Towards Reactive Planning With Digital Twins and Model-Driven Optimization

Martin Eisenberg (✉), Daniel Lehner, Radek Sindelar, Manuel Wimmer

CDL-MINT, Institute of Business Informatics - Software Engineering
Johannes Kepler University Linz, Linz, Austria
`firstname.lastname@jku.at`

Abstract. Digital Twins are emerging in several domains. They allow to connect various models with running systems based on bi-directional data exchange. Thus, design models can be extended with runtime views which also opens the door for many additional techniques such as identifying unexpected system changes during runtime. However, dedicated reactions to these unexpected changes, such as adapting an existing plan which has been computed in advance and may no longer be seen beneficial, are still often neglected in Digital Twins.
To tackle this shortcoming, we propose so-called reactive planning that integrates Digital Twins with planning approaches to react to unforeseen changes during plan execution. In particular, we introduce an extended Digital Twin architecture which allows to integrate existing model-driven optimization frameworks. Based on this integration, we present different strategies how the replanning can be performed by utilizing the information and services available in Digital Twins. We evaluate our approach for a stack allocation case study. This evaluation yields promising results on how to effectively improve existing plans during runtime, but also allows to identify future lines of research in this area.

Keywords: Digital Twin · Planning · Models@Runtime · Optimization.

1 Introduction

According to Kritzinger et al. [24], a Digital Twin (DT) is a digital object representing a physical object, with an automated data flow from the physical to the digital object, and vice versa. This bi-directional data flow can be used to gain insights into the running system and foster decision-making through visualization and prediction, or eventually achieve autonomous decision-making through self-adaptation [15].

Automated planning can be used to achieve such self-adaptation [18]. Automated planning approaches [17] calculate a set of actions, i.e., a plan, that can be executed to shift a system from an initial state to a desired goal state. Usually, such approaches separate the planning which happens offline and the execution of a plan which happens online while the system is running. This however neglects the online uncertainty of changes that might happen in the system in parallel to the realization of the plan [14]. As a result of these changes, (i) parts of the plan may not be executable any more, or (ii) realizing the initial plan leads to an inferior quality in the goal state of the system. In both cases, the old plan does not leverage its initial potential in the running system.

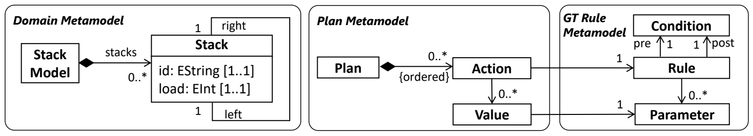
The reactive planning framework proposed in this paper accounts for these unexpected changes during plan execution by providing capabilities for DTs (i) to identify deviations in the running system, and (ii) to enable online replanning to react to these deviations. In particular, we make use of an existing model-driven optimization (MDO) framework [8]. The MDO framework can be leveraged to reason about the runtime states, and take into account the continuous evaluation of the quality of a plan, considering unforeseen changes in the system. To achieve this technical integration of MDO with DTs, we propose the infrastructure for this integration, i.e., the actual runtime model and the expected runtime model of the system at a specific point in time, a conformance checker component that identifies unexpected system changes by comparing the expected runtime model with the actual runtime model, and a decision maker component that decides how to react to these changes. This infrastructure supports three different reactive planning strategies: (i) repairing the old plan by skipping non-executable actions, (ii) stopping the system to calculate a new plan using its current state, with the option to use the old plan as a starting point, and (iii) calculating a new plan in parallel to executing the repaired plan on the system. We investigate the efficiency of these reactive planning strategies using a stack allocation case study. To sum up, the contribution of this work is (i) a reactive planning framework that integrates DTs with MDO, and (ii) an experimental investigation of the three strategies to react to unforeseen changes.

The remainder of the paper is structured as follows. Section 2 introduces a running example and outlines the necessary background. Section 3 presents the proposed reactive planning framework by investigating its architecture and realization. In Section 4, we describe the experimental evaluation of this framework. Section 5 discusses related work, while Section 6 concludes by giving an outlook to future work.

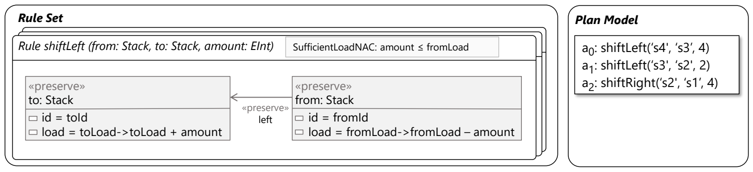
2 Background and Running Example

To make our work more tangible, we use a Stack Load Balancing use case which has been used in previous work on MDO [8] as the running example for this paper. The involved domain concepts are captured in a domain meta-model that is depicted in Fig. 1a. In this use case, a cyber-physical system encompasses several `Stack`s of items. In the context of a production system, these stacks can be thought of as circularly connected machines, each containing a specific quantity of goods, the `load`, to be processed. To simplify this example, we assume that each machine in the system can be treated equally, independently, and processes the same kind of item. The overall productivity depends on the distribution of workload amongst these machines. Therefore, one goal should be to distribute the items as equally as possible between available stacks, i.e., the standard deviation of the item load per stack should be minimized. To achieve this goal, items can be relocated between stacks by shifting them from its original stack to the `left` or `right` neighbour stack. As the transfer can take some time, shorter relocation plans are generally preferred.

