Integrating Smart Contracts in Manufacturing for Automated Assessment of Production Quality

Sebastiano Gaiardelli, Stefano Spellini, Michele Pasqua, Mariano Ceccato, Franco Fummi
University of Verona
[name].[surname]@univr.it

***Abstract*—Products and materials traceability is essential in modern manufacturing, where the production must meet certain standards that range from Quality Control (QC) to the quality of the used materials. In this environment, blockchain applications allow certifying data provenience and subsequent modification, offering trust and security along the entire supply chain. Nonetheless, the design and the development of such applications are usually performed manually and, thus, subject to errors.**

In this paper, we propose a methodology allowing to automatically generate smart contracts starting from a SysML model. This approach allows easing the integration of blockchain applications in a production system: by abstracting the implementations with models, it is possible to generate smart contracts for different blockchains, connecting to multiple production environments.

We applied the proposed methodology on a real manufacturing system, assessing the quality of a case-study production.

***Index Terms*—Smart contracts, System modeling language, Blockchains**

I. Introduction

Industry 4.0 has been proposed to respond to the shifting of the traditional market trend based on mass-production towards a more customization-oriented manufacturing. Such a paradigm shift typically involves bigger and more complex production chains networks. The complete traceability of materials and products along the entire chain is not guaranteed, especially considering the availability of distributed infrastructures to coordinate multiple facilities [1]. In this context, blockchain applications allow certifying data and their elaboration on a transparent, secure, and distributed infrastructure. Therefore, the manufacturing community has started using blockchain applications, to cope specifically with the trust problem among different participants [2]. Such a class of applications are specified through specific "programs" interacting with the blockchain: the *smart contracts*. The integration of production Internet of Things (IoT) data and smart contracts enables monitoring the consumption of raw materials and tools and certifying the quality [3].

Designing blockchain-based applications is a process that requires to manually specify the interactions between the software (*i.e.*, smart contracts) and the system to monitor. Especially regarding manufacturing, it requires to define the structure of the system, the data and the implemented processes. Therefore, a methodology assisting the creation of "manufacturing smart contracts" would be a step forward in supporting the traceability and certification various production aspects. A promising path in such a direction is to exploit Model-based Design (MBD) principles, an approach that has already been explored in the context of manufacturing: it has been proposed to automatically generate code for smart contracts [4], to ease the development and testing of heterogeneous Cyber-Physical Production Systems (CPPSs) [5] and to support the scheduling of orders [6].

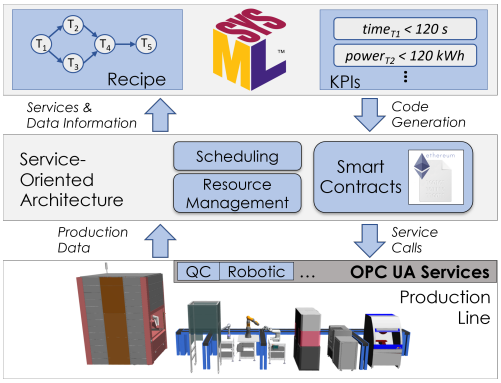

Figure 1: Overview of the proposed approach. The production recipe and the target KPIs are modeled using SysML. Such models are translated in a set of smart Contracts using a custom-built automatic tool. Each contract, integrated in a reference service-oriented manufacturing system, contributes to define and certify production quality.

In this context, this paper proposes an automated approach to generate smart contracts for assessing production quality. As depicted in Fig. 1, the proposed methodology exploits System Modeling Language (SysML), which is expressive enough to represent both the production recipe (*i.e.*, the sequence of production tasks) and the properties being verified (*e.g.*, KPIs). Nonetheless, to support the modeling phase and to consistently generate smart contracts code, this work proposes a specific SysML meta-model (*i.e.*, profile): it focuses on defining the production recipe with a behavioral diagram and the associated KPIs to be verified. Once the recipe and the KPIs are modeled, an automatic tool generates the set of smart contracts to be deployed on the blockchain.

