2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) | 978-1-6654-0289-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/SEAMS51251.2021.00029

Self-Adaptive Manufacturing with Digital Twins

Tim Bolender*, Gereon Bürvenich*, Manuela Dalibor*, Bernhard Rumpe*, Andreas Wortmann*†

* Software Engineering, RWTH Aachen University, Aachen, Germany, www.se-rwth.de
† Institute for Control Engineering of Machine Tools and Manufacturing Units
University of Stuttgart, Stuttgart, Germany, www.isw.uni-stuttgart.de

***Abstract*—Digital Twins are part of the vision of Industry 4.0 to represent, control, predict, and optimize the behavior of Cyber-Physical Production Systems (CPPSs). These CPPSs are long-living complex systems deployed to and configured for diverse environments. Due to specific deployment, configuration, wear and tear, or other environmental effects, their behavior might diverge from the intended behavior over time. Properly adapting the configuration of CPPSs then relies on the expertise of human operators. Digital Twins (DTs) that reify this expertise and learn from it to address unforeseen challenges can significantly facilitate self-adaptive manufacturing where experience is very specific and, hence, insufficient to employ deep learning techniques. We leverage the explicit modeling of domain expertise through case-based reasoning to improve the capabilities of Digital Twins for adapting to such situations. To this effect, we present a modeling framework for self-adaptive manufacturing that supports modeling domain-specific cases, describing rules for case similarity and case-based reasoning within a modular Digital Twin. Automatically configuring Digital Twins based on explicitly modeled domain expertise can improve manufacturing times, reduce wastage, and, ultimately, contribute to better sustainable manufacturing.**

***Index Terms*—Self-Adaptive Manufacturing, Digital Twins, Case-Based Reasoning, Domain-Specific Languages**

I. Introduction

Industry 4.0, the fourth industrial revolution, focuses on integrating Cyber-Physical Production Systems (CPPSs), their processes, and stakeholders to optimize the complete value-added chain to ultimately save time, cost, and reduce resource consumption [1]. These CPPSs are long-living complex systems deployed to and configured for diverse environments. Due to specific deployment, configuration, wear and tear, or other environmental effects their behavior as-operated can diverge from its behavior as-designed over time. Successfully using the CPPSs demands the expertise of human operators to mitigate these effects. In such cases, experienced operators employ significant manual efforts to configure the CPPSs before starting production. Making their expertise machine-processable can facilitate their self-adaptive operations.

One vision for implementing Industry 4.0 are so-called Digital Twins (DTs) [2], which are digital duplicates of CPPS that represent, control, and predict the behavior of their physical counterparts. Where DTs control CPPSs, they need to have knowledge about the CPPS and its operations. Consequently, they are connected to the CPPS and lend themselves for automatically adapting it to changing challenges. For their adaptation of the overall manufacturing system consisting of DT and CPPS, the DTs need to be enabled to sense changes in the CPPS's behavior, reason over reified domain expertise, and make changes to the CPPS accordingly.

Case-Based Reasoning (CBR) [3], [4] is an AI planning technique in which highly specific, and hence, sparse domain expertise is reified in cases. These describe undesired situations a system should react to together with suitable reactions to capture the expertise of CPPS operators. Based on this domain expertise, the DT of a CPPS can detect undesired situations, find matching or similar cases, and adjust the CPPS according to their reactions to produce a desired system state again.

This enables integrating highly domain-specific expertise (that can hardly be foreseen by the CPPSs' developers) in brownfield settings where the long-living CPPSs are already in place as well as in greenfield settings, where the CPPS and its DT are developed together.

To leverage CBR over domain expertise into self-adaptive manufacturing, we devised a modeling framework comprising multiple interrelated modeling languages and integrate it into our model-driven architecture for DTs [5], [6], [7]. The contributions of this paper are

- extensible modeling languages to capture domain expertise in the form of cases and to describe the similarity between them, and
- a modular architecture for integrating CBR into DTs and supporting all activities related to identifying, applying, and learning cases.

In the following, Sec. II illustrates the challenges of incorporating domain expertise into manufacturing on the example of injection molding. Sec. III then introduces preliminaries. Afterwards, Sec. IV presents our CBR modeling languages and Sec. V presents our realization of CBR within DTs. Sec. VI illustrates our method's application to the configuration of an injection molding process. Sec. VII discusses observations and Sec. VIII highlights related research. Sec. IX concludes.

II. Context

Injection molding is a popular form of batch processing for the mass production of 3D plastic parts that is performed daily billionfold around the world to manufacture identical parts repeatedly in high quality. Injection molding itself is

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2023 Internet of Production - 390621612.

2157-2321/21/$31.00 ©2021 IEEE
DOI 10.1109/SEAMS51251.2021.00029

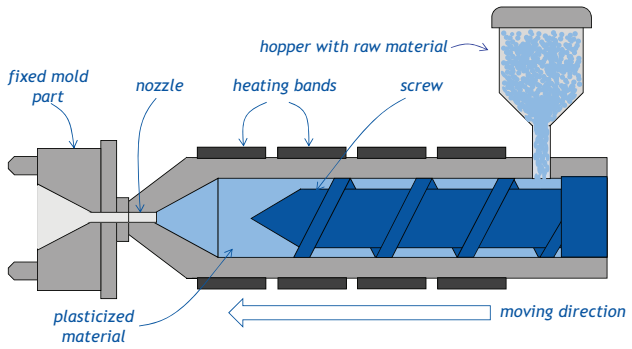

Figure 1. Detailed view of the plasticizing unit of an injection molding machine.

highly automated but requires a configuration that often has to be determined and adjusted manually due to changing CPPS properties, materials, or environmental characteristics.

Figure 1 illustrates the typical components of an injection molding machine. It consists of a hopper to insert the material into the plasticizing unit. The plasticizing unit heats the material until it melts and carries it to the front through a screw. Heating bands support the melting. Next, the material is injected into the mold, which is the 3D negative of the part to be produced. The material solidifies inside the mold and forms the desired shape.

Overall, the production process consists of four phases: (1) Dosing: The material is fed into the cylinder. Rotation of the plasticizing unit screw conveys the material to the nozzle. Through the heating and the movement friction, the material plasticizes. (2) Injecting: The screw moves towards the nozzle and injects the material into the mold. The movement speed and the viscosity of the material determine the speed of the injection. Parameters of this phase are temperature, volume, time, speed, and mold characteristics. (3) Holding: The screw slows down, and the clamping unit applies pressure to the mold. Thereby, the material fills the last parts of the mold. The characterizing parameters during the holding phase are the time and pressure. (4) Cooling: Before ejecting the finished part, the material has to solidify by cooling off. The correct time and temperature of the mold prevent defects such as warpage.