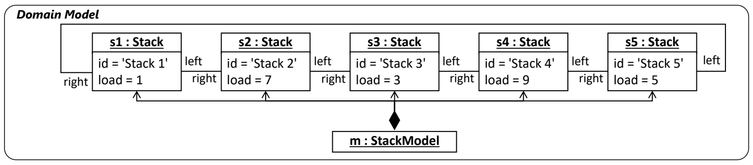
To provide an abstract system representation, the design-time and runtime aspects of such a system are represented using models expressed in a domain-specific modeling language [10]. In a `domain metamodel` (cf. Fig. 1a), the available static concepts (i.e., stacks), their attributes (id, initial load) and relationships (left and right neighbours) are described. Specific operations (e.g., shiftLeft and shiftRight) of a stack can



(a) Metamodels in UML class diagram syntax


Rule Set
Rule shiftLeft (from: Stack, to: Stack, amount: EInt)
SufficientLoadNAC: amount ≤ fromLoad
«preserve»
to: Stack
id = toId
load = toLoad->toLoad + amount
«preserve»
left
«preserve»
from: Stack
id = fromId
load = fromLoad->fromLoad – amount
Plan Model
a0: shiftLeft('s4', 's3', 4)
a1: shiftLeft('s3', 's2', 2)
a2: shiftRight('s2', 's1', 4)

(b) Rule set in Henshin syntax (left) and plan instantiating this rule set (right)



(c) Example Domain Model in UML object diagram syntax

Fig. 1: Artefacts of the running example.

be specified using graph transformation rules [21] (cf. Fig. 1b). These graph transformations use the graph-based structure of domain models (cf. Fig. 1c) to define rules on this graph structure (based on the `GT rule metamodel` (cf. Fig. 1a)). These rules can be executed to change the model accordingly. Henshin [5] is one prominent graph-based transformation framework that supports their specification and execution. In particular, the rules specify pre- and postconditions for their application and effects, respectively, as well as allow for parameters to be bound for a particular rule application. For our running example, Fig. 1b shows the `shiftLeft` rule with parameters `from`, `to`, and `amount` to declare source and target stacks as well as the shifted load.
The `domain model` (cf. Fig. 1c) is an instantiation of the domain metamodel representing specific items and their attribute values. This model can be used for offline planning, but also for online representation of the runtime state of the system [7]. Specific operations of the system (e.g., shifting stacks) can be simulated on such a domain model by applying the respective graph transformations that change the model accordingly (e.g., change the loads of stacks to simulate the shifting of items).

2.1 Automated Planning and Model-Driven Optimization

Automated planning [17] creates a `Plan` (cf. Fig. 1a—`plan metamodel`) which consists of an ordered list of `Actions`. In our setting, actions are represented as executions of graph transformation rules. They are executed to transfer a system from an initial state to a desired goal state (cf. Fig. 1b). For instance, a plan evolves the system from the state in Fig. 1c into a balanced state with equal item distribution between the available stacks by executing a sequential list of actions. These actions are chosen from the action types (represented as graph transformations rules in our setting) that are offered by the system which executes the plan (cf. Fig. 1b).

Such a plan can be found, e.g., by optimization techniques, to reach the goal state in the most efficient way. However, creating such an optimal plan usually requires to evaluate a large number of potential action sequences for which exhaustive approaches are infeasible. Domain-specific knowledge can facilitate efficient navigation in the solution space, although beneficial heuristics may not be available for the task at hand, or their development poses a challenging endeavour. Accordingly, meta-heuristics such as local search or Genetic Algorithms (GAs) [31] depict a problem-independent and therefore widely adopted alternative. The latter are based on natural selection within a population, i.e., the solution candidates, whose individuals are crossed and mutated while retaining only the fittest subset. In the case of our running example, each candidate resembles a plan consisting of a list of actions (i.e., individuals). The fitness of each plan is calculated based on a predefined fitness function that resembles the optimization goal (e.g., minimize standard deviation in the system, and minimize the number of actions of a plan to reduce its execution time). Presuming a suitable encoding, GAs are known to produce fit individuals fast. Their performance is influenced by the following parameters: (i) the population size, i.e., the number of individuals maintained over generations, (ii) the alteration probabilities concerning crossover and mutation, (iii) the selection operator, and (iv) the number of generations.

In MDO [22], the abstraction capabilities of Model-Driven Engineering [10] are leveraged to provide problem-agnostic frameworks for optimization tasks, e.g., [1, 8, 11]. In the context of automated planning, these frameworks use a model representation of the initial system state (cf. Fig. 1a for our running example), a list of action types represented as graph transformation rules, and produce a plan containing an ordered list of applications of these graph transformation rules (cf. Fig. 1b). This plan is produced by applying an optimization algorithm such as GA.

One MDO tool is MOMoT [8] which bridges the Eclipse Modeling Framework (EMF)[1] for domain modeling with the MOEA Framework[2] as a library for multi-objective search algorithms such as the Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [13], and Henshin [5] for specifying and executing graph transformations. MDO tools such as MOMoT are however intended for offline optimization. The created plans are simulated on the initial model using respective graph transformations to evaluate their effectiveness, but not on the physical systems that are represented by the model. This execution of the plan requires dedicated components, which decouples the execution from the actual planning. If a plan proves to not be beneficial any more during execution on the system level, there is no opportunity to perform online replanning as this would require runtime models which are derived directly from the systems [7]. DTs are intended to integrate the design-time and runtime phases of a system [30], thus they seem promising to solve this challenge of online replanning by reusing MDO tools.

2.2 Digital Twins

As mentioned before, DTs enable a bi-directional data flow between a physical system and its virtual representation [24], requiring different software components to enable this communication [33]. DT platforms [25] offer tool support for the creation of DTs,

[1] https://www.eclipse.org/modeling/emf
[2] http://moeaframework.org

and their connection to value-adding services that make use of the collected data, e.g., for simulation, visualization, or prediction. One aspect of these platforms is the modeling language used to represent the digital objects [32,38]. In particular, these languages consider design-time aspects (cf. Fig. 1a) and runtime aspects (cf. Fig. 1c), a view that is also supported, e.g., by [30].

Besides connecting the design-time model to the running system to get the runtime model representing the current system state, simulation [16, 19] also enables to create several alternative versions of the actual system (referred to as Experimentable Digital Twins (EDTs) [36]). In this work, EDTs can be used to provide an expected runtime model, simulating how the system would look like at a certain point in time when the created plan is executed as expected. However, even if the capabilities of such EDTs are used, the following challenges for achieving reactive planning are still to be tackled:

- **Challenge 1:** How to identify deviations between expected and actual runtime models? The different models need to be compared to identify unexpected changes in the running system which may trigger replanning.
- **Challenge 2:** How to react to unexpected changes in the running system? If such changes are detected, the DT should react to these deviations to achieve a high potential of the system.
- **Challenge 3:** How to avoid stopping the system while planning during runtime? Replanning should be performed efficiently, if possible in parallel to running the system, as stopping the system might be expensive, e.g., just-in-time production.

