Virtual Prototyping a Production Line Using Assume–Guarantee Contracts

Stefano Spellini, Roberta Chirico, Marco Panato, Michele Lora, *Member, IEEE*, and Franco Fummi, *Member, IEEE*

***Abstract*—This article presents a methodology to formalize the behavior of the machines composing a production line, and to automatically generate their virtual prototypes for efficient and correct plant simulation. The approach exploits assume-guarantee reasoning through contracts to model the interaction between the different components of a production line. The approach is guided by a well-known taxonomy of industrial machines and associated manufacturing processes to identify each elementary action related to a specific machine. Contracts enable to build executable models of all the machines available in the production line by using automatic synthesis. The generated models can be integrated into a state-of-the-practice industrial plant simulation software to estimate and validate the production line's behavior. The presentation of the methodology is supported by a running example based on a real production line, showing the step-by-step application of the approach to a concrete scenario.**

***Index Terms*—Advanced manufacturing, design automation, simulation and validation, virtual prototyping.**

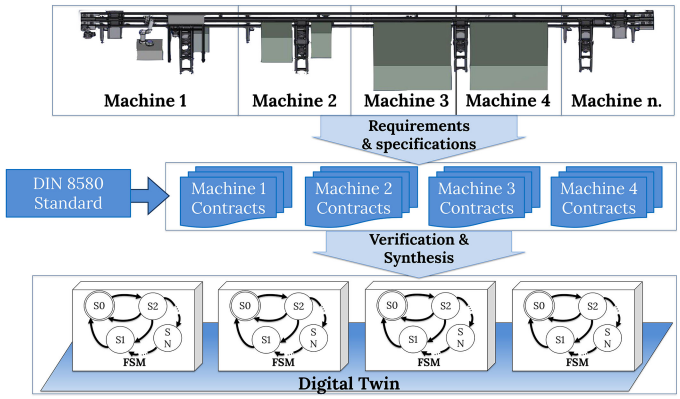

Fig. 1. Summary of the contribution: the production line is formalized by a set of contracts under the guidance of the DIN 8580 Standard. The contracts are used for the automatic synthesis of a virtual prototype of the production line.

I. Introduction

THE RECENT introduction of key-enabling technologies such as cyber-physical system, cloud computing, and internet of things into manufacturing processes have brought to life a new trend identified by the term Industry 4.0 [1]. These technologies co-operate to improve production efficiency. Among these novel concepts, *digital twins* have been introduced to replace expensive physical prototyping [2]. A digital twin is a model of the production plant used to simulate the manufacturing process before it is physically built, to early predict possible errors or production bottlenecks.

In this context, manufacturing plants are becoming every day more complex, involving different processes, requiring a high degree of reconfigurability and little to no-human presence directly working on machines. Furthermore, programmable devices technologies have matured enough to be integrated easily into manufacturing processes [1]. Additive and subtractive manufacturing are now efficient and cost-effective practices [3].

With such diversified scenarios of production requirements and production chains, it becomes fundamental to develop new methodologies for designing, verifying and simulating the manufacturing line to be built. Such methodologies imply the adoption of formal specifications, thus increasing the computational complexity involved in the design process: problem decomposition becomes mandatory to tackle such issues. The Assume-Guarantee (A/G) Contracts theory provides formal support to problem representation and decomposition, and to compositional reasoning about system design [4]. A/G contracts allow decomposing the problem of designing a system into smaller sub-problems representing different components or aspects of the complete design problem.

This article presents an approach, summarized in Fig. 1, exploiting a contract-based representation to build virtual prototypes of production lines. The behavior of each production machine available in the line is represented by a set of A/G contracts, whose assumptions and guarantees are expressed by using the linear temporal logic (LTL). The formalization is guided by the *Deutsches Institut für Normung (DIN)* standard 8580 on industrial machinery [5], in order to identify the granularity of the base actions considered in our approach. Each contract defined to model the system is synthesized into a finite state

Manuscript received September 26, 2020; accepted October 24, 2020. Date of publication November 17, 2020; date of current version June 16, 2021. This work was supported by the Ministry of Education, University and Research "dipartimenti Di Eccellenza" 2018–2022 Grant. Paper no. TII-20-4477. *(Corresponding author: Stefano Spellini.)*

The authors are with the Department of Computer Science, University of Verona, 37134 Verona, Italy (e-mail: stefano.spellini@univr.it; roberta.chirico@univr.it; marco.panato_01@univr.it; michele.lora@univr.it; franco.fummi@univr.it).

Color versions of one or more of the figures in this article are available at https://doi.org/10.1109/TII.2020.3038679.

Digital Object Identifier 10.1109/TII.2020.3038679

machine (FSM) implementing the behavior specified by the contract. This step relies on state-of-the-art synthesis techniques [6], [7], and it produces a C++ implementation of the contract. The machine's executable model is transformed into a block to be imported into a commercial industrial plant simulator [8]. Then, the plant simulator can be used to simulate the entire system. The virtual prototypes generated by the proposed methodology may be used to perform system-level validation through simulation. The advantage of creating system simulations from the formal specification of its sub-components is the possibility to formally validate the behavior of the single components. Thus, leaving to simulation only the burden to validate the composition.

The methodology presentation is paired with its application for the formalization of a robotic arm manipulator, and in particular to the "Turn" operation provided by the robot. Then, we apply the methodology to the entire production line comprising the robotic arm and we report the obtained results.