We demonstrate the applicability of such a methodology by modeling, generating and integrating the smart contracts assessing the production quality of a real manufacturing system. We evaluate a set of KPIs on different recipes, estimating the complexity of the resulting smart contracts and, thus, assessing the scalability of the proposed approach.

II. Background

We hereby report a set of background concepts useful to understand the proposed approach. In particular, we introduce SysML and smart contracts. We also detail the reference manufacturing system on which we implement the methodology.

A. SysML

SysML is a software and systems model engineering language. It has been developed as a "dialect" of Unified Modeling Language (UML), which is traditionally limited to representing software components. Therefore, SysML supports the engineering of systems and systems-of-systems, by proposing language constructs (*i.e.*, diagrams) capable of representing mechanical components, physical properties, software functionalities and information models. SysML introduces additional diagrams over UML, which aim at being more flexible and expressive in order to specify physical systems. In particular, SysML allows to: (1) define system requirements through *Requirements diagrams*; (2) represent the architecture of the system and sub-systems through *Block diagrams*; (3) model the behavior of components and their interaction with *Behavioral diagrams* (*e.g.*, *Sequence*, *State Machine* and *Activity diagrams*); (4) design the dynamic of the system and carry out physical analysis using *Parametric diagrams* and *Constraint Blocks*.

B. Smart Contracts

Ethereum is an open-source platform for decentralized applications, based on the *blockchain* technology. On the Ethereum network, it is possible to write programs, called *smart contracts* [7], that (semi-)automatically manage the underlying network cryptocurrency, called *Ether* (ETH). The actions that can be performed in Ethereum are basically transactions, *i.e.* transfer of funds or data between different ETH accounts. Every new transaction is irreversible and it permanently added in a new *block* that updates the blockchain [7].

In the Ethereum network each principle has an account identified by an *address*. An account address can emit and receive transactions (similarly to Bitcoin wallets) or it can be an identifier of a smart contract deployed in the network, which is run whenever a transaction is sent to the address [7]. Every operation in the network has a cost (called *Gas*) expressed in Wei, a fraction of Ether.

Smart contracts for Ethereum can be written with different high-level programming languages, but *Solidity* [8] is indubitably the most wide-spread. It is a Turing-complete object-oriented language, and smart contracts are basically objects with methods and fields. In order to actually run a smart contract on the Ethereum blockchain, the Solidity source code needs to be compiled into *EVM bytecode*, in order to be executed by the Ethereum Virtual Machine.

C. Reference Manufacturing System Architecture

The manufacturing system used as reference is a full-fledged production line available at our research facility [9]. Such a system implements additive and subtractive manufacturing, robotic manipulation and Quality Control (QC) processes. It also includes an automatic vertical warehouse, to store materials and products. The machines composing the plant are physically connected through a set of pallet-based conveyors.

The software stack controlling the production line is built around the concept of Service Oriented Manufacturing (SOM) [10]: each machine functionality is defined by set of production services, implemented at Programmable Logic Controller (PLC) level and exposed to higher software levels (*e.g.*, to the Manufacturing Execution System (MES)). The Machine to Machine (M2M) communication between automation components is implemented through the OPC Unified Architecture (OPC UA) protocol [11]: it consists in a client/server structure, where the information model is embedded and exposed by the server. On top of the OPC UA, the system includes *Apache Kafka* and *RabbitMQ* message brokers, to handle low latency real-time data and to send commands to machines (*i.e.*, service calls). The top layers of the architecture are occupied by a commercial MES and a set of software modules: the *Automation Manager* [12]. On one hand, the MES is dedicated to defining production concepts (*e.g.*, *recipes*, *resources*, *etc.*) and production lifecycles. The *Automation Manager*, on the other hand, has been developed to exploit such concepts and facilitate the reconfiguration of the system. Such a software architecture serves as a platform for the deployment and the execution of the smart contracts.

