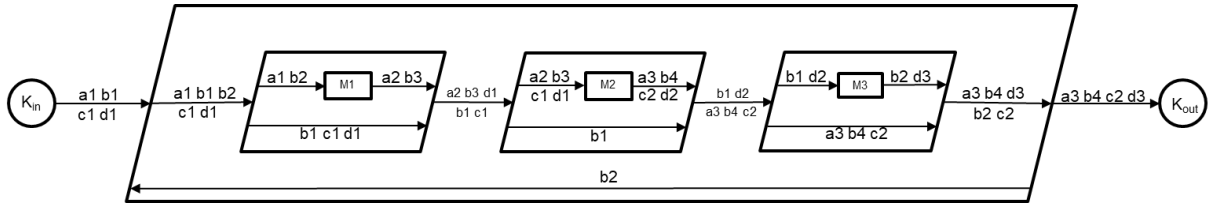

Figure 4: Graph with parallel nodes.

The upcoming Figure 5 delves into a comprehensive exploration of the implementation details, offering a thorough understanding of the applied methodologies and their implications.

As seen in Figure 1, the input to the CLS-Framework consists of a repository of combinators together with a target type (system boundaries specification). The repository consists of general structural combinators and case-specific combinators like machines. In our benchmark system, the structural node combinators allow the construction of graphs as described above and consists of combinators corresponding to the sequential or parallel combination of sub-graphs with the option of backwards paths. The case-specific combinators then correspond to the machines (M1, M2, M3). The target type is a description of the sources and sinks. The number of different machines and corresponding combinators is finite, however, the structural combinators only follow some kind of constraints to ensure they respect reasonable conditions w. r. t. the inputs and outputs of the subgraphs. To solve this, first, a set L of all relevant edge labels (i.e., inputs and outputs) is constructed, and then substituted into the variables of the structural combinators (see above). The repository uses two type constructors: Consume(x) and Produce(x), describing the set x of inputs/outputs to the subgraph constructed by the typed combinator. The structural and machine combinators will be described in detail below. Figure 5 depicts the formal specification of these combinators. The first part defines the pattern in which CLS can select subgraphs for the respective structure (enclosed in ⟨ ⟩), followed by the path conditions (literals) that must also be satisfied after the next arrow.

Sequential combination:

$$seq : \langle l_1 : L\rangle \Rightarrow \langle l_2 : L\rangle \Rightarrow \langle l_3 : L\rangle \Rightarrow Consume(l_1) \cap Produce(l_2) \Rightarrow Consume(l_2) \cap Produce(l_3) \Rightarrow Consume(l_1) \cap Produce(l_3)$$

Parallel combination:

$$par : \langle l_1 : L\rangle \Rightarrow \langle l_2 : L\rangle \Rightarrow \langle l_3 : L\rangle \Rightarrow \langle l_4 : L\rangle \Rightarrow \langle l_{in} : L\rangle \Rightarrow \langle l_{out} : L\rangle \Rightarrow l_{in} = l_1 \cup l_3 \Rightarrow l_{out} = l_2 \cup l_4 \Rightarrow$$
$$Consume(l_1) \cap Produce(l_2) \Rightarrow Consume(l_3) \cap Produce(l_4) \Rightarrow Consume(l_{in}) \cap Produce(l_{out})$$

Backwards path inclusion:

$$back : \langle l_1 : L\rangle \Rightarrow \langle l_2 : L\rangle \Rightarrow l_1 \cap l_2 \neq \emptyset \Rightarrow \langle l_{in} : L\rangle \Rightarrow l_{in} = \{x | x \in l_1 \wedge x \notin l_2\} \Rightarrow \langle l_{out} : L\rangle \Rightarrow l_{out} = \{x | x \in l_2 \wedge x \notin l_1\} \Rightarrow$$
$$Consume(l_1) \cap Produce(l_2) \Rightarrow Consume(l_{in}) \cap Produce(l_{out})$$