II. Background

A. State-of-the-Art and Contribution

Several tools have been developed in order to model and simulate industrial production systems that simplify the validation and the optimization of a manufacturing plant [9]. System-level simulation may rely on models at different abstraction levels and using diverse models of computations [10]. Such abstractions may go from the physical level, where every mechanical detail of each production machine is modeled by a set of differential equations, to the functional level, where the system is modeled as a set of machine actions and interactions between multiple machines. Indeed, the more detailed is a model, the more complex will be its simulation. As such, the choice of model's abstraction level must be based on the model's purpose. Discrete-event models [11] represent systems as the set of events occurring throughout its life. As such, they provide a high-level of abstraction while representing the essential details of the system behavior. For this reason, they are widely used to simulate manufacturing systems, and many commercial simulation tools relying on the event-based paradigm are available [12]. Different system stakeholders aim at evaluating various features of the manufacturing system using simulation. For instance, a designer may be interested in the validation of the production process, whereas a system engineer may aim at evaluating whether an already deployed production line could be optimized to increase the manufacturing line throughput. For this reason, each simulator can simulate different aspects of a manufacturing system.

In most cases, simulators are equipped with placeholder components depicting generic manufacturing processes. However, they usually provide extension mechanisms to precisely define the machine's behaviors by constructing and importing new custom models. Most of the available simulators provide interfaces to a well-known programming language (e.g., C/C++), thus allowing the definition and the import of custom models into the simulator. Tecnomatix Plant Simulation [8] is an industrial-grade tool [13], widely popular among industrial actors being also recognized as one of the most versatile tools available for the simulation of industrial processes [14]. It provides many interfaces to external tools and languages.

In order to simulate a manufacturing plant, it is necessary to produce its model first. While this is reasonably doable when designing a line from scratch, creating models of already existing machines may be an error-prone process that may lead to inaccurate models [10]. Discrete-event models may be also derived automatically by analyzing the system behavior [15]. However, the quality of the produced models will be constrained by the quality of the executed scenarios. As such, it may be ideal to start from formal specifications of systems and components when aiming at producing the executable models used for simulation. A formalization for manufacturing control systems has been developed for verification purposes in [16]. The authors developed a framework to automatically translate specifications to Linear Temporal Logic (LTL) formulas and using model checking techniques to verify the plant consistency. In [17], a method to formally specify industrial component behaviors has been proposed: It focuses on control logic components, the sensor/actuator behaviors and it presents how to build a formal specification to verify their correctness. However, none of these approaches relies on simulation but only on formal methods that usually lead to complexity issues. System execution is used to perform runtime verification of industrial systems [18]. However, this is usually applied to the control systems, rather than on the actions physically performed by the production line. Furthermore, it usually does not consider the state of the product and its evolution.

A combination of formal methods with simulation is described in [19]. The authors developed a formalism to specify properties over the system while being expressive enough to be translated into an Finite State Machine (FSM) useful to simulate the system. Spellini *et al.* [7] combined automatic synthesis from LTL specifications with simulation to design and validate a robotic transportation system. However, the proposed formalization is specifically tailored for robotics applications.

In this work, we extend the state-of-the-art by proposing a systematic (e.g., guided by the DIN 8580 standard) formalization of the production machines features. Then, we present a methodology producing the digital-twin of a manufacturing system from its formalization.

B. DIN 8580 Standard for Industrial Machinery

The DIN 8580 standard [5] defines a wide set of processes, products, activities, and facilities connected to the industrial domain. A *manufacturing process* is the production or the transformation of a workpiece. A process can be divided into multiple subprocesses, each of them changing or forming a different property or shape of the processed product. The DIN states that every manufacturing process can be classified into five main groups, according to the type of material transformation they provide, in particular: *Primary shaping*, *Forming*, *Cutting*, *Joining*, and *Coating*. These main groups are divided into subgroups, further characterizing processes by delineating elementary actions associated to the concept of manufacture, i.e.,

the Joining group is divided into *operations* such as assembling, fastening, soldering, etc.

The DIN 8580 taxonomy allows defining the granularity to use to describe a process. In particular, we consider the transformations defined in the taxonomy as the "atomic" actions of the considered manufacturing process, even if these transformations may be further decomposed into subactions. For instance, the "Turn" action performed by a robotic arm can be further decomposed into a set of rotations. However, the proposed modeling approach will consider the "Turn" transformation as an atomic action.

C. (A–G) Contracts for Reactive Systems

A contract C for a component M is a triplet (V, A, G), where V is the set of the component variables, and A and G are *assertions*, each representing a set of behaviors over V[4]. A are the contract's *assumptions:* The behavior of the environment of M assumed by the model. G are the *guarantees:* The behaviors guaranteed by M whenever the assumptions hold.

A component M implements a contract C whenever M and C are defined over the same set of variables, and all the behaviors of M satisfy the guarantees of C in the context of the assumptions. Moreover, a component E can also be associated with a contract C as an environment for the contract. E is said to be a legal environment of C, whenever the behaviors implemented by E are a subset of A. The A/G contract theory [4] defines a set of operations, such as *composition*, *compatibility*, *consistency*, and *refinement*, useful to manipulate the models throughout the design phases.

Behaviors described by assumptions or guarantees of a contract may be expressed using different formalisms [20]. Temporal contracts often rely on LTL formulas to describe reactive system [21]. However, the realizability of an LTL contract is an intractable problem due to its double exponential complexity [22].