The great variety of environmental and CPPS influences, as well as of process parameters and their impact on the process and part quality complicates finding optimal CPPS configurations. Deviations from the predictions of simulations are common, especially due to wear and tear. Consequently, only experienced operators can configure the CPPS properly by applying domain expertise learned during their career. DTs can help to overcome these difficulties by reifying the operators' domain expertise and automatically controlling the CPPS. We, therefore, identify the following four requirements for incorporating domain expertise via DTs in batch processing:

R1 **Comprehensibility**: Operator expertise must be reified in means that are comprehensible by domain experts and support non-clear cohesion as well as the integration of empirical knowledge.

R2 **Automatability**: The DT has to monitor the situation of the underlying CPPS permanently and, when encountering undesired states, has to automatically adapt the CPPS without further interaction based on the reified domain expertise.

R3 **Adjustability**: The DT and its domain knowledge have to be adjustable to different contexts, deployments, configurations without in-depth software engineering expertise.

R4 **Self-Explainability**: Extracting knowledge is difficult in a domain with unclear coherence, yet the CPPSs' decisions should be comprehensible by domain experts. To support self-explainability, the DT needs to support the creation of empirical-based analytical knowledge.

We conceived extensible modeling languages to fulfill R1 and R2, a modular DT architecture supporting R3, and a case synthesis realizing R4. In the following, we present these modeling languages, an implementation of our DT architecture for CBR, and a system for reifying, applying, and producing domain expertise through CBR.

III. Preliminaries

In our approach to self-adaptive manufacturing, we employ CBR, AI action planning, and software language engineering. This section introduces these preliminaries.

Case-Based Reasoning and Planning

A DT that controls a CPPS encounters situations that are not anticipated during specification and thus should adapt to new conditions autonomously. CBR [3] is a problem-solving paradigm that utilizes knowledge about previously encountered situations and reuses their solutions. Consequently, a case consists of a situation description (a condition over available data sources), its solution, and additional information about how the solution was derived. The CBR cycle is divided into four phases: (1) Retrieve the case most similar to the current situation. (2) Reuse the solution of the most similar case. (3) Revise that solution if the case differs too much from the current situation. (4) Retain the revised case in the knowledge base. Hence, an essential part of the CBR cycle is the identification of similar cases. If the case's condition can reference multiple heterogeneous attributes, a generally useful similarity cannot be specified; instead, this consideration is highly domain-specific. To support engineers and domain experts in specifying similarity measures, these often are broken down according to the *local-global-principle*: A distinct local measure defines the similarity for each individual attribute referenced in a case condition. A global similarity measure then enables computing the similarity for the whole case by using, *e.g.,* the weighted average of all the local similarities.

CBR, hence, is limited to applying existing cases and learning deviations of cases. It cannot, generally, produce new solutions to completely unforeseen challenges. General automated planning and scheduling supports creating new solutions (plans) to unforeseen challenges, if the necessary

primitives (types, actions) are provided. In our architecture, we leverage AI planning based on the Planning Domain Definition Language (PDDL) [8] as a fallback mechanism when CBR fails. PDDL is a language for representation and exchange of planning domain models comprising types, constants, and actions with preconditions and postconditions. A PDDL problem description is an instantiation of model elements and formulates a goal that describes which situation the DT shall achieve. A planning system, such as MetricFF [9] processes domain models and problem descriptions and derives a sequence of actions, a plan, that leads from the initial situation to the goal [10]. Thus, PDDL can be employed as a fallback if CBR cannot find a similar case to address undesired situations [11].

MontiCore Language Workbench

Our method presented in the following relies on modeling languages [12] to describe cases, case similarities, DT architecture components, and its ties to CPPS, model transformations, and code generation. All of these exist in the technological space of the MontiCore [13] language workbench [14] for the efficient engineering of modular, textual modeling languages. These modeling languages comprise context-free grammars, Java-based well-formedness rules, model-to-model transformations, and FreeMarker-based code generators [13]. From grammars, MontiCore derives an extensible infrastructure to parse, check, and transform models of the languages defined by the grammar. MontiCore comes with a multitude of reusable modeling language modules ranging from expressions and statements of various complexities, to UML fragments, software architectures, and more.

A particular kind of languages available in the technological space of MontiCore are domain-specific tagging languages [15] that support extending models of a base language with additional information without polluting these. To this effect, their infrastructure (grammar, well-formedness rules) is derived from a base language to enable enriching models of that language with information, *e.g.,* about platform-specific details of their use, without polluting them. As the tag model is separate from the base model, models of the base language are unaware of being tagged and can thus be reused in different contexts.

Digital Twin Architecture

MontiArc [16], [17] is an architecture description language for specifying reusable components within a software architecture and their connections through typed, directed ports. MontiArc comes with a Java code generator that generates Java classes conforming to the specified components, methods to access port values, and a mechanism to inject handwritten behavior specifications. In previous work, we built a DT with MontiArc, that enables automatic experiment execution on injection molding machines [5] and also provides a cockpit for visualizing the current state of the machine [18]. We define a Digital Twin (DT) of a system as a set of models of the system, a set of contextual data traces, and a set of services to use the data and models purposefully with respect to the original system [5]. In our notion, a DT is a software system representing a physical counterpart and encapsulating domain knowledge in the form of models that characterizes this physical counterpart. Further, it contains data about the physical counterpart and services to collect more data or interact with this counterpart. We built a DT (*cf.* Figure 2) with MontiArc that provides the following components:

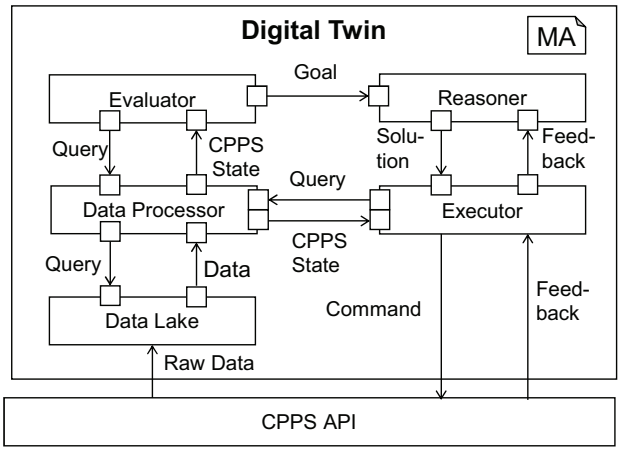

Figure 2. Reusable Digital Twin Architecture modeled with MontiArc.

- Data Lake: Encapsulates multiple databases that store data about the physical system, its context, and data produced by the DT
- Data Processor: Accesses the Data Lake and collects relevant data for the DT
- Evaluator: Supervises the physical system's state and triggers the reasoner if a malfunctioning is detected
- Reasoner: Based on data about the physical system and models describing its intended behavior, finds a solution to return to the intended state
- Executor: Accesses the physical system via OPC UA [19] and ensures that the solution provided by the Reasoner is executed on it.

While previous reasoner implementations offered a way to organize experiments, we will exchange this Reasoner with a new reasoner that performs CBR.