Forwards path inclusion:

$$forw : \langle l_1 : L\rangle \Rightarrow \langle l_2 : L\rangle \Rightarrow \langle new : L\rangle \Rightarrow new \not\subseteq l_1 \vee new \not\subseteq l_2 \Rightarrow \langle l_{in} : L\rangle \Rightarrow l_{in} = \{x | x \in new \vee x \in l_1\} \Rightarrow$$
$$\langle l_{out} : L\rangle \Rightarrow l_{out} = \{x | x \in new \vee x \in l_2\} \Rightarrow Consume(l_1) \cap Produce(l_2) \Rightarrow Consume(l_{in}) \cap Produce(l_{out})$$

Figure 5: Notation for combinators.

For example, for every machine M, with inputs I and outputs O, a combinator with the type Consume(I) ∩ Produce(O) is added to the repository. The implementation of a machine combinatory simply places the corresponding machine.

Two subgraphs can be concatenated (combinator *seq*) if, and only if, all of the outputs of the first one correspond to all of the inputs of the second one and the resulting graph then consumes the inputs of the first subgraph and produces the outputs of the second subgraph. The implementation of a sequential combinator places the implementation of its subgraphs next to each other and connects them with a conveyor that can transport the intermediate product(s). Two subgraphs can always be placed in parallel (combinator *par*) and the resulting graph consumes the combined inputs of the two, and produces the combined output. The implementation of a parallel combinator places the implementation of its subgraphs next to each other and connects them with a turntable. If one or both subgraphs already are parallel compositions, the implementation of the other subgraph can be added to that composition, or they can be combined, respectively.

A backwards path (combinator *back*) can be added to any subgraph, where the input and output have some intersection. Feeding back the output into the input using a backwards path, the resulting graph only requires the inputs that it cannot produce itself. The implementation of this combinator works the same way as the implementation for the parallel composition. The second subgraph consists of a backwards transport conveyor and the turntables must be able to handle multiple inputs and outputs. If the subgraph to which the backward path is added is already a parallel composition, a backward transport can simply be added to that composition. Forward paths (combinator *forw*) can also be used in the same way as backward paths. Here, however, the edge labels are not subject to such strict restrictions. Instead, forward paths can be used to route individual materials past nodes on a parallel path.

As eluded in section 1 and section 3, the described repository and synthesis have been implemented using the Python implementation of the CLS framework. Our implementation is available at https://ls14-scm.cs.tu-dortmund.de/kl4sim_publications/wsc24.git. For information about the CLSP Python library see https://github.com/tudo-seal/clsp-python. The repository consists of exactly the structural combinators as they are described above and the machine combinators from Figure 6.

M1: $Consume(a_1 \cap b_2) \Rightarrow Produce(a_2 \cap b_3)$
M2: $Consume(a_2 \cap b_3 \cap c_1 \cap d_1) \Rightarrow Produce(a_3 \cap b_4 \cap c_2 \cap d_2)$
M3: $Consume(b_1 \cap d_2) \Rightarrow Produce(b_2 \cap d_3)$

Figure 6: Structural combinators.

Our implementation successfully synthesizes two representations of the graph in Figure 3, namely the ones shown in Figure 4 and in Figure 7. It also generates an enumerable infinite set of other solutions, a selection of those can be seen in Figure 8. These solutions can be seen as structural variants of the original model. By generating these variants, we can automatically cover a wide range of possible modeler's choices. Through experimental execution of the individual variants, the most advantageous one can be determined, thereby identifying potential for structural improvements.

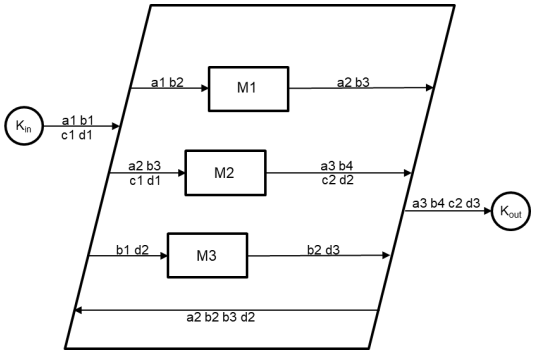

Figure 7: Generated solution.