Reducing the expressiveness of the formal language used for specification is a viable solution to reduce the complexity. In particular, the realizability of a specification belonging to the general reactivity of rank 1[GR(1)] fragment of the LTL can be solved in polynomial time [23]. A General Reactivity of rank 1 (GR(1)) formula is structured as follows [24]:

$$\left(\theta^e \wedge \Box\rho^e \wedge \bigwedge_{0<i\leq \mathrm{J}} \Box\Diamond J_i^e\right) \rightarrow \left(\theta^s \wedge \Box\rho^s \wedge \bigwedge_{0<i\leq \mathrm{J}} \Box\Diamond J_i^s\right)$$

where θ^e and θ^s are the environment's and system's initial conditions; ρ^e and ρ^s are the environment's and system's safety properties; J^e and J^s are the environment's and system's liveness properties. Each of them is a boolean formula defined over a set of environment and system variables. However, the *"next"* operator (i.e., $\bigcirc$) can be used within the formulas, being the operator considered "sugar syntax" expressing a relation between a pair of boolean variables. In the abovementioned formula, the premises of the logical consequence represents the assumptions, whereas the conclusion represents a guarantee.

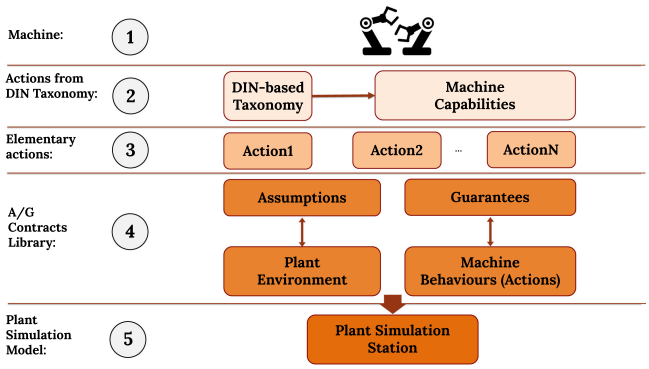

Fig. 2. Overview of the presented approach: Starting from a taxonomy of industrial machines, elementary actions associated with this machine are defined. Then, a A/G contracts library defining each action is assembled and synthesized, generating a plant simulation-compatible model that can be imported and simulated.

Thus, GR(1) specifications are intrinsically implementing a assume/guarantee mechanisms. As such, GR(1) is extremely suitable to express contracts. Indeed, it does not allow representing certain valid LTL formulas, still it allows specifying safety and liveness properties that are usually sufficient to express systems behaviors for validation and verification purposes [25].

III. Methodology Overview

The proposed methodology builds a formal representation of a production line through executable models of manufacturing machines, and it composes them into its virtual prototype. The models are meant to be simulated for evaluating the feasibility and correctness of the production line before building, deploying, and assembling the real plant. Fig. 2 summarizes the steps of the proposed approach to validate a production line.

Initially, the components are classified according to the DIN 8580 standard, cataloguing actions and industrial processes associated with production machines. For each machine, the DIN standard identifies and details a set of actions that the machine is built to execute. Each action identified in the standard is formalized as a A/G contact. An action describes a specific behavior of the component defining a set of guarantees (e.g., the ability to perform a certain modification to the worked material shape), and assumptions that specify the conditions necessary to perform the action (e.g., the presence of the worked material within the action's range of the component). Each manufacturing machine is represented by a contract that is the composition of the contracts modeling the machine's actions. This allows representing a manufacturing machine as the composition of its actions. As such, it would be possible to verify the consistency of the machine formally. However, verifying the consistency of the composition is affected by computational complexity issues [21]. For this reason, a *coordination contract* is specified to model the machine coordinating the actions identified. The coordination contract assumes the actions' behaviors, while it guarantees the possibility to perform all the actions. Such formalization of a manufacturing machine leads to a two-layer hierarchy of contracts: the first layer being composed by the

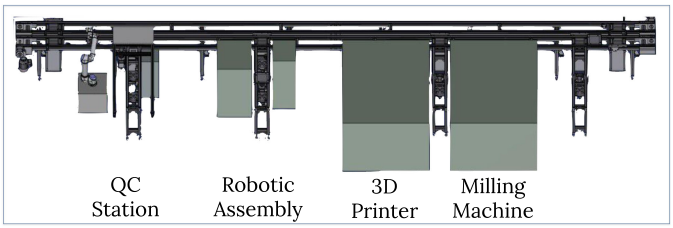

Fig. 3. Schematic of the ICE laboratory production line.

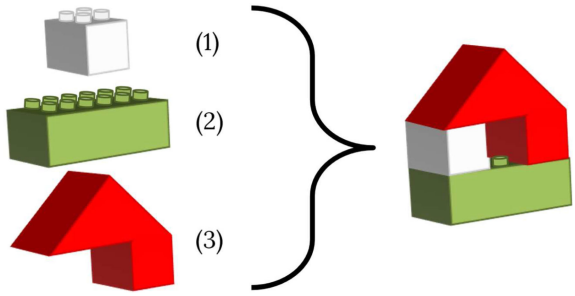

Fig. 4. Three-dimensional representation of the parts that compose the final product of the case study. The (1) and (2) pieces are gathered from the warehouse, whereas (3) has to be 3-D printed.

actions contract, while the second layer is the coordination contract. Such hierarchical decomposition allows verifying each action separately and then to verify the coordination between actions. Thus, it allows decomposing the verification problem both horizontally (i.e., over the different actions), and vertically (i.e., over the two different layers of the contracts hierarchy).

All the contracts are specified using the GR(1) fragment of LTL. This allows keeping the check of consistency computationally tractable. The methodology goes on by checking the consistency of each contract. Then, for each contract, it synthesizes a control strategy, i.e., an implementation of the guarantees given the assumptions. This allows producing an FSM implementing the specification expressed by the contract.