IV. Modeling Languages for Case-Based Reasoning

We present a modeling framework that supports the cycle of CBR and supports the creation, storage, retrieval, and comparison of cases via the case base. Since the DT architecture and connectivity to the CPPS are provided by the DT framework, domain experts only need to provide the essential domain knowledge, defining known experiences as cases and specifying case similarity measures to create a new DT for a CPPS. We utilize UML/P class diagrams (CDs) [20] and introduce further modeling languages to support the description of the domain knowledge. The integration of these models

supplements the framework configuration and incorporates the domain knowledge into the workflow. Additionally, the framework enables defining PDDL-based fallback strategies for circumstances where CBR fails to produce a suitable case. These fallback solutions are also modeled by domain experts and thus explicitly tailored to the underlying CPPS.

A. CBR Modeling Languages

Modeling languages facilitate the specification of the CBR framework and assist domain experts in making their expertise machine-processable. Their models tailor the steps of the CBR cycle and the case base to a specific application domain. Figure 3 gives an overview of the integrated modeling languages employed in our approach: (1) Class diagram models describe the elements and relations of the domain and specify data structures available to the framework. (2) Case base models describe acquainted cases of the physical system. The framework interprets and synthesizes case models at runtime. (3) Case similarity models specify how to compute the similarity between cases based on their attributes. (4) Models of the MontiArc architecture description language define the components and architecture of the DT implementing the CBR loop. These are predefined and provided with the DT framework. (5) OPC UA tagging models [5] define how the DT architecture connects to the API of the CPPS.

The case base language foresees extension with domain-specific expressions and actions using the language extension mechanisms of MontiCore [13]. The code generated from the case similarity models supports integrating handcrafted code to define more complex similarity analyses.

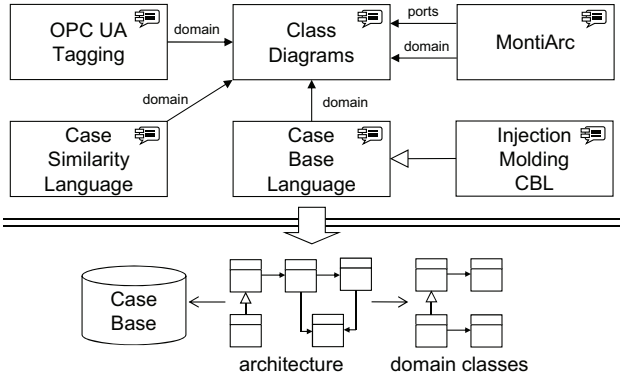

Figure 3. Modeling languages, relations, and artifacts specifying the domain, cases, and fallback option for an application of the CBR framework.

Class Diagrams

CDs describe the elements and relations of the application domain. The case-based reasoner utilizes the corresponding data structures to build and compare cases. Figure 4 presents a textual UML/P CD that illustrates an excerpt of the domain of injection molding. Class `ProcessData` symbolizes a data record in the molding process. Besides metadata like the `cycleId` and `cycleTime` (l. 3), it provides the values of the nozzle temperature and the injection pressure (l. 4).

```
01 classdiagram InjectionMolding {                    CD
02   class ProcessData {
03     int cycleId, cycleTime;
04     double nozzleTemperature, injectionPressure;
05     boolean heating;
06   }
07 }
```

Figure 4. Class diagram for an excerpt from the domain of injection molding, containing a small set of parameters of the injection process.

Case Base Language

The case base language supports domain experts in defining cases. To this end, they distinguish between *known* and *unknown* cases. Known cases describe undesired situations for which the domain expert knows a solution with its expected consequences. If a similar situation occurs, one or multiple solution steps can be repeated to solve the problem. Unknown cases describe undesired situations for which the expert does know that the situation might occur and that the system configuration has to adapt, but does not know how to adapt it. Although domain experts might not be able to provide precise instructions, they can provide helpful context knowledge to sort out the problem through a fallback system. By combining both types of cases, the domain expert can describe the whole space of situations that can occur and should be handled by a CBR system.

The Case-Base Language (CBL) is defined as textual MontiCore modeling language (*cf.* Figure 5). Every case definition references the domain CD by importing models (l. 2) using the non-terminal `ImportStatement` provided by inheriting from MontiCore's `MCExpressions` language for binary expressions, statements, and types. Each case base can contain multiple cases (l. 2) and each case consists of a head, a body, and an optional fallback (l. 3). Its head denotes the state of the case and specifies its name (l. 4). The body comprises a condition and an optional solution (l. 5). If the body features a `Solution` part, the case is known. Otherwise, it is treated as an unknown case. The condition essentially is a Boolean expression (l. 6) over any types and properties available through the domain model. Well-formedness rules of the CBL ensure that the expressions are valid Boolean expressions (*i.e.,* referenced types exist and can be compared as specified by the expression). The solution is a non-empty sequence of solution parts with a consequence (l. 7). Each solution part (l. 8) refers to the interface non-terminal `SolutionExpression` (l. 12) that is an extension point of this grammar and can be implemented in domain-specific sub-grammars. Per default, arbitrary assignments are supported as solutions (ll. 15-16) and corresponding well-formedness checks are provided. Further, domain experts can also specify java code that performs a solution and call this code in the solution part. Java calls are realized by the imported `MCExpressions`. The CBL also supports PDDL specifications (provided by importing `PDDL`) for planning if the solution for a case is unknown. The consequence describes a postcondition that should hold after the case has been applied. The CBR system relies on

the specified `yields` consequence to check whether a case was successfully applied. If the specified postcondition is not fulfilled, the DT applies this case less likely in the future. Similarly, the fallback (l. 10) is an extension point for fallback actions to be used if the case fails. Per default, notifying users (ll. 17-18) and falling back to PDDL planning (ll. 19-20) are supported. Both solutions and fallbacks are meant for extension through domain-specific or application-specific sub-grammars in which, *e.g.,* temporal expressions, fallback automata, or other means can be integrated using MontiCore's language extension mechanisms [13].

```
01 grammar CaseBaseLanguage extends MCExpressions, PDDL {          MC
02   CaseBase     = ImportStatement* Case+;
03   Case         = Head "{" Body "}" Fallback?;
04   Head         = "case" Name;
05   Body         = Condition Solution?;
06   Condition    = "if" Expression;
07   Solution     = SolutionPart+ Consequence;
08   SolutionPart = "do" SolutionExpression;
09   Consequence  = "yields" Expression;
10   Fallback     = "fallback" FallbackExpression;
11
12   interface SolutionExpression;        <- interface productions
13   interface FallbackExpression;           for domain-specific
14                                           extension of the CBL
15   AssignmentSolution implements SolutionExpression
16                = AssignmentExpression;   <- from
17   NotifyFallback implements FallbackExpression     MCExpressions
18                = "notify" message:String;
19   PDDLFallback implements FallbackExpression
20                = "goal" PDDLGoal;        <- from PDDL
21 }
```

Figure 5. Excerpt of the MontiCore grammar of the case base language.