A key challenge with the current implementation is the large amount of time required for synthesis, specifically constructing a grammar that enumerates the solutions. For the example above, the current implementation requires at least one hour in our experimental setup to construct the grammar. This is primarily caused by the large theoretical complexity of combinatory logic synthesis, as explained in detail in Bessai (2019). Still, some improvements to the CLSP implementation or a better or more exact specification of our model should enable large improvements for the runtime. Improving the synthesis time of CLS is part of the ongoing research progress on CLS.

Another challenge consists of the aforementioned improvement of the specification of the model using semantic intersection types. Amongst many other uses, this could enable prohibiting or enforcing the usage of some machines, or even allowing the user to specify the number of times a machine should be present in the synthesis result (either via an exact number, or a more sophisticated predicate). This would also allow for a more exact and sophisticated notion of redundancy w. r. t. structural variants (e.g., to decide if the two variants from Figure 8 should be considered structurally different to each other, and respectively, if both should be generated).

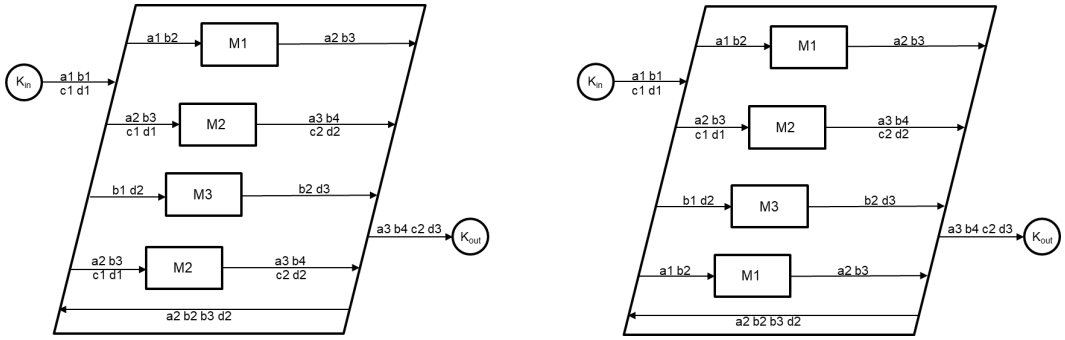

Figure 8: Synthesized structural variants.

Additionally, there exist some theoretical graphs that conform to the general graph model explained above, that cannot be easily represented in a cycle-free tree-structure using just parallelization nodes. We have investigated multiple approaches to resolve this, e.g., duplicating some nodes in the representation. However, these must be researched further to increase the number of MFSs that are representable using our approach.

6 RESULTS AND FURTHER WORK

Our paper is first contextualized in the state of the art in reducing efforts associated with the design and execution of simulation experiments. The paper uses a case study to illustrate how structural variants can be automatically generated based on an initial valid simulation model. Using the CLS-Framework, structural variants are automatically synthesized (taking into account a target description). The starting point and intermediate result of the synthesis are graph models of the simulation model and its structural variants. Using a connector, the synthesis result is translated into AnyLogic XML and is thus directly available to the user in the simulation tool.

Although the current solution is already able to automatically generate structural variants based on the relevant conversion-capable operators (see above), there are further research challenges, which are explained below. Adding and removing material flow system elements creates distribution or merging points, which require corresponding control strategies. Examples of distribution strategies are the targeted distribution of objects according to their object type or the percentage distribution of objects. Examples of merging strategies are first in first out (*FIFO*) or prioritizing inputs. In the case of merging according to FIFO, the control strategy can be applied automatically for any number of inputs that are added or removed by the synthesis with CLS. However, in the case of more complex user-specific strategies, variant-related strategies cannot be automatically synthesized unless additional synthesis rules are specified.