The final step of the methodology creates the digital-twin of each machine and then the one for the entire manufacturing line to be executed within Tecnomatix Plant Simulation. It generates a Plant Simulation "*stations*," a simulator-specific object simulating the behavior of a machine's controller. Plant Simulation adopts the concept of mobile unit (MU) to represent an abstraction of every physical object that moves throughout the production plant. Thus, each station describes the processing of a Mobile Unit (MU), outlining the variation of the object's properties. In other words, each station simulates the behavior of a machine manipulating the working material. The stations are instantiated within a plant model in the simulator according to the manufacturing line's initial specifications, thus assembling the plant's complete digital-twin from the component manufacturing machines models. Finally, simulating the plant model allows validating the production line.

The following of the article describes each step of the methodology applied to the case study presented hereby.

A. Case Study: Additive Manufacturing and Assembly

The case study used throughout the article is based on the industrial computer engineering (Industrial Computer Engineering (ICE)) laboratory,[1] a research facility focused on advanced manufacturing and equipped with a full-size configurable production line. Fig. 3 shows the structure of the manufacturing system, as depicted in the plant simulation tool used in this work. The production plant is composed (from right to left) of a milling machine, a plastic fused deposition modeling 3-D printer, a collaborative robotic assembly cell composed of two robotic arms composing an assembly station, and a quality check (QC) station. The transportation of final products or workpieces toward different stations is implemented through a complex system of conveyor belts. Raw materials, the components, as well as the final products, are stored in an automated warehouse.

The system is used to produce the object depicted in Fig. 4. The first operation is the 3-D printing of a small plastic LEGO-like brick (i.e., piece number (3) in Fig. 4). Once printed, the piece is checked for defects by using the Quality Check (QC) station. Meanwhile, two plastic bricks (i.e., pieces (1) and (2) of Fig. 4) are gathered from the warehouse to be assembled by the robotic assembly station. Finally, the printed piece is assembled to the ensemble of the other two pieces. The final product is verified in the QC cell for assembly defects. Finally, the product is transported to the warehouse.

The analysis of the actions in the DIN 8580 taxonomy highlights that some of these can be implemented by the composition of more elementary actions. In the abovementioned example, the identified actions can be decomposed as follows:

1) *Dismantle:* Decompose.
2) *Emptying:* Pick, Turn, Place.
3) *Shift one into another:* Pick, Move, Place.
4) *Screw:* Turn.
5) *Clamp:* Pick, Move, Turn, Place.
6) *Embedding:* Pick, Move, Place.

Therefore, we identify the elementary actions of the collaborative manipulators.

1) *Compose:* constitute or make up a piece.
2) *Decompose:* separate into two elements the piece.
3) *Pick:* collect the piece from a specific location.
4) *Place:* drop the piece to a specific location.
5) *Move:* move a piece.
6) *Turn:* rotate the piece to a specific angle.

In the following, the description of the formalization will be paired with its exemplification of the "Turn" operation, performed by the manipulator robot.

IV. Components Formalization

In the following, we assume that the same concepts can be applied to different machines, while we will concentrate on the formalization of the functionality associated to the robotic

[1]University of Verona, Department of Computer Science, Computer Engineering for Industry 4.0: *http://icelab.di.univr.it/*

TABLE I
SKETCH OF THE COORDINATOR AND TURN OPERATION CONTRACTS

Coordinator contract C_{coord}	**Turn operation contract** C_{turn}
$C_{coord} = (V_{coord}, A_{coord}, G_{coord})$	$C_{turn} = (V_{turn}, A_{turn}, G_{turn})$
$V_{coord} = \{grip, command, turn\_executed, place\_executed,$ $pick\_executed, available, angle, desired\_angle\}$	$V_{turn} = \{command, turn\_executed, grip, angle, desired,$ $step, turning\}$
$A_{coord} = \{\square\lozenge(available),$ $\square\lozenge(turn\_executed)\}$	$A_{turn} = \{\square(command = Turn \rightarrow grip = true),$ $\square\lozenge(command = Turn)\}$
$G_{coord} = \{(\square(turn\_executed \rightarrow \bigcirc(available))),$ $(\square(command = pick \wedge pick\_executed \rightarrow grip = true)),$ $(\square(command = place \wedge place\_executed \rightarrow grip = false)),$ $(\square(angle \neq desired \wedge grip = true \rightarrow command = Turn))\}$	$G_{turn} = \{(\square(command = Turn \leftrightarrow \bigcirc(turning))),$ $(\square(turning \wedge (angle < desired) \rightarrow \bigcirc(angle = angle + step))),$ $(\square(angle = desired \rightarrow \bigcirc(angle = desired))),$ $(\square(angle = desired \rightarrow turn\_executed))\}$

assembly cell (i.e., the collaborative manipulators). The two collaborative robots can manipulate objects and precisely assemble multiple pieces of material into a single product. The actions identified in the previous section can be further divided into two subgroups: those requiring a single manipulator arm, and those actions performing an assembly or disassembly procedure, thus requiring both manipulator arms to be active. The proposed formalization specifies a sequential behavior of the system and its components, enforcing its constrained evolution in discrete-time steps. At the chosen abstraction, action's specification differ due to the MU characteristics and whether to model the behavior of the manipulators. A cooperative action (e.g., Compose) is specified to happen at a synchronization point in space.

An elementary action is the specification of a process onto a work-material (i.e., a transformation defined in the DIN taxonomy). Its contract-based specification aims to model the modifications applied to the MU. The level of abstraction used to represent the work-material depends on the action. For instance, an appropriate abstraction of the MU to model a subtractive (or additive) manufacturing process should rely on a 3-D grid. The grid abstracts through discretization the shape of the working-material. Meanwhile, in the contract representing the "Turn" action, the MU can be represented by its rotation only. Thus, the MU is specified as a 2-D projection of its 3-D shape.

