A Knowledge-based Approach to Automated Simulation Model Adaptation

Andreas D. Lattner, Tjorben Bogon, Yann Lorion, and Ingo J. Timm
Information Systems and Simulation, Institute of Computer Science
Goethe University Frankfurt, P.O. Box 11 19 32, 60054 Frankfurt/Main, Germany
{lattner|bogon|lorion|timm}@cs.uni-frankfurt.de

Keywords: Simulation Model Adaptation, Optimization, Knowledge-based Modeling, Logic Programming.

Abstract
Simulation has become a widely accepted technology for analyzing or planning systems in various domains. In production logistics, for instance, many companies use simulation to evaluate scenarios before actually the construction or modifications of the production hall or processes are performed in order to get insights about the performance of planned configurations. In this paper, we propose an approach to knowledge-based adaptation of simulation models. The vision of this work is to go one step beyond parameter optimization, namely to provide means for automated structural changes in simulation models, and thus for the generation of simulation model variants. For a first evaluation of our approach, we introduce a system consisting of a simulation control as well as a model adaptation module with a set of adaptation operators. Our implementation is coupled to the simulation system Plant Simulation in order to perform simulation runs. For illustration we apply our system to a test scenario and present first results.

1. INTRODUCTION
Simulation has become a widely accepted technology for analyzing or planning systems in various domains. In production logistics, for instance, many companies use simulation to evaluate scenarios before actually the construction or modifications of the production hall or processes are performed in order to get insights about the performance of planned configurations.

One of the most important underlying goals of these investigations is to identify variants that lead to good solutions w.r.t. certain criteria like minimal makespan, maximal profit, maximal quality, or a combination of such factors. One problem of larger scenarios is that it is not tractable to enumerate and evaluate all possible variations.

In the fields of Artificial Intelligence and Operations Research many different optimization technologies have been proposed which address the problem of search in huge solution spaces. The application of such methods to simulation (or even the integration into the simulation system) in order to find good parameter values for a defined target function has been addressed by various researchers (for a review see [3]).

In this paper, we propose an approach to knowledge-based adaptation of simulation models. The vision of this work is to go one step beyond parameter optimization, namely to provide means for automated structural changes in simulation models, and thus for the generation of simulation model variants. In order to allow for automating this task, a description of the simulation model components as well as the definition of adaptation operators are needed. Such an adaptation makes sense in scenarios where components are added, or replaced, e.g., replacing a machine in the model by another type or adding new conveyors.

The remainder of the paper is structured as follows: Section 2 discusses work related to ours. In Section 3 we present the architecture of our approach and discuss requirements for the representation of model components and background knowledge. The automated adaptation of models including neighborhood operators and search strategies are addressed in Section 4. In Section 5 we introduce an initial implementation of our approach and present the application of the system to a simple scenario. In Section 6 we close the paper with a conclusion and ideas for future work.

2. RELATED WORK
With respect to the considerable investments in adapting or establishing new production lines to a manufacturing assembly line Qiao et al. [5] present an approach to a data driven manufacturing design and simulation system. Based on a Shop Data Information Model that contains information regarding orders, manufacturing, negotiation, resources and management a scenario manager combines and reconfigures function groups, each representing a typical manufacturing capability. This model describes the content of XML-based Shop Data Files which can contain elements of manufacturing operations or can be used as interchange format between manufacturing applications. Certain validity properties of the model can be verified using Petri Net methodology. A simulation generator transforms the model into an input for the simulation tool DELMIA QUEST. Depending on the output of the simulation the scenario manager can be used for modifications and a new simulation model can be generated. The adjustment of the layout of a manufacturing assembly line is applied as example of use.

During the last decade the use of ontologies in the modeling and simulation domain has been increasingly discussed
Figure 1. Knowledge-based Adaptation of Simulation Models

The main motivation of these papers is the assistance and facilitation of simulation modeling. It allows reutilization and composition of (distributed) sub models as well as consistency validation and may lead to a vision of a dynamic construction of simulations using simulation web services [4].