Figure 6 illustrates a model of the CBL that refers to the injection molding domain model `ProcessData` (l. 1) and shows two cases that may occur in the injection molding machine. The first case represents a known case (ll. 3-7) that handles a problematic temperature of the injection nozzle characterized by being higher than 500 degrees Celsius (l. 4). The attributes in this expression reference the domain model of Figure 4. As a solution, the case contains an assignment expression that specifies setting the `heating` to level `1`. The second case is unknown (ll. 9-12) and addresses dangerous pressure in the injection process (l. 10). When the condition holds, a retrieval of similar known cases is triggered. In case the search yields no cases, the DT uses the PDDL fallback expression to start finding a plan over the CPPS actions and properties that, when executed, will reduce the pressure.

Case Similarity Language

The second essential part of a case-based reasoning system is its ability to assess similarity between a situation in the CPPS and a case. Similarity as a metric is expressed as a positive rational number with `0` being considered equal. The Case Similarity Language (CSL) supports describing weighted global and local similarity based on a the types of domain models and promotes integration of further, handcrafted, similarity analyses using the top mechanism [13] with its generated code artifacts. We developed a Domain-Specific Language (DSL) for specifying similarities between cases (Figure 7). Every case

```
01 import InjectionMolding.ProcessData;                            CBL
02
03 case Overheating {
04   if ProcessData.nozzleTemperature > 500
05   do ProcessData.heating = 1          <- assignment as solution to situation
06   yields ProcessData.nozzleTemperature < 500
07 }
08                                       <- triggers search for known cases
09 case DangerousPressure {
10   if ProcessData.injectionPressure > 20
11   fallback goal (injectionPressure 10)  <- PDDL fallback if no similar case is found
12 }
```

Figure 6. Excerpt of a case base for injection molding regarding dangerous temperatures and pressures.

similarity definition is based on domain models and consists of global and local similarity metrics (ll. 2-5). After an import list establishing relation to the domain of discourse (l. 2), a name case similarity specification follows (ll. 3-5), which contains (l. 4) multiple local similarity metrics (l. 7) that relate to individual attributes of the domain models and a single global similarity metric (l. 8) describing how these individual similarities are weighted. Both kinds of metrics reference to interface non-terminals (ll. 10-11) that facilitate introducing new metrics into the CSL. The CSL features two kinds of metrics for local and global similarities (ll. 13-17) out of which the `manual` metric specifies that a handcrafted similarity analysis should be used. This demands implementing a specific Java interface of the CSL's runtime system, which is then invoked if the `manual` is used. Well-formedness rules ensure that the referenced domain types exist and are correctly used.

```
01 grammar CaseSimilarityLanguage extends MCExpression {           MC
02   CSD = ImportStatement*
03         "case" "similarity" Name "{"
04           LocalMetric* GlobalMetric
05         "}";
06                                       <- extension points for additional metrics
07   LocalMetric  = "local"  Name Local ";";
08   GlobalMetric = "global"      Global ";";
09
10   interface Local;
11   interface Global;
12                                       <- specifies use of a handcrafted metric
13   Manual   implements Local, Global = "manual";
14   Absolute implements Local  = "absolute";
15   Weighted implements Global = "weighted" "{" Weights "}";
16   Weights                    = (AttributeWeight || ",")+;
17   AttributeWeight            = Name "=" weight:Double;
18 }
```

Figure 7. Excerpt of the MontiCore grammar of the CSL.

Figure 8 displays a model of the CSL. It refers to the injection molding domain model `ProcessData` (l. 1) and specifies two local similarities for this domain (ll. 4-5). The model specifies the absolute distance local similarity metric for the `nozzleTemperature` and a handcrafted metric for the `pressure` attribute. Global similarity then is defined through the weighted combination of `nozzleTemperature` and `pressure` (ll. 7-10).

Similar to the CBL, the CSL also does not aim to be a catch-all language but supports leveraging MontiCores language ex-

```
01 import InjectionMolding.ProcessData;                          CSL
02
03 case similarity InjectionMolding {
04   local ProcessData.injectionPressure manual;
05   local ProcessData.nozzleTemperature absolute;
06                                  weighted sum of
07   global weighted {              local similarities   generate factory
08     ProcessData.injectionPressure = 0.7,              for handcrafted
09     ProcessData.nozzleTemperature = 0.3               solution
10   };
11 }
```

Figure 8. Exemplary case similarity model.

tension mechanisms to define highly specific similarity metrics (*e.g.,* featuring uncertainty, SI units, or domain terminology).

PDDL Fallback Modeling

The selection of cases to resolve a situation bases on their similarity. Depending on the size and scope of the case base, no case might be available. As this is not unusual, the DT has to be able to handle such situations. Therefore, we provide a fallback possibility to full AI planning.

Due to the dependence on the domain, we do not prescribe a specific modeling language for fallback activities but provide extension points in both CBL and DT. The default implementation of these extension points are notification of human operators and invoking a PDDL planning. Depending on the case defined by the domain expert, parameters for a `Fallback` component or PDDL goals can be defined. For the former, a DT component taking care of the fallback activities has to be provided, for the latter, the corresponding goal must be defined in the DT's PDDL knowledge base.

B. Integrating Domain-Specific Models into CBR

In our approach, the CSL models are interpreted at DT runtime, whereas the CSL models are used for code generation at design-time to enable the integration of handcrafted, more complex, similarity analyses (*cf.* Figure 9). This section explains their overall integration.

At Design-Time

The similarity measures do not change once the DT is running. Hence, at design-time, CSL models are translated into Java artifacts that are invoked when their computations are necessary. As the models feature planned extension with handcrafted Java computations (indicated by the `manual` keyword), we exploit the code generation to support injection of these handcrafted similarity analysis artifacts using the TOP mechanism [13], a variant of the generation gap pattern, and generate factories to inject implementations of known similarity analysis interfaces into the similarity computations. Leveraging Java for more complex analysis liberates domain experts specifying similarities from the complexities of an overly generic (possibly Turing-complete) modeling language and supports engineers in using established tools, frameworks, and libraries to develop the analyses. Where more complex analyses are required within in the CBL, MontiCore's language extension enables creating sub-languages whose grammars implement and extend the CBL's extension points.

Each model for local similarity is translated to a class with a name of the form `<Name>LocalSimilarity`. `<Name>` is replaced with the name of the domain model attribute. The class provides a single public method to calculate the local similarity. As a parameter, the respective attribute and the condition expression of the case to compare to are given. The domain model determines the type of the attribute. Therefore, type-safe artifacts can be produced, and their stable interfaces hide whether these are generated or handcrafted from the framework.

The model of global similarity results in a class with a name of the form `<Name>Similarity`, where `<Name>` is the name of the overall similarity model (Figure 8, l. 1). The class provides a single public method to calculate the global similarity. All domain attributes and all condition expressions are passed as parameters. Based on the defined similarity type, an implementation is generated. This holds for the global weighted similarity. The CBR modeling framework collects the artifacts for the local similarities based on the models. Based on the weights, their calculated similarity is summarized.