3 Reactive Planning Framework

In this section, we present an extended DT architecture and strategies for reactive planning which tackle the three challenges presented in the previous section.

3.1 Reactive Planning Architecture

We propose the reactive planning architecture for DTs as depicted in Fig. 2 which follows the general idea of MAPE-K [3,41]. In this architecture, the `Planner` calculates a `Plan Model` to achieve the goal specified in the `Goal Model` starting from the system state represented in the `Initial Model`, using the action types available in the system as functions. This part is the standard MDO processes as it is realized for instance in MOMoT.

The computed plan model is then sent to the `Digital Twin` (cf. Fig. 2), where the `Plan Execution Engine` extracts the ordered list of `Actions` from this plan model. For each Action a_i, the plan execution engine passes a_i to (i) the `Effector` to execute it on the actual system, and (ii) the `Simulator`, in our case Henshin, to calculate the `Expected Runtime Model` after a_i is executed on the system.

The expected runtime model is then passed to the `Conformance Checker` (cf. Fig. 2) together with the `Actual Runtime Model` that is collected from the `Monitor` after a_i is actually executed on the system. If the conformance checker cannot identify any deviation between the actual and expected runtime model, the plan execution engine continues to execute the next action in the plan (i.e., a_{i+1}) in the system and in the simulator. In the opposite case, i.e., a deviation is found, the `DecisionMaker`

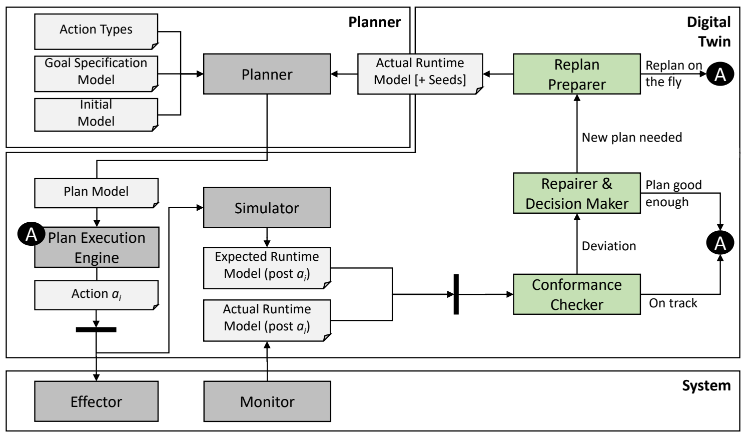


Fig. 2: Reactive planning architecture (components in green are newly introduced).

is invoked to judge the impact of the unforeseen change in the runtime model on the quality of the overall system. If this impact does not require a new plan, the old plan is simply repaired by deleting all actions that are not executable on the new runtime model any more (i.e., the precondition of the respective action type cannot be satisfied). If replanning is required, the `Replan Preparer` is invoked to perform the respective replanning using the planner component. The output of the planner component is the new plan that is then executed on the system by the plan execution engine.

3.2 Reactive Planning Strategies

If the repairer & decision maker component from Fig. 2 identifies that a new plan is necessary, two replanning strategies are provided by our framework (cf. Fig. 3):

- **On-the-fly Mode**: A predefined number of actions of the initial plan is still executed on the system in parallel to the replanning. Therefore, the `proceed` method of the plan execution engine (cf. Fig. 3) is called with the specified number of executed actions. Only after these actions are performed, the system is stopped (just in case that the replan process is not finished by then). The simulator is used to predict the expected runtime model after the execution of the steps, i.e., after execution of a_{a+i}, if `i` steps are executed in parallel to the planning. This runtime model can be injected into the planner as initial model for the replanning.
- **Idle Mode**: The execution of the old plan is stopped immediately, and the current actual runtime model (after a_i) is injected into the planner as the initial model.

In both strategies, the remaining actions of the old plan can be used as a seed for the replan by injecting them into the planner (cf. Fig. 3). This seed can be used as a starting point for the optimization algorithm as also proposed in previous studies [23, 34], e.g., as individual of the initial population of the GA, in contrast to using a full random starting point that does not take the previous plan into account. After this preparation of the replanning, the planner searches for a new plan, using the injected runtime model, the injected repaired plan in case that seeding is chosen, the initial goal specification model, as well as the action types available in the system. After this plan is created, it is executed on the system using the plan execution engine.

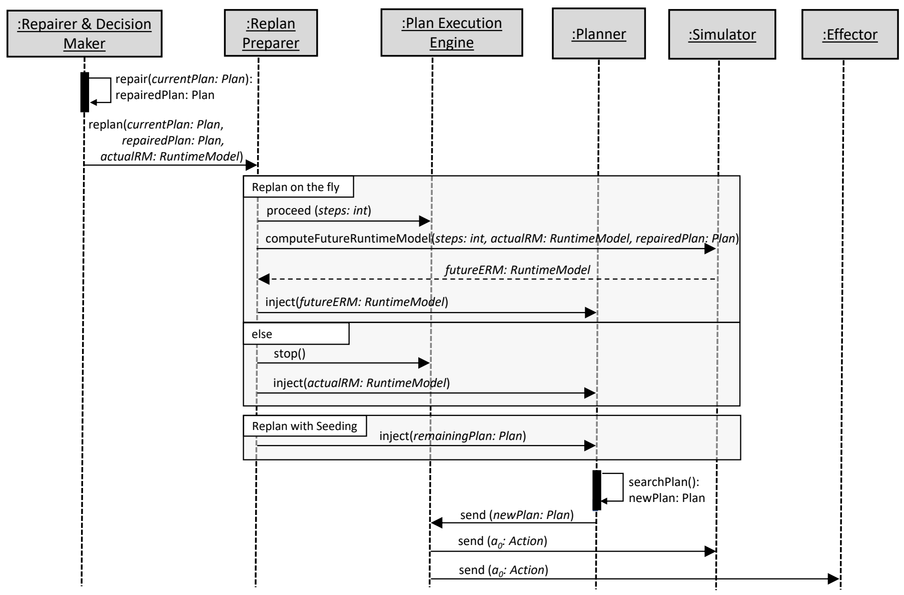

:Repairer & Decision Maker
:Replan Preparer
:Plan Execution Engine
:Planner
:Simulator
:Effector
repair(currentPlan: Plan): repairedPlan: Plan
replan(currentPlan: Plan, repairedPlan: Plan, actualRM: RuntimeModel)
Replan on the fly
proceed (steps: int)
computeFutureRuntimeModel(steps: int, actualRM: RuntimeModel, repairedPlan: Plan)
futureERM: RuntimeModel
inject(futureERM: RuntimeModel)
else
stop()
inject(actualRM: RuntimeModel)
Replan with Seeding
inject(remainingPlan: Plan)
searchPlan(): newPlan: Plan
send (newPlan: Plan)
send (a0: Action)
send (a0: Action)