Silver et al. present an approach for the ontology-based construction of simulation models [8]. They use two independent ontologies for the domain and the simulation model. These ontologies are then linked by mapping entities from the domain ontology to elements of the modeling ontology. Once linked, ontology instances are created which can be transformed to executable simulation models (indirect, using a XML-representation). Within the context of this paper the Discrete Event Modeling Ontology (DeMO) as well as an Ontology Driven Simulation design tool (ODS) have been developed for discrete event based simulation. Executable simulation models can be generated for JSIM and ARENA. The benefits of the resulting interoperability are discussed in [7].

Sakthivel and Agarval present an approach to knowledge-based model construction of information systems using Prolog [6]. They propose a two-step process: In the first step the information system is modeled as a Petri Net. In the second step, the Petri Net is used to generate a simulation model automatically.

The major difference of our approach compared to the ones mentioned above is that we aim at the automated adaptation of simulation model, i.e., the captured knowledge about models should not only be used for retrieval, reuse, transformation or validity checks of (sub-) models but actually for changing models in order to identify good model variants. There are many approaches that optimize parameters of a given simulation model (cf. [3]). In this paper, we propose that structural

changes of a model should be taken into account in such an optimization process as well.

3. ARCHITECTURE AND COMPONENT REPRESENTATION

The basic idea of knowledge-based adaptation of simulation models is to provide means for an automated generation of model variants for a given model. The overall architecture of the approach can be seen in Fig. 1. The process of simulation model adaptation is controlled by the Simulation Control component. It controls the simulation software, i.e., it provides the model to be used, starts the simulation run (repeatedly for different random seed values), and evaluates the simulation results. The Model Adaptation component creates valid variants for a given model and returns these variants to the simulation control. A converter transforms the simulation models to the internal format and also generates specific model output for the simulation software for all generated variants. The Knowledge Base consists of information about the components of the simulation model. More generally speaking, it holds information about existing and potential components to be used in the model as well as permitted relationships. This captured knowledge is used in order to generate valid variants of a given model (or a set of given models).

The knowledge base holds the relevant information about model adaptation. In order to represent different components that can be used or exchanged, it is necessary to setup a description model describing different component types as well as their properties. The component types are general descriptions on a schema level. A concept hierarchy describes the is-a relations among the different component types. An example for such a hierarchy is given in Fig. 2. The most general object type is component. As illustrated in the example, various types at different refinement levels can be defined. Which types have to be taken into account depends on the domain. If only some part of the simulation model is relevant for automated adaptation, a partial gathering of the corresponding model will be sufficient.

For each component in the concept hierarchy it is necessary to capture relevant properties (on a schema level once again),
4. AUTOMATED MODEL ADAPTATION

In order to operationalize the adaptation of simulation models, we propose to setup a set of neighborhood operations and search strategies working on the solution space of valid (w.r.t. the captured model information) models.

4.1. Neighborhood Operations

The neighborhood operations act on an existing (working) model. Each operation is defined in a way that one aspect of the model is changed. These changes can be of different kinds:

- Add or remove a component instance of the model.
- Replace on component instance by another compatible component instance.
- Change one parameter of an existing component instance of the model.

While the last kind of change can be seen as normal parameter tuning of an existing model, the other two intervene with the model structure. If these change operations are applied, it will have to be ensured that the resulting model is still able to perform the underlying task. For instance, removing all stations which can be used to perform a certain needed procedure for a production scenario would violate this requirement. Even if such an automated proof was not possible, simple test cases could be applied, e.g., check if producing one single job of all products of the portfolio is still working.

In order to add a component to the model, allowed positions for the new component have to be identified. Therefore, the model with connected components has to be analyzed for potential insertion positions, e.g., adding a new parallel station to an existing one.

For a valid replacement of a component, it has to be checked, if the new component can handle all (relevant) procedures of the previous one. Relevant in this context means that if there is a certain procedure lacking in comparison to the previous component but not needed for the current production plan, such replacements could be allowed if desired.

A helpful approach should support the definition of some generic neighborhood operations, e.g., the operation for the identification of possible replacements of components should be generic and thus independent of concrete station types. This reduces the modeling effort to capturing component specific parts (e.g., about machines and their supported procedures) and allows for using off-the-shelf neighborhood operations.