For each machine, the proposed methodology creates a contract for each *elementary action* and a *coordinator* contract. Each action contract assumes the action's environment and, as such, describes the type of processing and desired shape, place, or orientation of the MU. The contract guarantees the component's behavior, by discretizing the movement of the machine and, therefore, the transformation of the MU. On the other hand, the coordinator contract assumes the whole set of elementary actions the machine is capable of. It also assumes additional properties to represent the shape and position of the product after the machine processing. The position specification is used to guarantee that the processed work-piece is placed in a specific cell of the 2-D grid spatial representation, as well as its orientation and its shape (in case of a composition or separation operation).

The scenario considered in the case study outlines the rotation action of the MU performed by the manipulator arm. Two contract-based specifications are created to represent such an action: The coordinator contract and the "Turn" action contract. Both contracts are depicted in the Table I. The manipulator coordinator contract assumes that the machine is always eventually available to perform the required action. Furthermore, a variable shared between components may be controlled by the environment in at least one of the contracts, creating a circular dependency. Thus, the environment may trivially control the variable to satisfy the assumptions even when the guarantees are falsified. This troublesome condition may be overcome by adding a liveness assumption that forcing the environment to avoid trivial assignments of variables. For example, the action completion (e.g., the assumption on `turn_executed` in the coordinator) is the liveness property solving such an issue for the coordinator contract. On the other hand, the contract guarantees that whenever an action has completed its execution (in this case, the "Turn" action), the machine becomes available in the next discrete-time step. Another safety guarantee is in place to model that whenever the "Pick" action is completed, the `grip` variable becomes `true`. Contrariwise, another safety guarantee states that whenever "Place" action is completed, the `grip` variable becomes `false`. The last safety guarantee property states that the "Turn" command is produced whenever the actual angle of rotation of the MU is not equal to the desired one and, at the same time, the gripper has the material in its claws.

The contract modeling the "Turn" action assumes that the manipulator has already completed a "Pick" action: A safety assumption requires that whenever the coordinator issues a "Turn" action, the gripper has the work material in its clamps. The action contract also assumes that the command is set to the constant "Turn" (i.e., the turn action is requested by the coordinator) infinitely often: This liveness property is necessary to manage the same circular dependency issue as the coordinator contract. The first safety guarantee of the action contract ensures that the `turning` variable is `true` if and only if the command received is "Turn". This variable represents the turning mode of the component and it is useful for defining the status of the system. In fact, the second guarantee assures that the MU is actually rotated of a certain angle (at discrete time-steps) while the turning status is active and while the actual angle is less than the desired angle. The last two safety guarantees model the finished action status, signaling that when the actual angle is

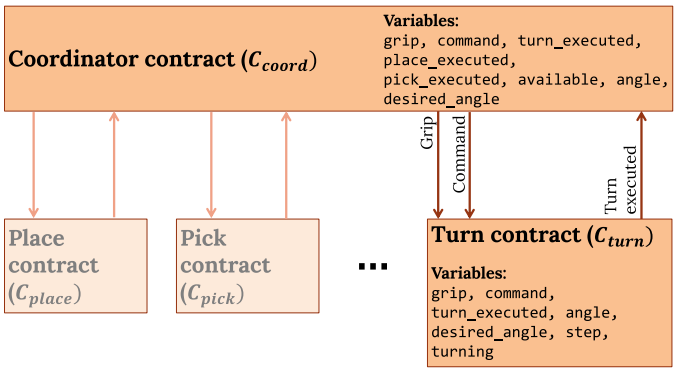

Fig. 5. Interactions between the coordinator contract and the turn action contract, implemented by a set of input/output variables.

equal to the desired one, the `turn_executed` variable become `true`.

Among the set of variables defined in each contract, a subset is related to the system while another subset is related to the environment. As such, system variables are controlled by the system and, therefore, can be viewed as the system's output or internal variables. On the other hand, environment variables act as input variables for the system. Fig. 5 represents the interactions between the coordinator contract and the turn contract, by outlining the input and output variables between each contract. In particular, the coordinator produces the command enabling the execution of an action. In addition, it provides the `grip` variable required for the initialization of the specific turn action. The "Turn" contract provides to the coordinator the `turn_executed` variable, that signals that the execution of such action is terminated.

Once completed the formalization of each action and machine, the contracts can be synthesized individually into its functional implementation.

V. Code Generation and Application

In this section, we show the application of the abovementioned formalization for the virtual prototyping of the production system. In particular, we show the steps necessary to generate the code implementing the contracts modeling the system, their interfacing with a commercial simulator of industrial production systems. Finally, we report the main results about the synthesis and code generation of the virtual prototype of the case study.

A. Executable Models Generation

The mealy machine synthesized by GR1C is represented by using a JSON format. We developed an automatic tool to generate the equivalent executable C++ code starting from each of the mealy machines synthesized from contracts. It translates the JSON model of each machine into an intermediate representation. Then, the tool generates a C++ class implementing the machine by exploiting automatic homogeneous code generation techniques [10]. This allows creating, for each A/G contract, an executable specification emulating the behavior of the component specified by the contract.