Fig. 3: Replanning options in UML Sequence Diagram syntax.

3.3 Prototypical Implementation

To demonstrate our proposed architecture and its capabilities, we provide a prototypical implementation on Github [12]. In this prototype, the MOMoT framework [8] is used as planner. The plan execution engine uses the interpreter engine from Henshin as simulator for the execution of actions and as a stub of the system. To mimic unexpected changes that occur during the execution on the actual system, the monitor reports the expected result by using the Henshin interpreter to produce the next model version and randomly introduces changes in this resulting model. The components that are newly introduced in this paper are all implemented in Java.

3.4 Demonstration using the Stack Example

In the following, our reactive planning framework is demonstrated by using the running example introduced in Section 2. Therefore, we assume Fig. 1c to be the initial model based on which the initial plan (cf. Fig. 1b) is created. After execution of action a_0 however, unexpected changes in the system may happen, which require one of the replanning strategies under consideration.

One such change may be that stack 2 (`s2`) breaks down, leading to its exclusion from the system along with the items residing on it. Following this, intermediate neighbours are rewired to maintain the continuous circular connection intact. Possible causes could be malfunction, or intentional shutdown for maintenance or safety reasons. Once this deviation (`s2` is not available in the actual runtime any more) is identified by the conformance checker, the decision maker repairs the remaining plan (i.e., deletes actions a_1 and a_2, as they involve the removed `s2`), and judges whether a replan is necessary. As the standard deviation (2.2) is significantly higher than the value expected by the original plan (0), a replan is triggered. Since none of the planned actions remain from repair, execution is stopped, and a new plan is calculated.

In an alternative change, a new empty (`load` = 0) stack (`s6`) is added to the system, and connected to the respective neighbours depending on the inserted location (to the right of `s5`). This may occur during reintegration after temporary exclusion to perform maintenance, or to increase throughput in a production context. In this case, although not directly affected by this change, the planned actions are no longer optimal. Thus, replanning would be appropriate. In this case, besides the idle replanning option, also on-the-fly replanning is possible. In the on-the-fly mode, a_1 and a_2 are executed during replanning. The initial model for the parallel replanning is the expected runtime model resulting from these two actions.

4 Evaluation

4.1 Case Study Setup

To evaluate the effectiveness of the proposed reactive planning approach, we perform a case study [35]. The aim is to answer the following research questions:
RQ1: What is the potential of a replanning approach to mitigate runtime deviations compared to naive repair?
RQ2: To which extent does the quality/execution time of replanning solutions change if parts of the initial plan are used as a seed for the replanning algorithm?
RQ3: What is the difference with respect to execution time between calculating the new plan in parallel to running the system, or stopping the system during replanning?

Experimental Setting. We perform the case study on two different systems, i.e., instances, of the stack example. Both systems comprise 50 stacks initially, which are more sparsely and extensively loaded, respectively. More precisely, in the first system, each stack holds between 1 and 10 items with an overall load amount of 250, whereas in the second system, each stack holds between 1 and 100 items with a total of 2500 items. In both cases, loads are fairly unequal distributed with a standard deviation of 27.359 and 3.181, respectively. Plans are calculated using NSGA-II (MOMoT encoding) until an improvement of 50% is achieved with respect to our primary planning objective, i.e., decreasing the standard deviation. We assume actions take time to execute, hence shorter plans are of interest besides equal distribution. Therefore, we face a multi-objective planning task which also takes into account the plan size. We initialize NSGA-II with the population size set to 100 individuals, each reflecting a plan with up to 200 actions. Descendants of a generation are subject to one-point crossover, effectively exchanging parts of two "parent" plans at the same position, with a probability $p = 0.8$. For mutation within plans, three operators are used to remove ($p = 0.1$) or add ($p = 0.2$) actions, or vary ($p = 0.2$) the actions' shifting amounts. The value range for the shift amount is set to 5 and 50, respectively, for the lesser and more loaded instance. Execution of a plan is disturbed by changes introduced to the running system at partly random points in time. With this setting, we perform the following three experiments.

Experiment 1. For RQ1 and RQ2, we investigate the objective value reached with (i) a **naive repair** treatment, i.e., skipping the now unfeasible steps in our original plan, to (ii) a replanning treatment in which we execute the GA for 200 generations to plan with the disturbed runtime model. For this, we distinguish between a traditional **replanning** (denoted replan) setting in which the initial population consists of randomly synthesized plans, and a **replanning with seed** (denoted $\mathrm{replan}_{10\%}$) approach, in which 10% of the population is initialized with the remaining actions of the initial plan. In this regard, initial experiments showed that 10% was most beneficial for the seeding factor, as increasing the proportion further degraded the GAs performance. Excessive embedding of action sequences appears to lead to an overwhelming preference for them, putting the GA on a suboptimal path from the start and affecting population diversity.

Experiment 2. To continue with the results of Experiment 1 for achieving a 50% improvement in the target value, we perform another experiment in which we run replan against $\mathrm{replan}_{10\%}$. This time, however, the GA runs until a plan is found that matches the improvement threshold, rather than over a fixed number of generations.