Ultimately, this enables domain experts to develop their own DTs and enrich these with domain knowledge without requiring any programming skills.

At Runtime

The case base comprises cases, which are interpreted at runtime. To this end, they are parsed, and their abstract syntax representation is stored in memory. During runtime, the DT monitors the CPPS and checks for undesired situations using the case models. If such a case is found, DT action is required, and it tries to retrieve the most similar case.

During the *retrieve* phase of the CBR cycle, the DT receives the undesired (current) situation and the list of known cases as input. To determine similar cases, the similarity is calculated for every case. The metric value is determined based on the conditions and the situation. This step relies on the similarity computation artifacts generated based on the CSL models and the related handcrafted artifacts. Next, the results are filtered by a predefined constant threshold. Similarities between cases range between 0 and 1 in our implementation, and we consider a value smaller than 0.2 as similar enough to try to apply the solution of a case.

In the reuse phase, the DT then determines the actual solution to execute. For this, the previously selected set of similar cases is taken. By default, the DT tries to employ the most similar case. How new cases should be constructed and under which assumptions their solutions can be synthesized again is highly domain-specific and depends on the context our framework is employed in and the connected CPPS. For instance, synthesizing multiple new cases to experiment with finding CPPS behavior optima might be a valid approach in an initial deployment setting but not during normal operations. Hence, our framework supports extension with more sophis-

ticated reuse mechanisms, such as constructing new cases by deviating case conditions and solutions systematically or interpolating between multiple similar cases.

After executing a solution, the DT uses the retain phase to learn from the result, which requires the situation before and after applying the solution. Based on the situation after executing the solution, the expected outcome is compared to the resulting of applying the solution. If the resulting outcome matches the expectation, the existing case is either reinforced as being useful or the new case is added to the case base. For the latter, the situation's properties are therefore converted into equality equations. Next, the similarity of the new case to those in the case base is assessed. If the smallest similarity is above a domain-specific learning threshold, the DT considers the case as new and adds it to the case base models. Independent of whether a new case was learned, the situation triggering the CBR, the selected cases, solutions, and outcomes are logged for the operators to support explaining system behavior.

V. Model-Based Framework for Case-Based Reasoning

By providing DSLs and adequate code generators, we enable domain experts to adapt the DT framework we built to individual CPPS and specific requirements. Figure 9 presents the realized framework for generating DTs. It contains the general DT services for storing data, sending OPC UA commands and evaluating data to identify the current system state. Furthermore, it offers the general functionality to perform CBR. The components for assessing the current CPPS state and storing data are predefined in the framework and tailored to the application scenario by generating, *e.g.,* the actual data base structure and OPC UA commands according to the domain and OPC UA models. Since all parts of the DT can be generated, explicitly no software developers are required to create a DT. The generator creates a DT that is self-adaptive based on the domain knowledge that is provided as cases. If the DT detects an unintended behavior, it adapts the configuration of the physical twin accordingly. If this does not improve the CPPS's behavior and results in the state described in the case's consequence part, the DT learns that the case is not successful and tries to apply an alternative.

The DT is tailored to a specific CPPS through models, describing this CPPS. A domain expert specifies the CPPS in a domain model and adds information for data retrieval via an OPC UA tag model. These models serve as input for the generator that creates Java code for OPC UA access, data objects, and storage. To enrich the generated DT with domain knowledge for self-adaptation, the domain expert also creates case models that characterize critical situations at runtime and how the DT should handle these situations. Besides, the domain expert specifies similarities to determine whether the CPPS situation at hand resembles one of these cases. The case and similarity models are interpreted while the DT is running. Thus after generation, the domain expert can add further cases to the case base if necessary. The DT calculates the similarity of the situation in the CPPS and a case in the case base

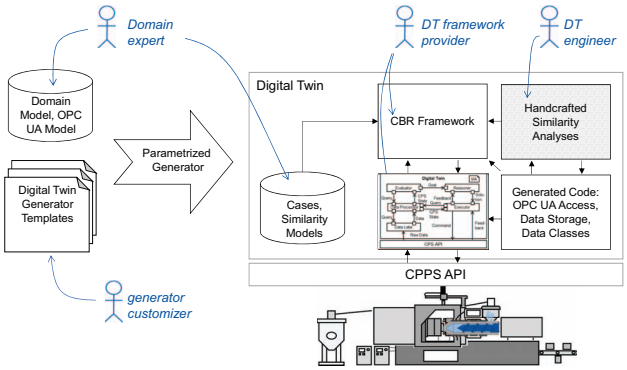

Figure 9. Framework for generating a DT based on domain knowledge provided in models.

by mapping the actual machine values with parameters in the case. Since the case and the data access are consistent with the domain model (*cf.* Figure 4), the DT can map sensor values with parameters in the case. *e.g.,* the current value of the `nozzleTemperature` (l.4) is sensed by a sensor in the machine and mapped to the parameter in the case (*cf.* Figure 6 (l.4)) when the DT calculates the similarity.

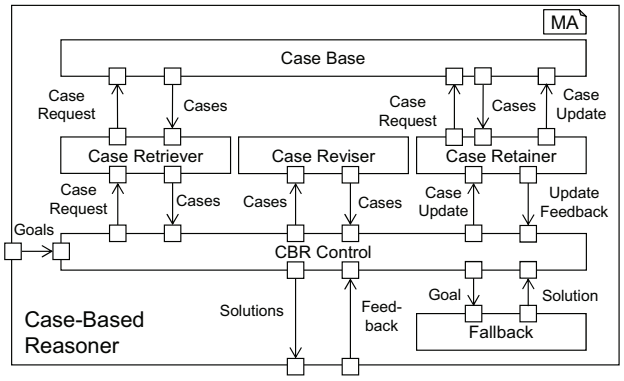

Figure 10. Internal composition of the `Case-Based Reasoner`. It connects to the `Case Base` and comprises components for the steps of the CBR cycle as well as a fallback.

Internally, the `Case-Based Reasoner` comprises six sub-components that are responsible for the individual CBR activities (*cf.* Figure 10). A control component manages the process and interacts with the respective CBR components. The `Case Retriever` obtains those cases from the `Case Base` that are similar to the current problem situation. The `Case Reviser` tailors the contained solutions to the problem at hand. Additionally, it reacts to feedback received from the Executor and further adapts the solution if necessary. When the `Case Base` does not contain known cases, the `Case Reviser` employs the `Fallback` which is usually notifying the machine operator or stopping the machine. Finally, the `Case Retainer` stores the experience, including the encountered the problem, applied solution, and its success,

in the Case Base.