Listing 1: Sketch of the Simulate C-Interface Function that Performs a Property Abstraction Operation Over MU Physical Properties, Then it Calls the Simulate of the Synthesized Controller and Finally Back-Propagates Time, Power and MU Properties to the Simulator

```
1:  extern "C" __declspec(dllexport)
2:  void simulate (UF_Value* ret,  UF_Value*
    arg) {
3:  UF_Value &mu=arg[0]; UF_Value
    &station=arg[1];
4:  int operation=arg[2].value; //Pick,
    Place, ...
5:  double angle = MU_READ_ANGLE(mu);
6:  // ... other properties
7:  bool end;
8:  do {
9:  manipulator->simulate(angle,...);
10: end = manipulator->end;
11: } while(end);
12: ST_WRITE_TIME(station, calcTime
    (operation));
13: ST_WRITE_POWER(station, cal-
    cPower(operation));
14: MU_WRITE_ANGLE(mu,toAngle
    (manipulator->angle));
15: // ... other changed properies
16: }
```

The integration relies on the interface provided by the simulator. In this article, we focus on the C-Interface provided by Tecnomatix Plant Simulation, which allows loading custom shared libraries into the simulator, and SimTalk, the Plant Simulation internal scripting language. However, many other simulators provide conceptually similar interfaces [26].

The simulator provides a C interface that cannot call C++ methods and neither instantiate classes. Because of this limitation, `instantiate` and `deinstantiate` methods need to be implemented separately. Their purpose is to initialize and destroy the C++ class before and after the simulation execution. The `UF_Value` structure is defined in the `cwinfunc.h` header file and it is used to perform data exchange between the component implementation and the simulator environment. Each `UF_Value` contains two values: the data to exchange and its datatype. Each simulator datatype is mapped to the least bit-consuming C data-type providing enough bits: for example, `time` and `acceleration` are mapped to `double`, `string` to `char*` and so on. The C-Interface requires that each function expects an `UF_Value` to use as an argument and another one to bring its return value. To interact with the C++ class a `simulate` method is then added. It performs the following operations:

1) a *property abstraction* translates the MU physical features into a boolean representation (Listing 1, line 5);
2) the C++ controller is simulated with the provided inputs till the *end* output is raised indicating the operation completion (Listing 1, lines 8–11);

3) time and power consumption are calculated according to the simulation, and exported to the Plant Simulation *station* object (Listing 1, lines 12and 13);
4) the simulation results are converted into physical properties which are back-propagated to the MU (Listing 1, line 14).

Such a set of operations allows the product state to be kept consistent between machines, by storing it in the MU object.

Finally, the C++ strategy enriched with C-Interface headers and the previously defined functions are compiled into a shared library. The compiled file can be loaded by Plant Simulation.

B. Plant Simulation C-Interface Import

Listing 2 Sketch of the EntranceControl of the ManipulatorsStation Object in Plant Simulation (Written Using SimTalk Language).

```
1:  - load dll
2:  var fd := loadLibrary (".\printer.dll")
3:  if fd > 0 - check loading phase
4:  - simulate the controller
5:  callLibrary(fd, "simulate", @, ?)
6:  - unload dll
7:  freeLibrary(fd)
8:  else
9:  - stop the station
10: ?.Failed := true
```

Plant Simulation strongly relies on object-oriented concepts. Each simulator object is an instance of a class with its properties and methods. The import phase generates special *station* objects that uses the previously generated shared library functions to simulate their operation.

The `ManipulatorsStation` object is created by deriving the *Station* class and editing its `EntranceControl` method. The method written using the *SimTalk* language and it is automatically called by the simulator whenever an MU enters the station. Its purpose is to call the `simulate` function of the synthesized C++ controller. The code is sketched in Listing 2 and it relies on the *SimTalk* methods `loadLibrary`, `callLibrary`, and `freeLibrary`, which are the ones developed to handle external libraries. The *loadLibrary* method (Listing 2, line 2) opens the provided file path and returns its file descriptor, or a number lower than zero when an error occurs. Such an eventuality is checked in Listing 2 at line 3 and sets the station to a `Failed` state (in this state the object cannot process any MU), leading to the end of simulation. The `callLibrary` method calls a function into the provided library file descriptor by name. It is used to execute the external code (Listing 2, line 5). Two special arguments are provided.

1) @ represents the current MU;
2) ? indicates the *ManipulatorsStation*.

The `freeLibrary` method unloads a previously loaded file (Listing 2, line 7). The `ManipulatorsStation` object is finally saved and made available to the simulator libraries.

TABLE II
TIME REQUIRED TO PERFORM THE CONSISTENCY CHECKING AND SYNTHESIS STEP, AND THE CODE GENERATION FOR THE NONDECOMPOSED (I.E., HOLISTIC) SYSTEM SPECIFICATION, AND FOR THE DIFFERENT ACTIONS OF THE ROBOTIC ASSEMBLING STATION. THE NUMBERED COLUMNS REFER TO THE MACHINE'S ELEMENTARY ACTIONS: (1) COMPOSE, (2) DECOMPOSE, (3) PICK, (4) PLACE, (5) MOVE, (6) TURN. THE LAST COLUMN REPORTS THE TIME REQUIRED TO OBTAIN THE MACHINE COORDINATOR

	Holistic System	**Decomposed System**						
		(1)	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**Coord.**
Consistency Checking & Synthesis (s)	Time Out	0.20	0.20	0.19	0.17	0.13	0.11	0.24
Code generation (s)	-	0.12	0.12	0.08	0.08	0.07	0.06	0.15
Total Time (s)	-	**0.33**	**0.32**	**0.27**	**0.25**	**0.20**	**0.17**	**0.39**

C. Experimental Results

We build a tool-chain implementing the proposed methodology: the consistency checking and synthesis of the contracts is performed by using GR1C [27]. Meanwhile, we develop a tool receiving as input the JSON specifications produced by GR1C, and implementing the code generation described abovementioned. We applied the tool-chain to build a virtual prototype of the ICE laboratory case study. Then, we used the prototype to validate the system through simulation, and to evaluate the power consumption of the different machines involved in the production system.