Experiment 3. We leverage the DT's simulation capabilities for on-the-fly replanning, where the planner is requested to find a plan for the runtime model after projecting it to a future time step. This makes it possible to execute the original (repaired) plan during replanning to avoid stopping the system. We examine on-the-fly replanning for 1 up to 10 prediction steps and compare it to idle replanning, where no further execution on the system during replanning is performed. In all settings, the planner searches until a solution is found that satisfies the 50% improvement over the initial model.

All three experiments are performed under different conditions, varying in the type of perturbation and the time of its occurrence. Recall the two types of perturbation introduced in Section 3.4. Accordingly, five stacks are either added or removed from the environment. The locations are chosen randomly but evenly from parts of the stack configuration, i.e., the first stack is inserted/removed somewhere between the first ten consecutive stacks, the second between the next ten, and so on. In each case, it is ensured that the remaining stacks are connected in such a way that the circular connection remains intact. In terms of timing, the disruption occurs (i) in the first 10%, (ii) in the middle 10%, or (iii) in the last 10% of scheduled actions. That is, for a delegated plan with 200 actions, a deviation for the particular case occurs during one of actions 1-20, 90-110, or 180-200, in the latter case guaranteed before the scheduled plan is completed. Each experiment is run 30 times for each setting, except for Experiment 3. On occasion, perturbation in the last 10% of planned actions obstruct forecasting considering that simulation steps may exceed the remaining planned actions. Therefore, this setting is excluded from this experiment.

Evaluation Metrics. In Experiment 1, we measure the objective value of the system (i) after the deviation, (ii) after executing the naive repair approach, (iii) after replan, and (iv) after $\mathrm{replan}_{10\%}$. Also, to gain more insights into the GAs performance over time, we plot the lowest objective value resulting from the current best available plan in each generation for both replan and $\mathrm{replan}_{10\%}$. In Experiment 2, we measure the time required to achieve the desired 50% improvement as an indicator of the execution time of the replanning variants. Note that this is not achievable by merely removing infeasible actions, thus the naive repair approach is omitted here. In Experiment 3, the execution time is measured. It corresponds to the sum of (i) the time needed to find the new plan and (ii) the time needed to execute the tasks of the new plan (assuming an execution time of 10 seconds per task).

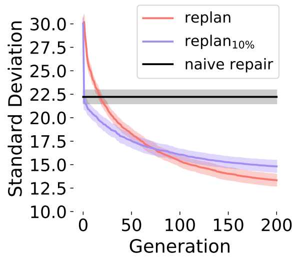
Error Occurence	Initial Plan Median Value	Naive Repair Median Value	Naive Repair Diff to Initial Plan	Replan Median Value	Replan Diff to Naive Repair	Replan with seed Median Value	Replan with seed Diff to no seed
First 10 %	13.602	22.176	63.0 %	13.583	-38.8 %	14.971	10.2 %
Middle 10 %	13.636	20.063	47.1 %	12.317	-38.6 %	14.544	18.1 %
Last 10 %	13.603	19.384	42.5 %	11.688	-39.7 %	17.905	53.1 %
First 10 %	13.650	17.068	25.0 %	11.737	-31.2 %	12.536	6.8 %
Middle 10 %	13.585	15.752	16.0 %	10.751	-31.8 %	13.038	21.3 %
Last 10 %	13.574	13.760	1.4 %	9.771	-29.0 %	13.108	34.2 %

Table 1: Experiment 1: 50 stacks with 1-100 items per stack. First three rows report median values for deviation of adding 5 stacks. Rows 4-6 report values for removing 5 stacks. Results are replicable for the system with 1-10 items per stack.

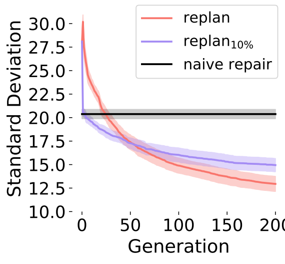
Regarding results, we report on the median values after recording 30 runs to deal with the stochastic nature concerning deviation settings and the GA. In addition, we conduct statistical tests [4] to identify whether our observations are significant. We perform a Mann-Whitney U test [29] with a significance level $\alpha = 0.01$, and opt for a two-sided test where appropriate. It is a non-parametric test that enables comparison of two random variables without the premise on having a normally distributed sample. If the p-value is less than or equal to α, the null hypothesis (H0) is rejected and we assume a true difference; if the p-value is greater than α, H0 is accepted. All data and scripts are available in the Github repository [12].

4.2 Case Study Results

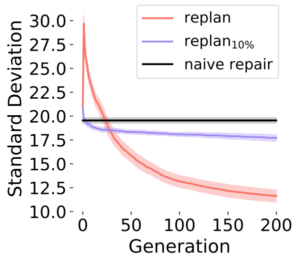
Experiment 1: A first result is the impact of the deviations (i.e., adding and removing stacks) on the standard deviation in the system. Conducting tests comparing the standard deviation in the system before and after deviation shows that adding five (empty) stacks has a significant impact on the objective value of the system ($p < .001$, in all settings), whereas removing five does not ($.641 < p < .739$, in all settings).
Table 1 reports the objective values expected from the initial plan prior deviation, and those resulting from plans after naive repair/replan/$\mathrm{replan}_{10\%}$ post deviation for previously described add/remove scenarios. Albeit removing stacks does not necessarily lead to a more or less balanced item allocation, in the first or middle 10% of planned execution, continuing with the repaired plan results in a much worse setting than executing the initial plan on the system would have, provided that no deviation has occurred ($p = 5.07e^{-10}$ and $p = 1.31e^{-08}$). Only when removing stacks after 90% plan execution, the difference between executing the repaired plan and the expectation from initial planning turns out insignificant ($p = 0.091$). For the removing case with 1-100 items per stack, results indicate that replan performs better than naive repair and $\mathrm{replan}_{10\%}$ with a deviation at the beginning (replan vs. naive repair, $p < 1.51e^{-11}$, and replan vs. $\mathrm{replan}_{10\%}$, $p = 3.178e^{-05}$), in the middle (replan vs. naive repair, $p < 1.509e^{-11}$, and replan vs. $\mathrm{replan}_{10\%}$, $p = 1.431e^{-05}$), and towards the end of execution (replan vs. naive repair, $p < 1.51e^{-11}$, and replan vs. $\mathrm{replan}_{10\%}$, $p = 1.51e^{-11}$). These results can also be replicated for the smaller model with 1-10 items per stack.