4.2. Search in the Simulation Model Space

If a set of neighborhood operations exists, a certain strategy is needed how to apply these operations. As we focus on optimization tasks in this paper, i.e., finding a model variant which optimizes a target function, this strategy can be seen as a search in the space of potential simulation models.

For small models and a small number of adaptation steps, a complete search can be performed. In the general case, a complete search to find the optimum is not tractable due to the high number of possible combinations. In order to perform searches even for high-dimensional problems, many different metaheuristics like Genetic Algorithms, Simulated Annealing, Tabu Search, or Particle Swarm Optimization have been developed (see e.g., [10]). Although these approaches do not guarantee to find the optimum, usually they identify good solutions in reasonable time.

For optimization of complex simulation models via model adaptation as sketched above, metaheuristics have to be applied, even if certain (invalid) variants can be avoided by well-defined neighborhood operations or simple validity tests. In this case, the Simulation Control in Fig. 1 implements such a metaheuristic search by using the adaptation approach to generate neighborhood solutions. The fitness of the model variants would be evaluated by running a number of experiments (replications) for each variant.

5. CASE STUDY

For a first evaluation of our approach, we have set up a simple scenario, a set of neighborhood operations for model adaptation, and implemented a straightforward neighborhood-based search. Our implementation uses Plant Simulation\(^1\) as simulation system. In the following, we describe the initial simulation model (and supporting components), the generation of variants via a Prolog program, simulation control with Dynamic Data Exchange (DDE), as well as initial results.

5.1. Simulation Model

Figure 3 shows the basic Plant Simulation model of our case study. This model contains only a source and a sink. Providing these components as a frame for our first evaluation is due to pragmatic reasons for easy interaction with our simulation control. We need to access the outcome of the sink

\(^1\)http://www.emplant.de/english/
and to set the seed value in the source. If these elements were not available in the model, some of our helper methods in Plant Simulation would not find them and would generate a failure. The method to load the model (loadModellFromFile) copies the content of the Prolog generated file into an empty method and calls this new filled method afterwards in order to build up the simulation model. The result for loading the initial model with one buffer and one station can be seen in Fig. 4. In our scenario, three different fictitious machines can be used as a station: *mm15, mm17, nn28*. The machines provide different procedures and have different processing times.

In order to control the simulation system from an application two methods are built in the simulation model: The start (START) and clean-up (CLEAN) methods. To start a simulation experiment, the start method executes the loadModelFromFile and sets the stopping criterion. After the simulation stops (in our case after one simulation day), the outcome of the sink is stored into a variable (OUTPUT) and the system waits for order. The clean-up method resets the simulation run and deletes the current model. The delete method depends on the loaded model and is automatically generated by Prolog, too. The timing method is used to define the timing behavior of the source. It creates timing intervals based on a (random) exponential distribution. The seed value for the random distribution is taken from the seed variable (SEED) in order to get different behavior for the input stream at the source for different replication runs.

### 5.2. Knowledge-based Generation of Variants using Prolog

The module for a knowledge-based generation of variants has been implemented with XSB Prolog [9] – a freely available logic programming and deductive database system developed (among others) at the Computer Science Department of the Stony Brook University, New York, USA, hosted by Sourceforge².

---

insertStation(model(DefList,LinkList,PropList), NewModel) :-
    writeln(model(DefList,LinkList,PropList)),
    member(obj(X,station),DefList),
    member(link(X,Z),LinkList),
    getNextIdForComponent(station, model(DefList,LinkList,PropList),NewId),
    member(prop(X,type,Type),PropList),
    NewModel = model([obj(NewId,station)|DefList],
    [link(Y,NewId)|LinkList],
    [prop(NewId,type,Type)|PropList]).

increaseBufferCapacity(model(DefList,LinkList,PropList), NewModel) :-
    writeln(model(DefList,LinkList,PropList)),
    member(obj(X,buffer),DefList),
    member(prop(X,capacity,Size),PropList),
    SizePlus1 is (Size + 1),
    removeElement(prop(X,capacity,Size),PropList),
    NewModel = model(DefList,LinkList,
    [prop(X,capacity,SizePlus1)|TmpPropList]).

replaceStation(model(DefList,LinkList,PropList), NewModel) :-
    writeln(model(DefList,LinkList,PropList)),
    member(obj(X,station),DefList),
    subsumesProcedures(NewType,Type),
    removeElement(prop(X,type,Type),PropList),
    NewModel = model(DefList,LinkList,
    [prop(X,type,NewType)|PropList]).

createNeighbor(Model, NewModel) :-
    insertStation(Model, NewModel).
createNeighbor(Model, NewModel) :-
    increaseBufferCapacity(Model, NewModel).
createNeighbor(Model, NewModel) :-
    replaceStation(Model, NewModel).