Furthermore, we recognize that our current approach is primarily focused on the application in AnyLogic 8 (the result of the procedure model outlined in section 4 is a direct modification of the AnyLogic XML file) and that other simulation tools will have to be connected to the developed methodology in order to reach a wide range of users. This connection is technically feasible, as the formalism of the CLS graph model works on the basis of fundamental material flow structures that can basically be modeled in all relevant simulation tools for the investigation of MFSs.

Further research needs are identified regarding the complexity and expressivity of the synthesis and the evaluation of structural variants. By complexity and expressiveness, we refer to the further development of our approach to allow more precise specification of the desired variants, e.g., through constraints. In addition, we aim to generate more complex structures and synthesize more sophisticated components, including distribution strategies. Although our methodology reduces the effort required to build structural simulation model variants, these structural variants only really serve their purpose when they are used in simulation experiments. For this purpose, the evaluation of structural variants is performance-related (e.g., throughput or utilization rate) based on the experiments. However, it should be expanded to include further criteria with regard to structure-related changeability. For this purpose, Sutherland et al. (2024) offer concrete criteria, some of which, such as determining the proportion of modular or redundant elements, can already be applied statically (i.e., without running the simulation model). These criteria are to be further developed into a superordinate evaluation methodology in order to enable a differentiated evaluation of structural variants of changeable MFSs.

FUNDING ACKNOWLEDGEMENT

This article was written as part of the project “KL4SiM – automated generation of structural variants for simulation models in the field of production and logistics using combinatorial logic” (funded by the German Research Foundation (DFG) – 511349842).

REFERENCES

Barendregt, H., W. Dekkers, and R. Statman. 2013. "Lambda calculus with types*"*. Cambridge University Press

Bessai, J. 2019. "A type-theoretic framework for software component synthesis". TU Dortmund

Bessai, J., A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. 2016. "Combinatory Process Synthesis". In: *Proceedings of 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques, ISoLA*, Imperial, Corfu, Greece, Part I, 266-281, https://doi.org/10.1007/978-3-319-47166-219

Bessai, J., A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. 2014. "Combinatory Logic Synthesizer". In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, edited by T. Margaria and B. Steffen, 26-40, Berlin, Heidelberg: Springer.

Bicalho-Hoch, A. L., F. Özkul, N. Wittine, and S. Wenzel. 2022. "A Tool-Based Approach to Assess Simulation Worthiness and Specify Sponsor Needs for SMEs". In *2022 Winter Simulation Conference (WSC)*, 1818-1829, https://doi.org/10.1109/WSC57314.2022.10015373.

Cisek, R., C. Habicht, and P. Neise. 2002. "Gestaltung wandlungsfähiger Produktionssysteme". *Zeitschrift für wirtschaftlichen Produktbetrieb* 97(9): 441-445

Curry, H. 1934. "Functionality in combinatory logic" In *Proceedings of the National Academy of Sciences*, 20(11):584-590.,

Howard, W. 1980. "The formulae-as-types notion of construction". *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479-490.

Düdder, B., M. Martens, J. Rehof, and P. Urzyczyn. 2012. "Bounded Combinatory Logic" In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*. Leibniz International Proceedings in Informatics (LIPIcs), Vol- 16, 243-258, Schloss Dagstuhl - Leibniz-Zentrum für Informatik https://doi.org/10.4230/LIPIcs.CSL.2012.243

Feng, B. and G. Jiang. 2020. "Reusing Simulation Outputs of Repeated Experiments Via Likelihood Ratio Regression". In*: 2020 Winter Simulation Conference (WSC)*, 325-336, https://doi.org/10.1109/WSC48552.2020.9383879

Gulwani, S. 2010. "Dimensions in program synthesis". In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. 13-24.

Heineman, G., A. Hoxha, B., Düdder and J. Rehof. 2015 "Towards migrating object-oriented frameworks to enable synthesis of product line members". In *Proceedings of the 19th International Conference on Software Product Line*, 56-60.

Hindley, J. R. and J. Seldin. 2008. "Lambda-calculus and combinators: an introduction" Cambridge University Press.

Jain, S., and D. Lechevalier. 2016. "Standards based generation of a virtual factory model". In 2016 Winter Simulation Conference (WSC), 2762-2773. IEEE.