(a) After 0-10% execution



(b) After 45-55% execution



(c) After 90-100% execution

Fig. 4: Development of objective value over 200 generations for the add stack deviation.

Looking at the target evolution over several generations (cf. Fig. 4 for the add case), $\mathrm{replan}_{10\%}$ performs better than replan with its random initial population in the first generations. In fact, the random initialization performs so poorly initially that the plans produced are also inferior to the naive repair solution. After about 25 generations, however, replan generally outperforms the naive repair solution, and between 25 and 75 generations, it also outperforms the $\mathrm{replan}_{10\%}$. It can also be seen that the more advanced the execution of the plan, the less advantageous reseeding is and the less different the result is from a naive repair. The remove cases show a similar pattern, as do these cases in regard to the less loaded system.

Experiment 2: In Experiment 1, we found that the performance of replanning options varies with the number of generations developed by the GA. Clearly, more generations are required in general to find a new plan that satisfies higher target requirements. The progressions in Fig. 4 nevertheless indicate that choosing the right option depends on the desired target value. Therefore, in Experiment 2, we compare the generations required to achieve the initial goal value (i.e., 50% reduction of standard deviation in the system). Inspection of the median execution times for replan and $\mathrm{replan}_{10\%}$ reveals both to be a beneficial choice dependent on the application context: when introducing new stacks, the bias in the initial population leads to substantially longer execution times to find a plan conformant with the improvement requirement, compared to its unbiased counterpart ($\mathrm{replan}_{10\%}$ vs. replan, $1.505e^{-11} < p < .002$, in all cases). In contrast, after removing stacks from the environment, the search time is significantly reduced when considering the outdated plan with $\mathrm{replan}_{10\%}$, given a deviation at the beginning or towards the end ($p = 8.0e^{-06}$ and $p = 2.93e^{-04}$).

Experiment 3: In Fig. 5, the median over 30 runs is shown for the relative decrease in execution time when planning from states the system will enter during the next 10 actions. A trend towards lower overall execution times is observable in all cases although seemingly more substantial for later deviations w.r.t. completion of the initial plan. Hence the further we can anticipate the system situation,

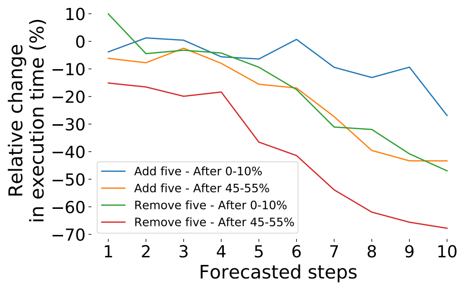


Fig. 5: Results of Experiment 3.

Case		**Forecasted steps**									
		1	2	3	4	5	6	7	8	9	10
Add	**0-10%**	.45	.754	.685	.404	.305	.679	.069	.015	.3	.001
	45-55%	.255	.163	.562	.228	.058	.015	$1.06e^{-04}$	$7.4e^{-07}$	$7.4e^{-07}$	$2.110e^{-09}$
Remove	**0-10%**	.929	.178	.573	.63	.26	.031	$1.340e^{-04}$	$2.804e^{-05}$	$1.127e^{-04}$	$7.733e^{-10}$
	45-55%	.007	.004	.001	.019	$1.0e^{-06}$	$1.1e^{-05}$	$1.505e^{-07}$	$3.56e^{-09}$	$2.039e^{-11}$	$4.533e^{-11}$

Table 2: Test statistic (p-value) comparing idle replanning vs. replanning on-the-fly. Results are reported for the system with 1-100 items per stack, but express the same tendency for the system with 1-10 items per stack. Significant results are highlighted.

the more efficient planning seems to become while plans still comply with the 50% improvement condition, but seemingly facilitate faster system operation at the same time. Table 2 contains the p-values to establish significance between on-the-fly planning for up to 10 upcoming plan steps, and disruptive planning where the system is idle. Following the trend of Fig. 5, planning from states further down the line leads to increasingly more efficient system operation. Evidently, parallel replanning outperforms idle replanning from seven anticipated steps onwards, in 3 of 4 cases, with exception of the system being augmented soon after execution start. Planning 10 steps in advance turns out superior in any case.

4.3 Discussion

Answering RQ1: As one would expect, accepting downtime to raise a new plan that is dedicated to the current state is superior to enforcing a partly obsolete and broken plan. This holds true for both augmentation and reduction in the system, and regardless of the deviation occurring sooner or later in the execution span. Nonetheless, late-occurring perturbations naturally carry a lower potential for degeneration, where few actions remain in the plan, which should be taken into account to consider replanning worthwhile. It is also worth noting that our case may not be affected by disruption to the extent that would be observed in other cases. Indeed, actions that do not involve added or removed stacks remain unaffected. Thus, simply omitting infeasible actions is more detrimental when the actions are causally interdependent, so it is expected that a plan with the naive repair approach becomes infeasible in many parts.

Answering RQ2: Results show that the prior plan in the GA population provides an advantage in replanning, while using randomly synthesized plans is more beneficial in the long run. Obviously, the remaining plan provides a good starting point, with further improvements observed over several generations. Therefore $\mathrm{replan}_{10\%}$ can provide good results quickly which is particularly useful for urgent scenarios. The noticeable difference in $\mathrm{replan}_{10\%}$'s performance when perturbations tend to occur later could be due to the reuse of fewer remaining actions, which still provide a small improvement and therefore spread quickly through the rest of the population, but also significantly limit further development. In contrast, the exclusive randomness present in the replan option allows for better solutions overall, with the advantage of $\mathrm{replan}_{10\%}$ disappearing over time. Moreover, results show that it makes sense to use one or the other option depending on the situation. If the goal is to reach a certain quality threshold, the seed option is preferable when stacks are disconnected from the system. In this case, $\mathrm{replan}_{10\%}$ provides such a solution much faster. However, if the quality of the plan is paramount or if