Figure 5. Prolog snippet for variant generation

- Replace a station by a compatible one.

The implementation of these three operations are shown in Fig. 5. The clause for adding a station checks for positions in the model where a new station can be inserted. The model is then extended by copying a station of the same type and linking it to the material flow in the model. Increasing the buffer capacity creates a new model where the capacity for one existing buffer is increased by one. The clause to replace a station uses another clause which checks for compatible stations (i.e., those stations that can perform at least the same procedures as the one to be replaced). In our example, only \textit{mm17} is identified as a valid replacement of \textit{mm15}.

The uniform \texttt{createNeighbor} clauses capture the alternatives for model adaptation. For a given model, all possible model variants for a set of given neighborhood operations are generated. For the model presented above, the automatically generated model variants are (additional station, increased buffer capacity, replaced station):

\begin{verbatim}
model([obj(src1,source), obj(snk1,sink),
    obj(b1,buffer), obj(s1,station)],
    [link(src1,b1), link(b1,s1),
    link(s1,snk1)],
    [prop(b1,capacity,6),
    prop(s1,type,mm15)]),

model([obj(src1,source), obj(snk1,sink),
    obj(b1,buffer), obj(s1,station)],
    [link(src1,b1), link(b1,s1),
    link(s1,snk1)],
    [prop(s1,type,mm17),
    prop(b1,capacity,5)])
\end{verbatim}

As mentioned above, the Prolog program generates input to Plant Simulation in order to build up the corresponding models. The generated Plant Simulation input for the initial model (Fig. 4) is shown in Fig. 6. The first part of the SimTalk code declares all used variables. In the second part, all objects are created ("createObject") and linked ("connect"). The final entries set different attributes of components ("Capacity" is the capacity of the buffer and "ProcTime" is the processing time).

Some code lines are commented out (indicated by "−−").  “MaterialFlow" is the library containing all components.

```
-- src1 : object;
-- snk1 : object;
b1 : object;
s1 : object;
edge_3 : object;
eave_2 : object;
eedge_1 : object;

createObject(.Models.Frame, 100, 200, "src1");
createObject(.Models.Frame, 200, 200, "b1");
createObject(.Models.Frame, 300, 200, "s1");
createObject(.Models.Frame, 400, 200, "snk1");
snk1.setPosition(400, 200);
src1.setPosition(100, 200);
edg1e_1 := .MaterialFlow.Connector.connect(src1, b1);
edge_2 := .MaterialFlow.Connector.connect(b1, s1);
edge_3 := .MaterialFlow.Connector.connect(s1, snk1);
b1.Capacity := 5;
b1.ProcTime := 10.0;
s1.ProcTime := 60.0000;
end;
```

Figure 6. Generated SimTalk code for Plant Simulation

The uniform \texttt{createNeighbor} clauses capture the alternatives for model adaptation. For a given model, all possible model variants for a set of given neighborhood operations are generated. For the model presented above, the automatically generated model variants are (additional station, increased buffer capacity, replaced station):

\begin{verbatim}
model1([obj(s2,station), obj(src1,source),
    obj(snk1,sink), obj(b1,buffer),
    obj(s1,station)],
    [link(b1,s2), link(s2,snk1),
    link(src1,b1), link(b1,s1),