Table II reports the time necessary to generate the virtual prototype of the collaborative robotic assembly machine. It reports distinctly the time required to perform the synthesis from contracts, that also incorporates the time required for the consistency checking, and the time required for the code generation. The *holistic system* column refers to the machine specified by a single A/G contract: it specifies all the operations of a single machine and their coordination. Therefore, it is specified using a different modeling approach. The same reactive synthesis tools and algorithms are used for both the decomposed and the nondecomposed scenarios. The *decomposed system* columns report the synthesis and code generation time required when using the decomposed system specification, as proposed in this article. The nondecomposed (i.e., holistic) system specification leads to complexity issues, as its consistency check and synthesis reaches the time-out we set to 6 h: this contract is characterized by a much higher number of LTL properties and, consequently, is harder to consistency check and synthesize. On the other hand, decomposing the system specification into multiple subproblems allows keeping the required time extremely limited.

Table III reports the results for all the machines in the production line. It is reported both the time required for the synthesis from the A/G contracts and the time required for the code generation. Each column refers to a machine. For instance, Column (4) refers to the robotic assembly station. As such, its values are the sum of the values in Table II. For all the machines, the processing time is minimal. The most time-consuming specification is the QC cell. This is due to the fact that the cell

TABLE III
TIME REQUIRED TO PERFORM THE CONSISTENCY CHECKING AND SYNTHESIS STEP FOR ALL THE MACHINES IN THE PRODUCTION LINE. MACHINES ARE IDENTIFIED BY THE FOLLOWING NUMBERS: (1) 3-D PRINTER, (2) CONVEYOR BELTS, (3) QUALITY CHECK, (4) ROBOTIC ASSEMBLY, (5) MILLING MACHINE. THE LAST COLUMN REPORTS THE TOTAL TIME REQUIRED FOR THE ENTIRE PRODUCTION LINE

	(1)	(2)	(3)	(4)	(5)	Total
Consistency Checking & Synthesis (s)	0.25	0.67	2.58	1.24	0.48	**5.22**
Code generation (s)	0.07	0.20	0.76	0.68	0.20	**1.91**
Total Time (s)	0.32	0.87	3.34	1.92	0.68	**6.39**

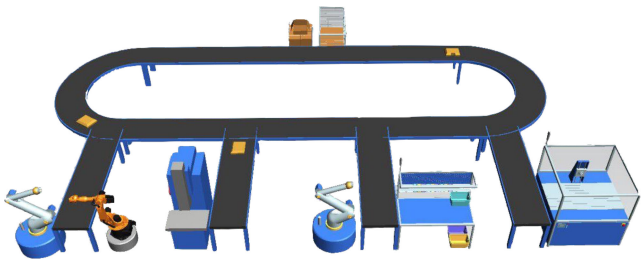

Fig. 6. ICE case study being simulated into Plant Simulation.

relies on cameras. Thus, its specification has to model multiple 2-D spaces to represent the signals analyzed by the cameras. The last column reports the total synthesis and code-generation time. It shows the efficiency of our methodology that allows generating virtual prototypes for production lines from their specifications.

The validation of a production line consists of verifying that the composition of each manufacturing step fulfills the production requirements, following the components specifications. The process validation is obtained by simulating the manufacturing plant agents with appropriate inputs and verifying that each component produces expected outputs. To perform such a simulation, we load into Plant Simulation the components generated by our methodology, and we build the simulation scenario, as depicted in Fig. 6.

VI. CONCLUSION

In this article, we presented a methodology to build the virtual prototype of a production line through the formalization of its specifications and automatic code generation. We showed the effectiveness of the proposed methodology by applying it to a real production line. We were able to generate its virtual prototype, and used the generated virtual prototype to analyzed the correctness of the line.

The experiments showed that the virtual prototypes for production machines were generated in a few seconds, after having formalized the production line specification as temporal contracts. This could be particularly hard for untrained designers, but it sensibly reduced its complexity starting from a library of predesigned contracts associated to each action of a taxonomy, like the DIN 8580.