III. Proposed Approach

A. Modeling Framework

To guide the designer in representing production recipes and attached KPIs, this work proposes both a modeling strategy and a meta-model (*i.e.* profile): the former is focused on providing a guideline on how to represent a production recipe, while the latter constrains the designer in defining KPIs, with parameters and the logical/mathematical property to evaluate.

Regarding the production recipe, the proposed modeling strategy involves creating a SysML *Activity Diagram*, to structure the tasks and their dependencies. Such a diagram is also capable of describing the sequential and parallel execution of tasks using the fork-join principle. Therefore, syntactic elements are available to define forks from a node (*i.e.*, a task) and joins, where tasks execution are split into parallel flows or joined into a finite sequence. Regarding the KPIs, the meta-model serves to give a precise structure to Block Definition Diagrams (BDDs). A KPI can be associated to a specific task or to the recipe itself. The modeling strategy differentiates the two types by establishing that a recipe KPI must be inserted within the recipe *package*, while a task KPI is encapsulated in the task *package*. Three constitutive elements must be present in the model, to be consistent with the presented profile: (1) the *inputs* and the *parameters* used for the evaluation of the KPI, (2) a *sub-diagram* (*e.g.*, an *Activity Diagram*), to specify the mathematical formula, (3) the *relations* between input, parameters, formulas and the KPI itself. For the first point, the profile provides a set of block *stereotypes*:

- the `ConfigurationParameters`, outlining the values coming from the production line, which are input(s) to the specific KPI;
- the `UserDefinedParameters`, to characterize numerical bounds or, in general, values constraints over the input parameters;
- the `KPI`, which is the central block, connected to the parameters blocks and to the sub-diagram determining the KPI formula.

The KPI block, in particular, also specifies the `Repeatable` property, to specify whether the KPI has to be evaluated on a

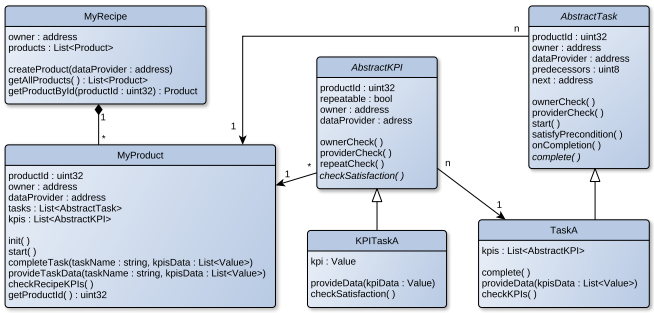

Figure 2: A UML *Class Diagram* that represents a Solidity smart contracts recipe.

set of data or on a single input value. For the sub-diagram specifying the mathematical formula, we chose to rely on a behavioral diagram; to define the evaluation steps, each block (*i.e.*, action) contains a textual mathematical expression composed by values, parameters identifiers and operators (*e.g.*, LT for less-than, GE, for greater-or-equal, *etc.*). Finally, the meta-model defines a set of labeled relations between stereotyped blocks. Such relations are constrained between the types of blocks. As an example, a KPI *GetsValuesFrom* a `UserDefinedParameter` block. Depending on the tool, the meta-model guides the designer constraining the SysML block-relation-block possibilities to the set defined in it. Therefore, the proposed profile is a fundamental methodological contribution to modeling and, consequently, to the automatic generation of smart contracts.

B. Smart Contract Infrastructure

Given a SysML recipe, we automatically generate a set of smart contracts that record on the blockchain the data collected from the production line. Furthermore, smart contracts automatically compute product KPIs and certify production quality. To mitigate the high transaction fees actually affecting the Ethereum main network (Mainnet), we deploy our smart contracts on a *side chain*, *i.e.* a separate blockchain connected to the Mainnet by a two-way bridge. Side chains offers almost the same security guarantees as the Mainent, but they have very low gas consumption (and, hence, low transaction fees). Side chains are fully compatible with Ethereum, hence we generate smart contracts written in Solidity, the most used programming language for the Ethereum network.