-- Automatically generated by AdaptSim
-- -----------------------------------
is

src1 : object;
smk1 : object;
b1 : object;
s1 : object;
edge_3 : object;
edge_2 : object;
edge_1 : object;

createObject(.Models.Frame, 100, 200, "src1");
createObject(.Models.Frame, 200, 200, "b1");
createObject(.Models.Frame, 300, 200, "s1");
createObject(.Models.Frame, 400, 200, "snk1");
snk1.setPosition(400, 200);
src1.setPosition(100, 200);
edge_1 := .MaterialFlow.Connector.connect(src1, b1);
edge_2 := .MaterialFlow.Connector.connect(b1, s1);
edge_3 := .MaterialFlow.Connector.connect(s1, snk1);
b1.Capacity := 5;
b1.ProcTime := 10.0;
s1.ProcTime := 60.0000;
end;
```

As mentioned above, the Prolog program generates input to Plant Simulation in order to build up the corresponding models. The generated Plant Simulation input for the initial model (Fig. 4) is shown in Fig. 6. The first part of the SimTalk code declares all used variables. In the second part, all objects are created ("createObject") and linked ("connect"). The final entries set different attributes of components ("Capacity" is the capacity of the buffer and "ProcTime" is the processing time). Some code lines are commented out (indicated by "−−") because one source and one sink are currently fixed components of the model. For these two components, only the position is set ("setPosition").

Another input file after repeated application of neighborhood operations (replacement of station, adding two stations) can be seen in Fig. 7. The corresponding Plant Simulation
The DDE interface enable three types of connections: server to connect, to get and to set data in an opened model. This connection executes a method or a static property on the server and returns the result to the client. The request connection lets the Java program request a server method or operation. The poke connection lets the Java program set the content of a variable.

To connect with these different types of connections we set the DDE client connection topic to System for execute and Data for poke and request.

The control cycle of the program is structured as follows: As a first step we generate the different seed values for the number of replications. This set of seed values is used for the replications of every model. In the next step, we enter the main loop which controls the Plant Simulation input. This loop selects the next Prolog generated model file and copies it to a file where our basic Plant Simulation model expects it. The START command then starts the simulation and it is waited until the run ends. This is currently done by requesting the status variable (STATUS) and waiting for the value indicating the end of the simulation run. The next step is to obtain the outcome value (of the sink) of the experiment by a poke command from the corresponding variable in the model (OUTPUT). This value is stored in our statistics object and the loop repeats for the next replication by changing the seed value in the Plant Simulation model. When all model variants and replications are finished, the statistics component generates a file with all outcome values of the experiments and calculates the median and variance of each model variant. This information is stored in combination with the model variant ID for later use.

5.3. Simulation Control via Java and DDE

The actual control of the simulation system is performed by a Java program. In order to set the right Prolog generated model input for Plant Simulation and to test the outcome of the different model variants we used a Dynamic Data Exchange (DDE) bridge to connect to Plant Simulation. As library for the DDE the Google DDE wrapper for Java jDDE is integrated into our program. Plant Simulation offers a DDE server to connect, to get and to set data in an opened model. The DDE interface enable three types of connections:

1. **execute** This connection executes a method or a static command in Plant Simulation.

2. **request** The request connection lets the Java program get the content of a variable.

3. **poke** If we want to set new content to a variable in Plant Simulation we have to use the poke connection.

The results of the single replication runs as well as a mean value of the model output for (a selection of) the model variants can be seen in Table 1. The best mean output has been generated by variant no. 51 (among others) which has already been shown in Fig. 7 and Fig. 8. This model is using three of the faster machines (nnm17) in parallel and generates an output of 5750.3 on average.

---

```
-- Automatically generated by AdaptSim
-- -----------------------------------

is

s1 : object;
s2 : object;
s3 : object;
snk1 : object;
b1 : object;
s2 : object;
s3 : object;
s1 : object;
snk1 : object;

edge_1 : object;    edge_2 : object;    edge_3 : object;    edge_4 : object;    edge_5 : object;    edge_6 : object;    edge_7 : object;

edge_1 := .MaterialFlow.Connector.connect(b1, s1);
edge_2 := .MaterialFlow.Connector.connect(s1, snk1);
edge_3 := .MaterialFlow.Connector.connect(b1, s2);
edge_4 := .MaterialFlow.Connector.connect(s2, snk1);
edge_5 := .MaterialFlow.Connector.connect(b1, s3);
edge_6 := .MaterialFlow.Connector.connect(s3, snk1);
edge_7 := .MaterialFlow.Connector.connect(s1, snk1);

snk1.setPosition(400, 200);
s2.setPosition(300, 400, "s2");
s3.setPosition(300, 300, "s3");
s1.setPosition(300, 200, "s1");
b1.setPosition(200, 200, "b1");
src1.setPosition(100, 200);

-- -----------------------------------

Figure 7. Generated SimTalk code for Plant Simulation (variant no. 51)

model is presented in Fig. 8. As it can be seen, the processing time (German term “Bearbeitungszeit” in the figure) has been adapted for the replaced machine nmn17.