We implement the CBR framework (*i.e.,* the Case-Based Reasoner and its sub-components) as an extension for the base DT architecture (*cf.* Figure 2). For that purpose, we provide a general implementation for the CBR components and define the domain-specific details via CBR models. Models of the Case Base Language describe known cases in the domain at hand and, thus, determine the Case Base contents and guide a system's management. The Evaluator monitors the system by checking the occurrence of unknown cases. The Case-Based Reasoner utilizes the known cases and the Fallback to find a solution for the detected situations. To that end, it relies on the Case Similarity Language models to determine the similarity between a case and the given situation to find an applicable case or adequately store new experiences. Fallback models provide an alternative method of solution-finding when CBR does not yield a suitable solution. The generated DT relies on this framework when performing self-adaption through CBR but is enriched with domain-specific models that experts can provide.

VI. Application Example

We created a DT with CBR for an injection molding machine as a demonstrator. The CBR framework and the DT architecture were specified by us while domain experts from injection molding created models of the CBL and CSL. We tested the generated DT on real data from a filling experiment series in injection molding. After mounting a new mold part for series production, the exact parameter settings are unknown, and the operator usually runs a so-called filling study to slowly approach an ideal configuration. Step by step, the amount of injected plastic is increased until the mold is filled. Then, fine-tuning finds a configuration that also ensures a smooth surface of the part and reduces leakage. We aim to speed up the process of finding the correct parameters while focusing on one specific machine and one mold.

Our DT is tailored to a ALLROUNDER 520 A 1500 by ARBURG. The adaption efforts can roughly be structured as follows: 1) provide data binding to the machine, 2) identify domain model, 3) devise a case base, and 4) establish a notion of similarity. For data access, the manufacturer provides an OPA UA interface through which the DT can access runtime data.

In cooperation with domain experts from injection molding, we identified representative parameters (*cf.* Figure 11) for capturing the machine's state. PhaseData comprises all parameters of an injection process. DosingTime determines for how long plastic is loaded into the plasticizing unit. The attributes cylinderHeating, injectionFlow, and switchOverVolume (ll. 7-9) describe the injection parameters. cylinderHeating sets the temperature inside the plasticizing unit, injectionFlow and switchOverVolume influence how fast and long plastic is injected. The meltCushion (l. 12) is the surplus of material left in the plasticizing unit after the injection.

```
01 classdiagram InjectionMolding {                    CD
02   class PhaseData {
03     int cycleId;
04
05     double dosingTime;
06
07     double cylinderHeating;
08     double injectionFlow;
09     double switchOverVolume;
10
11     double backPressure;
12     double meltCushion;
13     // further attributes
14   }
15 }
```

Figure 11. Injection molding domain model. PhaseData comprises parameters of the production of a single part. Only the most critical parameters are depicted.

```
01 import InjectionMolding.PhaseData;                 CBL
02
03 case tooMuchMaterial {
04   if PhaseData.meltCushion > 20
05   fallback notify("Material remainder high")
06 }
07
08 case increaseBackPressure {
09   if PhaseData.switchOverVolume == 70 &&
10      PhaseData.meltCushion == 20 &&
11      PhaseData.backPressure == 10
12   do PhaseData.backPressure = 15
13   yields PhaseData.meltCushion == 12 &&
14   // further effects
15 }
16
17 case injectMore {
18   if PhaseData.switchOverVolume < 70 &&
19      PhaseData.meltCushion < 10
20   do PhaseData.switchOverVolume = 70
21   yields  // further effects
22 }
```

Figure 12. Example cases from the case study in injection molding. The first case defines the problematic parameter space. The others are solutions to handle more specific situations. Repeating imports are omitted.

Figure 12 exemplarily shows cases identified for the case study. The first case is unknown (ll. 3-6). It describes the problematic parameter state of having too much residual material. The other two cases (ll. 8-21) feature possible solutions. The first handles a situation where too little material is injected. The specified backPressure is too low, leading to missing material in the mold. The second covers the situation where more material can be injected.

For similarity, we employ a weighted similarity calculation, as specified in Figure 13. The local similarities define the critical parameters with influence on the metric (ll. 4-8). The values of backPressure and dosingTime are more sensitive to changes. Therefore, we use a squared local similarity for them. It is offered by the CSL where the difference is squared. The global similarity characterizes the weights (ll. 10-16). The similarity of switchOverVolume has the most influence with a weight of 0.4. A minor influence has the similarity of cylinderHeating with a weight of 0.05.

Using models and the customized DT components, one

```
import InjectionMolding.PhaseData;                                    CSL

case similarity InjectionMolding {
  local PhaseData.meltCushion      absolute;
  local PhaseData.switchOverVolume absolute;
  local PhaseData.backPressure     squared;
  local PhaseData.dosingTime       squared;
  local PhaseData.cylinderHeating  absolute;

  global weighted [
    PhaseData.meltCushion      = 0.2,
    PhaseData.switchOverVolume = 0.4,
    PhaseData.backPressure     = 0.2,
    PhaseData.dosingTime       = 0.15,
    PhaseData.cylinderHeating  = 0.05
  ];
}
```

Figure 13. Model for similarity calculation in injection molding.

receives a fully functional DT generated based on domain models, case models, and similarity models to provide a CBR for an injection molding machine.

For evaluating the generated DT, we captured the cycle-times for CBR cycle execution in the DT while it was running on a local computer with Intel(R) Core(TM) i7-7600U CPU. The DT operated on real historical data of the injection molding machine but, due to safety issues, could not change settings on the machine.

Initially, the DT started with a case base of 20 cases that we identified in cooperation with the injection molding experts. The measured results are displayed in Table I. The DT's

Table I
CYCLE TIMES OF THE DT'S CBR CYCLE.

	Minimum *ms*	**Maximum** *ms*	**Average** *ms*
First Cycle	$16,9181$	$110,2881$	$42,2136$
No Case	$2,6234$	$50,6226$	$16,16197$
Case Detected	$1,7137$	$75,6592$	$13,01902$

CBR cycle was triggered every time that the machinecycle counter in the machine changed. This parameter simply counts the number of performed production cycles on the ARBURG. During the first cycle, the DT loads the initial cases from the file system. Consequently, this cycle's duration had a longer execution time ($42,2136ms$ in average) than other cycles. The DT monitors the injection molding machine and compares the current state to identified cases. When no case matched the current machine's state, this comparison took $16,16197ms$ on average. If a case was present, the DT detected it within $13,01902ms$ on average and tried to adapt its behavior based on the solution stated in the case model. If the machine data confirmed the case's success, the DT marked the applied case as successful or unsuccessful, respectively. We expect an increase in cycle time due to communication latency if the DT connects to the CPPS and autonomously changes process parameters. Given that injection molding is a cycle-based process where the process settings can only be updated for the next cycle, and that production cycles take between 50 seconds and 2 minutes, the computing times of the DT are sufficient to adapt the process settings in time.

VII. DISCUSSION

We applied the presented framework and modeling languages to create a DT of an injection molding machine. The realized DT establishes a connection to the injection molding machine and reads its sensor values. Based on these, the DT autonomously detects unintended system behavior and produces solutions based on similar cases provided by the case base.