Kallat, F. 2023. "Komponentenbasierte Synthese von Simulationsmodellen". Dortmund: Universitätsbibliothek Dortmund.

Kallat, F., C, Mieth, J. Rehof, and A. Meyer. 2020. "Using component-based software synthesis and constraint solving to generate sets of manufacturing simulation models". *Procedia CIRP* 93:556-561.

Kallat, F., J. Pfrommer, J. Bessai, J. Rehof, and A. Meyer. 2021. "Automatic building of a repository for component-based synthesis of warehouse simulation models". *Procedia CIRP* 104:1440-1445.

Kassen, S., H. Tammen, M. Zarte, and A. Pechmann. 2021. "Concept and case study for a generic simulation as a digital shadow to be used for production optimization". *Processes* 9 (8):1362-1380.

Lattner, A. D., T. Bogon, Y. Lorion, and I. J. Timm. 2010. "A knowledge-based approach to automated simulation model adaptation". In *Proceedings of the 2010 Spring Simulation Multiconference*, edited by R. McGraw, E. Imsand, and M. J. Chinni, 1-8. San Diego: Society for Computer Simulation International.

Law, A. M. 2019. "How to Build Valid and Credible Simulation Models". In *2019 Winter Simulation Conference (WSC)*, 1402-1414, https://doi.org/10.1109/WSC40007.2019.9004789.

Mages, A., C. Mieth, J. Hetzler, F. Kallat, J. Rehof, C. Riest, and T. Schäfer. 2022. "Automatic component-based synthesis of user-configured manufacturing simulation models". In *2022 Winter Simulation Conference (WSC)*, 1841-1852, https://doi.org/10.1109/WSC57314.2022.10015425.

Neyrinck, A., A. Lechler, and A. Verl. 2015. "Automatic Variant Configuration and Generation of Simulation Models for Comparison of Plant and Machinery Variants". *Procedia CIRP* 29:62-67.

Özkul, F., R. Sutherland, S. Wenzel, and S. Spieckermann. 2023. "Einsatz von Process-Mining zur Verifikation und Validierung von Simulationsmodellen in Produktion und Logistik". In *20. ASIM Fachtagung Simulation in Produktion und Logistik*, edited by S. Bergmann, N. Feldkamp, R. Souren and S. Straßburger, 463-472, Ilmenau: TU Ilmenau Universitätsbibliothek.

Peng, D., T. Warnke, F. Haack, and A. M. Uhrmacher. 2017. "Reusing simulation experiment specifications in developing models by successive composition – a case study of the Wnt/ β -catenin signaling pathway". *SIMULATION* 93(8):659-677.

Rabe, M., S. Spieckermann, and S. Wenzel. 2008. "A new Procedure Model for Verification and Validation in Production and Logistics Simulation". In *2008 Winter Simulation Conference (WSC)*, 1717-1726, https://doi.org/10.1109/WSC.2008.4736258.

Reinhardt, H., M. Weber, and M. Putz. 2019. "A Survey on Automatic Model Generation for Material Flow Simulation in Discrete Manufacturing". *Procedia CIRP* 81:121-126.

Rehof, J. and M. Y. Vardi. 2014. "Design and synthesis from components (Dagstuhl seminar 14232)". *Dagstuhl Reports*.

Ruscheinski, A., K. Budde, T. Warnke, P. Wilsdorf, B. C. Hiller, M. Dombrowsky, and A. M. Uhrmacher. 2018. "Generating Simulation Experiments Based on Model Documentations and Templates". In *2018 Winter Simulation Conference (WSC),* 715-726, https://doi.org/ 10.1109/WSC.2018.8632515

Schlecht, M., R. de Guio, and J. Köbler. 2023. "Automated generation of simulation model in context of industry 4.0". *International Journal of Modelling and Simulation* 1-13.