additional stacks are introduced, random initialization is recommended. In this context, the bias associated with the seed variant seems counter-intuitive when the updated runtime model indicates severe impairments of the system, but works well to counteract minor impairments of the system. Similar to naive repair, which is highly dependent on how functional the remaining plan part is, this likely also affects building on previous planning solutions, as is the case with $\mathrm{replan}_{10\%}$, and requires further investigation.
Answering RQ3: Results of Experiment 3 suggest that execution times can be reduced by anticipating future system states and coordinating the planning effort with the running system, assuming things go as planned. To this end, on-the-fly replanning outperformed the "stop and replanning" treatment, i.e., idle replanning, in 3/4 of the cases where execution of at least seven more steps was granted. It should be noted that the execution times are calculated assuming that an action takes 10 seconds. Remarkably, on-the-fly replanning is favored the longer the actions take, leaving more time for the planner to search for better quality and shorter plans, which in turn leads to faster execution overall. In addition, more time for planning also increases the likelihood of finding a plan that meets the requirements with fewer steps executed in parallel, which can reduce the risk of further disruptions in the meantime.

4.4 Threats to Validity

The presented study build on assumptions and decisions regarding their design, employed tools, and methods used that can affect its validity.
Internal Validity. As elaborated in Section 2.1, we use MOMoT as MDO tool for planning, thereby employing NSGA-II to derive the plan models on request of the DT. For plan evaluation we consider an objective dedicated to the task we investigated, minimizing the plan size. Reproducibility is threatened as the Henshin engine proposes actions during planning in a non-deterministic way which affects NSGA-II's initialization. Moreover, GAs are of stochastic nature and performance relies on several parameter settings whereby we adhere to commonly reported values. We counteract by repeating experiments for 30 times and conducting statistical tests to establish significance. Finally, the possibility of obtaining better results with a seeding factor other than 10% can not be ruled out and is left to further work.
External Validity. MOMoT uses Henshin to execute graph transformations on models and the MOEA framework to encode rule applications. Therefore action types have to be defined as graph transformation rules, and the runtime model of the DT has to be conformant with EMF. To demonstrate our DT integration we used a Stack Load Balancing case with two different configurations. Although suitable to clarify our contributions in this work, it is unclear whether our observations convey to other, possibly more complex domains and may be subject to more severe restrictions. For this reason, applicability needs to be shown also for other use cases, e.g., to envisage further deviation and action types. In this regard, assuming 10 seconds per action in Experiment 3 poses a strong limitation for the on-the-fly approach and is essential for our results. Moreover, we deal with several circumstances that may not be present elsewhere, including (i) enough feasible steps to enable forecasting post disruption, (ii) no system deviations during parallel replanning, and (iii) estimations for action execution times, all of which can suffer from variations and uncertainties in the environment, making adoption in other contexts challenging.

5 Related Work

Two threads of related work are discussed: (i) DT architectures and (ii) previous efforts to leverage prior planning information to tackle unforeseen changes.

DT Architectures. Different architectures have been proposed to exchange and handle data in DT systems in a unified, expandable way (e.g., see [28]). Whereas these architectures give a conceptual overview on how a DT can be used together with a CPS and value-adding services, there is also work that discusses specific key elements of a DT, e.g., Talkhestani et al. [37] mention synchronization with the physical asset, data acquisition, and simulation as key requirements and emphasize the importance of intelligence in DTs to enable autonomous CPS, e.g., to enrich them with predictive and learning capabilities, and also propose an architecture for AI-assisted decision making. An alternative to this AI-based solution is to employ case-based reasoning [9].

In comparison, our framework supports targeted adaptation in response to perturbation. Instead of acting on predefined patterns, the planner in our architecture is triggered by a separate checker component, e.g., when a critical quality threshold is reached. A new plan is then calculated, possibly reusing the outdated plan, to reach a desired system state. Furthermore, we use simulation to investigate the implications of planning in anticipation of future system states (i.e., on-the-fly replanning). This mitigates downtime and decreases execution time, and thus, may become a valuable property for DTs. Such a forecasting module has already been considered, e.g., for iterative plan refinement parallel to execution [40], but without considering uncertainty in plan execution.

Automated Planning and Adaptation. Initialization of GAs was investigated to improve adaptation in case-based reasoning [20], also using a memory to be leveraged on similar problems [27]. There is already work on considering reactions to runtime changes after an initial plan is calculated, e.g., using rule-based rewriting and local search [2], contemplating negative impacts for upcoming actions during execution to find a more robust alternative plan [6], an iterative procedure of plan simulation and adjustment upon detection of error-prone actions [26], and reasoning from past decisions and outcomes [39]. In [34], the authors even use a GA, similar to our work, and in order to strive for quick adaption to newly emerging tasks in a real-time mission replanning case. Therefore, the GA is initialized with previously computed task assignments (comparable to the seeding in our work). This improved initialization of a GA is also investigated by Kinneer et al. [23], but they face substantial evaluation overhead after embedding previous plans into the population. However, in our approach randomly generated plans demand more evaluation effort than solutions from prior planning as they exhaust the maximum solution length, hence we observe speedups after reseeding the obsolete plan, in contrast to the result from [23]. In general, their work focuses on handling uncertainty in a non-reactive manner whereas the planning layer in our framework couples disturbance detection with subsequent treatment. The MDO engine integrated into our architecture potentially also supports adaptation to unforeseen scenarios such as changing action types, goals, or system developments.

To sum up, our work is the first approach to use an MDO framework for replanning, also utilizing DT features to realize on-the-fly replanning. Seeding and initializing GA for replanning has been already used in the past, but our results provide interesting insights in case time critical replanning is required for CPS.

6 Conclusion and Future Work

In this paper, we have connected MDO with DTs and utilized the resulting overall framework for reactive planning. As the initial results are promising, the framework is also considered as a testbed for future experiments on reactive planning as all components are published as open source solutions.

For future work, we see the following lines of research. First, additional scenarios have to be explored to further validate the different replanning strategies and integration with running systems. Moreover, learning-based methods seem beneficial to be integrated in our framework to reach predictive planning. Finally, dedicated repair mechanisms may be a valuable extension as an alternative to replanning, e.g., if replanning is costly or only a few actions need to be repaired.