In particular, given a SysML recipe, our tool will automatically generate a set of Solidity source code files, following the Class Diagram depicted in Fig. 2, where each class translates to a Solidity (`.sol`) file. Then, the source code will be compiled with the Solidity compiler and the resulting EVM bytecode is deployed on the side chain by means of an *Automation Manager* plugin. The latter is attached to the production line, as we will explain in the next subsection III-C.

A SysML recipe yields a *recipe* smart contract (class MyRecipe in Fig. 2). The latter, deploys on-demand a new instance of the recipe (*i.e.* a smart contract) that corresponds to a product (by means of the method createProduct(...)). Each product is assigned a unique identifier by the recipe contract and it is parameterized by an account address, demanded to provide production data. The *product* smart contract (class MyProduct in Fig. 2) exposes some methods in order to record the data coming from the production line (provideTaskData(...)). Depending on the data provided, the product KPIs may or may not be satisfied, and the behavior of the particular recipe instance is consequently affected (*i.e.* tasks may or may not be activated).

SysML recipe tasks translate to standalone smart contracts (class like TaskA in Fig. 2), that are automatically instantiated by the product contract. The causality between tasks is retrieved from the SysML recipe Activity Diagram and encoded into smart contracts as follows. Each task contract has a counter (field predecessors in the abstract class AbstratTask of Fig. 2), that holds the number of predecessor tasks that must be completed in order to activate the current task. Furthermore, each task has a pointer (field next in the abstract class AbstratTask) to its successor task, which is unique. When a task terminates, it sends a completion notification (by means of the onCompletion() method) to its successor task. The latter decrements its predecessors counter (by means of the satisfyPrecondition() method) and if no predecessors are still executing, the task can be activated (the method start() is called). In Solidity there are no threads, so we have no race condition problem on the predecessors counter.

A task is *active* when it can receive data from the production line, by means of public methods, that can be called only when the task is in the active state. The data provided to a task are recorded on the blockchain and they are immutable. Once all data are sent to a task, a *completion* method (completeTask(...) of MyProduct class) is called, checking the task KPIs before notifying its successor task. The life-cycle of a product ends when all its tasks are inactive (*i.e.* they are completed).

Recipe and task KPIs are standalone smart contracts (class like KPITaskA in Fig. 2) that are attached to a product or a task contract, respectively. The only difference is that the first are not repeatable by default, while the latter may be set repeatable. This means, in particular, that recipe KPIs are provided with data from the production line only when a task is completed, while task KPIs can continuously receive data.

Finally, access control is performed by means of methods modifiers and the product contract private fields owner and dataProvider. Indeed, each public method exposed by the smart contracts has either the modifier owner, checking that only the address of the account that instantiated the product can exercise such method, or the modifier dataProvider, checking that only the address of the account that the owner has delegate to provide production data can exercise such method. In particular, only owner has permission to complete tasks and only dataProvider has permission to feed task/KPI contracts with data coming from the production line.

C. Automated Code Generation and Data Stream Integration

The Solidity source code is automatically generated by our tool, starting from a SysML recipe. First, we inspect the recipe Activity Diagram to identify the tasks composing the recipe: for each block of the diagram we generate a corresponding task smart contract. The control flow of the activities is encoded as

explained in the previous subsection, setting the predecessors counter and the next pointer for each task contract.

Then, for each KPI linked to a given task we generate a KPI smart contract, translating its BDD. In particular, `UserDefinedParameters` translates to constant variables of the contract and for each `ConfigurationParameters` we define a setter method provideData(...), that it is called from the product contract to provide data for the KPI. This method is equipped with a modifier checking whether the KPI is repeatable or not. Finally, the KPI formula is translated to a Solidity boolean expression, that is checked when calling the KPI contract method checkSatisfaction(). For each task we instantiate a list of KPI contracts.