While cases consisting of conditions and effects are very intuitive, the modeling languages of our framework rely on some experience with data types and structures (int, float, Boolean, objects), an understanding of model relations (imports), and might even relate to PDDL knowledge bases. The first challenge can be mitigated by providing even more domain-specific extensions of the CBL that rely only on data types and data structures well known by the domain experts and by intelligently translating these to the data structures communicated via OPC UA to the CPPS. The notion of model imports could be omitted by fixing a CBR DT to a single domain model class diagram and adjusting the CBL again. Similarly, PDDL fallbacks can be prohibited in domain-specific sublanguages of the CBL. Hence, the languages employed within our framework can be tailored precisely to the complexity suitable for the domain experts operating the systems. These, of course, limit the usefulness of the overall framework. Nonetheless, due to MontiCore's language extension mechanisms, making the language as comprehensibly as necessary is possible (**R1**). In general, CBL and CSL were regarded as easy to understand and use. However, injection molding experts had difficulty in explicitly expressing similarities between cases because they often also work by gut feeling and could not pinpoint the exact point that triggers their adaptation of the machine configuration.

The generated DT works autonomously (**R2**) and evaluates the current CPPS state every time that a new production cycle starts. If the CPPS state matches the condition of a case, the DT adapts the CPPS configuration based on the solution specified in the case. If this adaptation does not lead to the expected behavior of the CPPS the DT learns to prioritize this case lower in the future.

The presented DT can connect to any CPPS that provides a communication interface; thus, it receives the data for evaluating if any unintended situations occurred. Active writing of parameters to the machine while it is running remains critical and, due to liability issues, might be prohibited in other domains. Nonetheless, the DT provides solutions for detected cases and attempts to implement these autonomously. If the connected CPPS prohibits manipulation of settings without human interaction, the DT can at least provide a recommendation for adapting the machine configuration. Moreover, the presented framework is reusable for other CPPS (requirement **R3**) as essential parts of the DT are modeled independent of the underlying CPPS. Transferring the DT to another CPPS

requires implementation of adapters to communicate with the machine, manipulating the domain model, and specifying an application-specific case base and similarity measurements. The model-driven development of the DT based on a generator that derives the concrete implementation from domain models further speeds up the development process. Overall, the various configuration means support tailoring our approach to a variety of self-adaptive manufacturing scenarios (**R3**).

When the DT that we realized encounters new cases, it first searches through the case base to find the most similar case and tries to adapt its solution to the case at hand. If this adaptation is successful, it creates a new case and adds it to the case base. Thus, when running over a more extended period of time the DT learns more cases and becomes more effective. Thus, the realized DT improves over time by persisting experiences that domain experts can review as explanations of self-adaptive behavior (**R4**). Interesting challenges arise due to the indeterministic nature of CPPS, the actions taken in the past may not be relevant for similar cases in the future. Nonetheless, since the DT is able to adapt cases in terms of their success, it at least does not try to apply solutions that verifiably do not lead to desired situations.

VIII. Related Work

An approach similar to ours utilizes an IIoT Gateway with an OPC UA interface as a mediator between a DT and the physical system [21]. We suggest exchangeable adapter components for both, data retrieval and control, supporting a range of different communication technologies and protocols. A different idea investigates model-based DTs that support and guide product development in all phases of the life cycle [22]. During design and engineering, DTs comprise collections of digital artifacts (data and models) to provide simulations of the expected system behavior.

A similar concept utilizes DTs to merge different kinds of system data to model its behavior [23]. Thus, the DT shows the effect of design changes on the physical system and supports virtual verification of its behavior. Further research demonstrates the extent of technologies and application domains for DTs in manufacturing. In a framework for smart workshops, DTs control CPPS, providing local optimizations and communicating to achieve a global optimization [24]. Another approach employs edge, fog, and cloud computing to implement DTs [25]. The DTs control physical entities via virtual models and are connected on a network level or through the cloud to perform optimizations of increasing degree. However, these proposed DTs are tailored to the given tasks or application domain while we present a customizable approach that is applicable to a wide variety of purposes.

Autonomic system must be able to handle unexpected and novel situations. Thus, CBR is well suited for application in autonomic systems and especially in DTs. This includes employing CBR for self-configuration in autonomic systems [26] or utilizing CBR to detect and repair system failures at runtime (self-healing) [27]. These approaches face the cold-start problem, though. As a solution is derived from existing cases, considerable effort and knowledge about the domain is required to set up an extensive case base. A solution to the cold-start problem is a combination of CBR and goal reasoning [28], [29]. A case-based reasoner tries to solve problems based on the experiences in the case base. If the cases do not yield a solution, the system applies goal reasoning as a fallback to create new cases and adds these to the case base. An alternative is building the case base in an offline learning phase [30]. The approach utilizes reinforcement learning for creating new cases. In the online learning phase, the system finds appropriate cases via CBR and applies reinforcement learning to adapt the solution to the situation at hand.

Multiple contributions deal with CBR in the domain of injection molding. A prototype recommendation system for parameter determination provides an interface for manual parameter input and suggests corrections using CBR [31]. Another interactive user system for the shop floor determines parameters first by CBR and improves these through a rule-based system [32].

Further research reviews different methods for parameter determination injection molding [33]. The authors identify CBR as one of three main approaches. They report no commercial or systematic solution since feedback on quality remains challenging. Instead of requiring manual reading and writing of parameters, a more integrated approach incorporates the injection machine into the system [34]. However, it employs a very rudimental read/write approach without the goal of creating a digital machine representative. A concept of DTs in injection molding identifies all different phases of the process and their linking [35]. However, it provides no defined method for the individual steps. Our system focuses on the manufacturing procedure on the machine itself and employs CBR for this.

IX. Conclusion

To leverage CBR over domain expertise into self-adaptive manufacturing, we devised a modeling framework comprising multiple interrelated modeling languages and integrate it into our architecture for DTs [5], [6], [7]. We have presented a collection of modeling languages to support domain experts in encoding their knowledge into DTs that perform self-adaptation at runtime. This enables the DTs to react to unforeseen situations quickly and learn from past situations. Models of these languages describe domain-specific cases and their similarity and are processed by a modular DT architecture that manages the CBR cycle of retrieving cases similar to the current situation, reusing these to handle the situation, revising these if necessary, and retaining these if the revisions were successful. The realized framework is not tailored to one specific CPPS but can be customized to any other CPPS. The model-driven development simplifies developing DTs and can lead to more efficient manufacturing, less misproduced goods, and, ultimately, reduced production cost.

REFERENCES

[1] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling Languages in Industry 4.0: an Extended Systematic Mapping Study," *Software and Systems Modeling*, vol. 19, no. 1, pp. 67–94, January 2020.

[2] F. Tao, H. Zhang, A. Liu, and A. Y. Nee, "Digital twin in industry: State-of-the-art," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2405–2415, 2018.

[3] A. Aamodt and E. Plaza, "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches," *AI Communications*, no. 1, pp. 39–59, 1994.