Sutherland, R., F. Özkul, L. Grusie, S. Wenzel, J. Winkels, J. Löhn, and J. Rehof. 2024. "Strukturvarianz in wandlungsfähigen Produktions- und Logistiksystemen". *Zeitschrift für wirtschaftlichen Fabrikbetrieb* 119:141-145.

Teran-Somohano, A., A. E. Smith, J. Ledet, L. Yilmaz, and H. Oguztuzun. 2015. "A model-driven engineering approach to simulation experiment design and execution". In *2015 Winter Simulation Conference (WSC)*, 2632-2643, https://doi.org/10.1109/WSC.2015.7408371

Wenzel, S. J. Stolipin, J. Rehof, and J. Winkels. 2019. "Trends in Automatic Composition of Structures for Simulation Models in Production and Logistics". In *2019 Winter Simulation Conference (WSC)*, 2190-2200, https://doi.org/10.1109/WSC40007.2019.9004959

Wiendahl, H. P. and C. L. Heger. 2004. "Justifying Changeability. A Methodical Approach to Achieving Cost Effectiveness". *Journal for Manufacturing Science and Production* 6(1-2):33-40.

Wilsdorf, P., M. Dombrowsky, A. M. Uhrmacher, J. Zimmermann, and U. van Rienen. 2019. "Simulation Experiment Schemas – Beyond Tools and Simulation Approaches". In *2019 Winter Simulation Conference (WSC)*, 2783-2794, https://doi.org/10.1109/WSC40007.2019.9004710

Wilsdorf, P., N. Fischer, F. Haack, and A. M. Uhrmacher. 2021. "Exploiting Provenance and Ontologies In Supporting Best Practices For Simulation Experiments: A Case Study On Sensitivity Analysis". In *2021 Winter Simulation Conference (WSC)*, 1-12, https://doi.org/10.1109/WSC52266.2021.9715362

Wilsdorf, P., J. Heller, K. Budde, J. Zimmermann, T. Warnke, C. Haubelt *et al.* 2022. "A Model-Driven Approach for Conducting Simulation Experiments". *Applied Sciences* 12(16):7977.

Wilsdorf, P., A. Wolpers, J. Hilton, F. Haack, and A. M. Uhrmacher. 2023. "Automatic Reuse, Adaption, and Execution of Simulation Experiments via Provenance Patterns". *ACM Transactions Modelling and Computer Simulation* 33(1-2):1-27.

AUTHOR BIOGRAPHIES

JAN WINKELS is a post-doctoral researcher at the Department of Computer Science at TU Dortmund University. His research focusses on component-based software synthesis. His email address is jan.winkels@tu-dortmund.de.

FELIX ÖZKUL is a research associate and PhD candidate at the Department Organization of Production and Factory Planning (pfp), University of Kassel. His research primarily focusses on the use of discrete-event simulation and process mining in production and logistics systems. His email address is felix.oezkul@uni-kassel.de.

ROBIN SUTHERLAND is a research associate and PhD candidate at the Department Organization of Production and Factory Planning (pfp), University of Kassel. His work focusses on the design of changeable production networks and the structural variance of simulation models. His email address is robin.sutherland@uni-kassel.de.

JANNIK LÖHN studies computer science at TU Dortmund University and works as a student assistant at the Chair of Software Engineering at TU Dortmund University. His E-Mail Address is jannik.loehn@tu-dortmund.de.

SIGRID WENZEL is Professor and head of the Department Organization of Production and Factory Planning (pfp), University of Kassel. In addition to this, she is a board director of the Arbeitsgemeinschaft Simulation (ASIM), spokesperson for the ASIM working group Simulation in Production and Logistics, member of the advisory board of the Association of German Engineers Society of Production and Logistics (VDI-GPL), and head of the Committee Modeling and Simulation of the VDI-GPL. Her email address is s.wenzel@uni-kassel.de.

JAKOB REHOF is professor at the Department of Computer Science at TU Dortmund University. He heads the Chair of Software Engineering and is Director of the Lamarr Institute for Machine Learning and Artificial Intelligence in Dortmund, Germany. His email address is jakob.rehof@tu-dortmund.de.