5.4. Initial Results

For our initial experiment, we started with the simple model and background knowledge as described above. A breadth-first search starts with the initial model and then generates variants w.r.t. the neighborhood operations until a search depth of three. Additionally to the existing model 55 variants are generated. Each variant is run ten times using Plant Simulation with different seed values.

The defined neighborhood operations lead to the intended effects: Stations are added to the model, the processing time for replaced stations is changed, as well as changed buffer capacities without user intervention. The latter can be seen in Fig. 9 where the buffer capacity is set to six (German term “Kapazität” in the figure).

The results of the single replication runs as well as a mean value of the model output for (a selection of) the model variants can be seen in Table 1. The best mean output has been generated by variant no. 51 (among others) which has already been shown in Fig. 7 and Fig. 8. This model is using three of the faster machines (nnm17) in parallel and generates an output of 5750.3 on average.

http://code.google.com/p/jdde/
Figure 8. Model variant no. 51 with three faster stations

Table 1. Mean output of the model variants for 10 replication runs with different random seed values

<table>
<thead>
<tr>
<th>No.</th>
<th>Replication results</th>
<th>Mean output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0</td>
<td>2879.0</td>
</tr>
<tr>
<td>1</td>
<td>5729.0, 5705.0, 5662.0, 5712.0, 5729.0, 5741.0, 5707.0, 5758.0, 5645.0, 5726.0</td>
<td>5711.4</td>
</tr>
<tr>
<td>2</td>
<td>2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0, 2879.0</td>
<td>2879.0</td>
</tr>
<tr>
<td>3</td>
<td>3455.0, 3455.0, 3455.0, 3454.0, 3455.0, 3455.0, 3455.0, 3455.0, 3455.0, 3455.0</td>
<td>3454.9</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>5743.0, 5831.0, 5663.0, 5712.0, 5736.0, 5805.0, 5717.0, 5841.0, 5645.0, 5810.0</td>
<td>5750.3</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6. CONCLUSION AND FUTURE WORK

In this paper we have proposed a knowledge-based approach to automated simulation model adaptation. We have implemented a first system that actually performs such an automated simulation model adaptation for a restricted portion of production logistics simulation and have illustrated how our approach works with a simple scenario.

We believe that providing means for the automated generation and evaluation of structural variants of simulation models would be very helpful in simulation. The system could relieve users from rather monotonous works like variant generation and testing, and might even identify solutions, the simulation user has not been thinking of.

Once the automated simulation model adaptation is realized, the consequential steps are the development of objective functions for the evaluation of different modifications as well as their optimization. Several criteria of the objective function may be determined straightforwardly (with domain and goal specific weights), other, e.g. acceleration of production, may be subject to randomness. Hence repeated simulations have to be performed to estimate the expected values. As simulations can be very time consuming the number of simulation runs should be minimized and techniques for the early detection of unsatisfactory results come in handy.

Different optimization methods have been discussed for simulation: model-free methods, e.g. gradient-based procedures, random search, sample path optimization and metaheuristics or model-based methods like the estimation of distribution algorithm [3]. Most of these methods have in common that they need a well-defined neighborhood operator on the search space or a metric respectively. Therefore a reasonable neighborhood operator has to be developed. In the current status of our work, we have only realized three neighborhood operations; this set definitely has to be extended in future work.

The model and the properties taken into account for the case study presented in this paper are rather simple. The specification of an adequate model description language as well as format transformations for existing simulation systems for more complex models poses another challenge.
ACKNOWLEDGMENT
The content of this paper is a partial result of the AssistSim project (Hessen Agentur Project No.: 185/09-15) which is funded by the European Union (European Regional Development Fund - ERDF) as well as the German State Hesse in context of the Hessen ModellProjekte.

REFERENCES