Acknowledgements. Work partially funded by the Austrian Science Fund (P 30525-N31) and by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development (CDG).

References

1. Abdeen, H., Varró, D., Sahraoui, H.A., Nagy, A.S., Debreceni, C., Hegedüs, Á., Horváth, Á.: Multi-objective optimization in rule-based design space exploration. In: ASE (2014)
2. Ambite, J.L., Knoblock, C.A.: Planning by rewriting. JAIR (2001)
3. Arcaini, P., Riccobene, E., Scandurra, P.: Modeling and analyzing MAPE-K feedback loops for self-adaptation. In: SEAMS (2015)
4. Arcuri, A., Briand, L.C.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: ICSE (2011)
5. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: MODELS (2010)
6. Beetz, M., McDermott, D.V.: Improving robot plans during their execution. In: AIPS (1994)
7. Bencomo, N., Götz, S., Song, H.: Models@run.time: a guided tour of the state of the art and research challenges. SoSyM (2019)
8. Bill, R., Fleck, M., Troya, J., Mayerhofer, T., Wimmer, M.: A local and global tour on MOMoT. SoSyM (2019)
9. Bolender, T., Bürvenich, G., Dalibor, M., Rumpe, B., Wortmann, A.: Self-adaptive manufacturing with digital twins. In: SEAMS (2021)
10. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, 2nd Edition. Morgan & Claypool Publishers (2017)
11. Burdusel, A., Zschaler, S.: Towards Scalable Search-Based Model Engineering with MDEOptimiser Scale. In: MODELS-C (2019)
12. CDL-MINT: ReactiveMOMoT (2022), https://github.com/cdl-mint/momot-reactive
13. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. TEC (2002)
14. Esterle, L., Porter, B., Woodcock, J.: Verification and uncertainties in self-integrating system. In: ACSOS-C (2021)
15. Feng, H., Gomes, C., Thule, C., Lausdahl, K., Iosifidis, A., Larsen, P.G.: Introduction to digital twin engineering. In: ANNSIM (2021)
16. Fitzgerald, J.S., Larsen, P.G., Pierce, K.G.: Multi-modelling and co-simulation in the engineering of cyber-physical systems: Towards the digital twin. In: From Software Engineering to Formal Methods and Tools, and Back (2019)

17. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: theory and practice. Elsevier (2004)
18. Gil, R.: Automated planning for self-adaptive systems. In: ICSE (2015)
19. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a survey. CSUR (2018)
20. Grech, A., Main, J.: Case-base injection schemes to case adaptation using genetic algorithms. In: ECCBR (2004)
21. Heckel, R., Taentzer, G.: Graph Transformation for Software Engineers. Springer (2020)
22. John, S., Burdusel, A., Bill, R., Strüber, D., Taentzer, G., Zschaler, S., Wimmer, M.: Searching for optimal models: Comparing two encoding approaches. JOT (2019)
23. Kinneer, C., Garlan, D., Goues, C.L.: Information reuse and stochastic search: Managing uncertainty in self* systems. TAAS (2021)
24. Kritzinger, W., Karner, M., Traar, G., Henjes, J., Sihn, W.: Digital twin in manufacturing: A categorical literature review and classification. IFAC (2018)
25. Lehner, D., Pfeiffer, J., Tinsel, E., Strljic, M.M., Sint, S., Vierhauser, M., Wortmann, A., Wimmer, M.: Digital twin platforms: Requirements, capabilities, and future prospects. IEEE Softw. (2022)
26. Lesh, N., Martin, N.G., Allen, J.F.: Improving big plans. In: AAAI/IAAI (1998)
27. Louis, S.J., McDonnell, J.R.: Learning with case-injected genetic algorithms. TEC (2004)
28. Malakuti, S., Schmitt, J., Platenius-Mohr, M., Grüner, S., Gitzel, R., Bihani, P.: A four-layer architecture pattern for constructing and managing digital twins. In: ECSA (2019)
29. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Ann Stat **18**(1) (1947)
30. Margaria, T., Schieweck, A.: Towards engineering digital twins by active behaviour mining. In: Model Checking, Synthesis, and Learning. Springer (2021)
31. Mitchell, M.: An introduction to genetic algorithms. MIT Press (1998)
32. Pfeiffer, J., Lehner, D., Wortmann, A., Wimmer, M.: Modeling capabilities of digital twin platforms - old wine in new bottles? In: ECMFA (2022)
33. Qi, Q., Tao, F., Hu, T., Anwer, N., Liu, A., Wei, Y., Wang, L., Nee, A.: Enabling technologies and tools for digital twin. J. Manuf. Syst. (2021)
34. Ramirez-Atencia, C., Bello-Orgaz, G., R-Moreno, M.D., Camacho, D.: MOGAMR: A Multi-Objective Genetic Algorithm for real-time Mission Replanning. In: SSCI (2016)
35. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. EMSE (2009)
36. Schluse, M., Priggemeyer, M., Atorf, L., Rossmann, J.: Experimentable Digital Twins - Streamlining Simulation-Based Systems Engineering for Industry 4.0. TII (2018)
37. Talkhestani, B.A., Jung, T., Lindemann, B., Sahlab, N., Jazdi, N., Schloegl, W., Weyrich, M.: An architecture of an intelligent digital twin in a cyber-physical production system. at-Automatisierungstechnik (2019)
38. Tao, F., Zhang, H., Liu, A., Nee, A.Y.C.: Digital twin in industry: State-of-the-art. TII (2019)
39. Veloso, M.M.: Flexible strategy learning: Analogical replay of problem solving episodes. In: AAAI (1994)
40. Wally, B., Vyskocil, J., Novák, P., Huemer, C., Sindelár, R., Kadera, P., Mazak-Huemer, A., Wimmer, M.: Leveraging iterative plan refinement for reactive smart manufacturing systems. TASE (2021)
41. Weyns, D., Iftikhar, M.U., de la Iglesia, D.G., Ahmad, T.: A survey of formal methods in self-adaptive systems. In: C3S2E (2012)