REFERENCES

[1] R. Drath and A. Horch, "Industrie 4.0: Hit or Hype?," *IEEE Ind. Electron. Mag.*, vol. 8, no. 2, pp. 56–58, Jun. 2014.
[2] J. Vachalek *et al.*, "The digital twin of an industrial production line within the industry 4.0 concept," in *Proc. 21st Int. Conf. Proc. Control*. Jun. 2017, pp. 258–262.
[3] L. Hao *et al.*, "Enhancing the sustainability of additive manufacturing," in *Proc. 5th Int. Conf. Responsive Manufacturing—Green Manuf.*, Jan. 2010, pp. 390–395.
[4] A. Benveniste *et al.*, "Contracts for system design," *Found Trends Electron. Des. Autom.*, vol. 12, no. 2/3, pp. 124–400, 2018.
[5] *German Institute for Standards*, DIN Standard 8580, 2003. [Online]. Available: https://www.din.de/en
[6] R. Bloem *et al.*, "Synthesis of reactive (1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012.
[7] S. Spellini *et al.*, "Compositional design of multi-robot systems control software on ROS," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, p. 71, 2019.
[8] "Tecnomatix Plant Simulation," Siemens, Munich, Germany, 2017.
[9] D. Mourtzis *et al.*, "Simulation in manufacturing: Review and challenges," *Procedia Cyclic Inventory Routing Prob.*, vol. 25, pp. 213–229, 2014.
[10] M. Lora *et al.*, "Translation, abstraction and integration for effective smart system design," *IEEE Trans. Comput.*, vol. 68, no. 10, pp. 1525–1538, Oct. 2019.
[11] J. Banks *et al.*, *Discrete-Event System Simulation*. London, U.K.: Pearson, 2005.
[12] L. Dias *et al.*, "Discrete simulation software ranking—a top list of the worldwide most popular and used tools," in *Proc. Winter Simul. Conf.*, Dec. 2016, pp. 1060–1071.
[13] L. Büth *et al.*, "Introducing agent-based simulation of manufacturing systems to industrial discrete-event simulation tools," in *Proc. IEEE Int. Conf. Ind. Informat.*, Jul. 2017, pp. 1141–1146.
[14] "Simulation software survey," 2017. [Online]. Available: https://www.informs.org/ORMS-Today/OR-MS-Today-Software-Surveys/Simulation -Software-Survey
[15] M. V. Moreira and J.-J. Lesage, "Fault diagnosis based on identified discrete-event models," *Control Eng. Pract.*, vol. 91, 2019, Art. no. 104101.
[16] S. Preuße and H. Hanisch, "Verifying functional and non-functional properties of manufacturing control systems," in *Proc. 3rd IFAC Workshop Dependable Control Discrete*, 2011, pp. 41–46.
[17] O. Ljungkrantz *et al.*, "Formal specification and verification of industrial control logic components," *IEEE Trans. Autom. Sci. Eng.*, vol. 7, no. 3, pp. 538–548, Jul. 2010.
[18] R. Savolainen *et al.*, "A framework for runtime verification of industrial process control systems," in *Proc. IEEE 15th Int. Conf. Ind. Informat.*, 2017, pp. 687–694.
[19] A. Yacoub *et al.*, "A method for improving the verification and validation of systems by the combined use of simulation and formal methods," in *Proc. IEEE/ACM 18th Int. Symp. Distrib. Simul. Real Time Appli.*, 2014, pp. 155–162.
[20] P. Nuzzo *et al.*, "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems," *Proc. IEEE*, vol. 103, no. 11, pp. 2104–2132, Nov. 2015.
[21] P. Nuzzo *et al.*, "CHASE: Contract-based requirement engineering for cyber-physical system design," in *Proc. IEEE/ACM DATE*, 2018, pp. 839–844.
[22] R. Rosner "Modular synthesis of reactive systems," Ph.D. dissertation, Dept. Appl. Math. Comput. Sci., Weizmann Inst. Sci., Rehovot, Israel, 1991.
[23] I. Filippidis and R. M. Murray, "Symbolic construction of GR (1) contracts for synchronous systems with full information," in *Proc. Amer. Control Conf. (ACC)*, Boston, MA, USA, 2016, pp. 782–789, doi: 10.1109/ACC.2016.7525009.
[24] S. Maoz and J. O. Ringert, "GR (1) synthesis for LTL specification patterns," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 96–106.
[25] A. P. Sistla, "Safety, liveness and fairness in temporal logic," *Formal Aspects Comput.*, vol. 6, no. 5, pp. 495–511, 1994.
[26] M. Lora *et al.*, "Automatic integration of Cycle-accurate descriptions with continuous-time models for cyber-physical virtual platforms," in *Proc. IEEE/ACM DATE*, 2018, pp. 676–681.
[27] I. Filippidis *et al.*, "Control design for hybrid systems with TuLiP: The temporal logic planning toolbox," in *Proc. IEEE Conf. Control Appl.*, 2016, pp. 1030–1041.

Stefano Spellini (Student Member, IEEE) received the B.S. and M.E. degrees in computer science and engineering in 2016 and 2018, respectively, from the University of Verona, Verona, Italy, where he is currently working toward the Ph.D. degree in computer science with the Department of Computer Science.

He is a member of the Electronic System Design Research Group, working on methodologies for the design automation and verification of cyber–physical production systems.

Roberta Chirico received the B.Sc. degree in computer science and the M.E. degree in computer science and engineering from the University of Verona, Verona, Italy, in 2016 and 2019, respecively.

She is a Research Assistant with the ICE Laboratory of the Department of Computer Science, University of Verona. Her research focus on methodologies for the design of advanced manufacturing systems.

Marco Panato received the B.Sc. and M.E. degrees in computer science and engineering from the University of Verona, Verona, Italy, 2016 and 2018, respectively.

He is currently with the Department of Computer Science, the University of Verona. As a Research Assistant with the ICE Laboratory, he focuses on the design and the administration of smart manufacturing systems.

Michele Lora (Member, IEEE) received the Ph.D. degree in computer science from the University of Verona, Verona, Italy, in 2016.

He currently is a Marie Skłodowska Curie Fellow with the University of Verona. His research focuses on design automation methodologies for modeling, integration and efficient simulation of heterogeneous embedded systems, contract-based requirement engineering, and verification for cyber–physical systems.

Franco Fummi (Member, IEEE) received the Ph.D. degree in electronic engineering from Politecnico di Milano, Milano, Italy, in 1995.

Since 2000, he is a Full Professor with the Department of Computer Science, University of Verona, Italy, and where he became an Associate Professor in computer architecture in 1998. Since 1995, he has been with the Department of Electronics and Information, Politecnico di Milano, as an Assistant Professor. He is a Co-Founder of EDALab, an EDA company developing tools for the design of networked embedded systems. His current research interests include electronic design automation methodologies for modeling, verification, testing, and optimization of cyber–physical production systems.