[4] C. K. Riesbeck and R. C. Schank, *Inside Case-Based Reasoning*. Psychology Press, 2013.

[5] P. Bibow, M. Dalibor, C. Hopmann, B. Mainz, B. Rumpe, D. Schmalzing, M. Schmitz, and A. Wortmann, "Model-driven development of a digital twin for injection molding," *Advanced Information Systems Engineering: 32nd International Conference on Advanced Information Systems Engineering, CAiSE 2020, France, June 8-12, 2020, Proceedings*, 2020.

[6] J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, October 2020, pp. 90–101.

[7] M. Dalibor, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits," in *International Conference on Conceptual Modeling*. Springer, 2020, pp. 377–387.

[8] M. Fox and D. Long, "PDDL2. 1: An extension to PDDL for expressing temporal planning domains," *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003.

[9] J. Hoffmann, "The Metric-FF Planning System: Translating"Ignoring Delete Lists"to Numeric State Variables," *Journal of artificial intelligence research*, vol. 20, pp. 291–341, 2003.

[10] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *jair*, vol. 14, pp. 253–302, 2011.

[11] K. Adam, A. Butting, R. Heim, O. Kautz, J. Pfeiffer, B. Rumpe, and A. Wortmann, *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse*, ser. Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.

[12] K. Hölldobler, B. Rumpe, and A. Wortmann, "Software Language Engineering in the Large: Towards Composing and Deriving Languages," *Computer Languages, Systems & Structures*, vol. 54, pp. 386–405, 2018.

[13] K. Hölldobler and B. Rumpe, *MontiCore 5 Language Workbench Edition 2017*, ser. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[14] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh *et al.*, "The state of the art in language workbenches," in *International Conference on Software Language Engineering*. Springer, 2013, pp. 197–217.

[15] T. Greifenberg, M. Look, S. Roidl, and B. Rumpe, "Engineering Tagging Languages for DSLs," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE. Ottawa, ON, Canada: IEEE, 2015, pp. 34–43.

[16] A. Butting, A. Haber, L. Hermerschmidt, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language," in *European Conference on Modelling Foundations and Applications (ECMFA'17)*, ser. LNCS 10376. Springer, July 2017, pp. 53–70.

[17] A. Butting, O. Kautz, B. Rumpe, and A. Wortmann, "Architectural Programming with MontiArcAutomaton," in *ICSEA 2017*. IARIA XPS Press, May 2017, pp. 213–218.

[19] S.-H. Leitner and W. Mahnke, "Opc ua–service-oriented architecture for industrial applications," *ABB Corporate Research Center*, vol. 48, pp. 61–66, 2006.

[18] M. Dalibor, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits," in *Conceptual Modeling*, G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, and H. C. Mayr, Eds. Springer International Publishing, October 2020, pp. 377–387.

[20] B. Rumpe, *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017. [Online]. Available: http://www.se-rwth.de/mbse/

[21] V. Souza, R. Cruz, W. Silva, S. Lins, and V. Lucena, "A Digital Twin Architecture Based on the Industrial Internet of Things Technologies," in *2019 IEEE International Conference on Consumer Electronics (ICCE)*. Las Vegas, NV, USA: IEEE, Jan 2019, pp. 1–2.

[22] S. Boschert and R. Rosen, *Digital Twin—The Simulation Aspect*. Cham: Springer International Publishing, 2016, pp. 59–74. [Online]. Available: https://doi.org/10.1007/978-3-319-32156-1_5

[23] F. Tao, J. Cheng, Q. Qi, M. Zhang, H. Zhang, and F. Sui, "Digital twin-driven product design, manufacturing and service with big data," *The International Journal of Advanced Manufacturing Technology*, vol. 94, no. 9-12, pp. 3563–3576, 2018.

[24] J. Leng, H. Zhang, D. Yan, Q. Liu, X. Chen, and D. Zhang, "Digital twin-driven manufacturing cyber-physical system for parallel controlling of smart workshop," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 3, pp. 1155–1166, 2019.

[25] Q. Qi, D. Zhao, T. W. Liao, and F. Tao, "Modeling of Cyber-Physical Systems and Digital Twin Based on Edge Computing, Fog Computing and Cloud Computing Towards Smart Manufacturing," in *ASME 2018 13th International Manufacturing Science and Engineering Conference*, American Society of Mechanical Engineers. College Station, Texas, USA: ASME, 2018, pp. V001T05A018–V001T05A018.

[26] M. J. Khan, M. M. Awais, and S. Shamail, "Improving Efficiency of Self-Configurable Autonomic Systems Using Clustered CBR Approach," *IEICE TRANSACTIONS on Information and Systems*, vol. 93, no. 11, pp. 3005–3016, 2010.

[27] S. Montani and C. Anglano, "Achieving self-healing in service delivery software systems by means of case-based reasoning," *Applied Intelligence*, vol. 28, no. 2, pp. 139–152, 2008.

[28] W. Qian, X. Peng, B. Chen, J. Mylopoulos, H. Wang, and W. Zhao, "Rationalism with a Dose of Empiricism: Case-Based Reasoning for Requirements-Driven Self-Adaptation," in *2014 IEEE 22nd International Requirements Engineering Conference (RE)*. Karlskrona, Sweden: IEEE, Aug 2014, pp. 113–122.

[29] ——, "Rationalism with a Dose of Empiricism: Combining Goal Reasoning and Case-Based Reasoning for Self-Adaptive Software Systems," *Requirements Engineering*, vol. 20, no. 3, pp. 233–252, 2015.

[30] T. Zhao, W. Zhang, H. Zhao, and Z. Jin, "A Reinforcement Learning-Based Framework for the Generation and Evolution of Adaptation Rules," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*, IEEE. Columbus, OH, USA: IEEE, 2017, pp. 103–112.

[31] C. K. Kwong, G. F. Smith, and W. S. Lau, "Application of case based reasoning injection moulding," *Journal of Materials Processing Technology*, vol. 63, no. 1-3, pp. 463–467, Jan. 1997.

[32] K. Shelesh-Nezhad and E. Siores, "An intelligent system for plastic injection molding process design," *Journal of Materials Processing Technology*, vol. 63, no. 1-3, pp. 458–462, Jan. 1997.

[33] H. Gao, Y. Zhang, X. Zhou, and D. Li, "Intelligent methods for the process parameter determination of plastic injection molding," *Frontiers of Mechanical Engineering*, vol. 13, no. 1, pp. 85–95, Mar. 2018.

[34] H. Zhou, P. Zhao, and W. Feng, "An integrated intelligent system for injection molding process determination," *Advances in Polymer Technology*, vol. 26, no. 3, pp. 191–205, 2007.

[35] Y. Liau, H. Lee, and K. Ryu, "Digital Twin concept for smart injection molding," in *IOP Conference Series: Materials Science and Engineering*, vol. 324, Mar. 2018, p. 012077.