Now, the product smart contract can be generated, instantiating all tasks, activating the one that starts at the beginning of the recipe and generating all public methods needed to provide data to KPIs and completing tasks. In case of recipe KPIs, we instantiate a list of KPIs also for the product contract (the KPIs translation is the same as the one for task KPIs).

Then, the smart contracts are compiled with the Solidity compiler and imported inside the application developed on top of *Automation Manager*. *Automation Manager* is a software architecture that allows extending the functionalities of commercial MES, also automating the scheduling and execution of tasks on the production line. Furthermore, it allows to develop new independent applications in a more abstract way, offering different functionalities such as events carrying information about what the system and the production plant are doing. Therefore, it allows to uniquely associate IoT data coming from the machines to the correct task and KPIs. The developed application extends also the representation of production recipes, associating for each task the definition of the KPIs inside the corresponding smart contracts. Then, each time a work order is released inside the *Automation Manager* the application create and deploys the corresponding smart contracts on the blockchain. After that, the application listens to the events generated from the lower infrastructure and acts accordingly. When a task starts its execution on a machine the application call the method start() to activate the smart contract inside the blockchain. Furthermore, until the task completes its execution each IoT data coming that is recived from the machine are also notified to the KPIs associated with the method provideData(...). Lastly, when the task is completed the method complete() is called to compute and lock the last KPIs on the blockchain.

IV. Experimental Validation

A. Case Study

We present a case study production applied to the ICE Laboratory, to assess the approach proposed by this paper. The production line is used to construct a small LEGO®-like toy: the system must 3D print, assemble and qQC the object through a specific sequence of actions (*i.e.*, recipe). In particular, the toy is composed by three pieces of different colors and shapes. Two of the three pieces, the *green* one and the *white* one, are collected from the warehouse and transported to the robotic assembly cell. Meanwhile, the third piece (the *red* one) must be 3D printed. Then, the plant must quality check both the assemble of the first two pieces and the correct shape of the printed brick. Finally, all the pieces must be assembled together and checked for defects. If the quality of the product is consistent, the final toy is then stored in the warehouse. To simplify the presentation, the logistic aspects (*i.e.*, transportation) of the case-study production are left out. We concentrate on tasks implementing an actual material transformation, with the only exception of material retrieval from the warehouse.

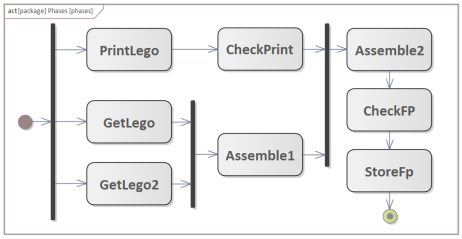

Figure 3: SysML *Activity Diagram* representing the case-study production recipe. It is composed of a set of tasks and their dependencies.

The execution of the tasks composing the recipe generates production data, consisting of processing times, power consumption, failures and quality check results. To assess the overall quality of the production, we identified a set of constraints over such data. The constraints are represented as a set of KPIs. The case-study must be compliant to:

- *processing time KPIs*, defining whether a task starts and completes within a certain time-span;
- *power consumption KPIs*, to assess the energy costs of producing the product following the established recipe;
- *accuracy and failures KPIs*, to check the accuracy of intermediate manipulation steps and errors; this data may indicate materials deterioration or semi-finished products.

In particular, processing time and power consumption KPIs can be associated to all tasks composing the production recipe (*e.g.* material retrieval, QC, additive processes, *etc.*) and to the recipe itself. On the other hand, accuracy KPIs are correlated to assembly and 3D printing. Moreover, the printed plastic piece may result in melted parts or scorches, that must be detected and handled correctly.

B. Modeling

The first step to validate and evaluate the methodology proposed is to model the case-study production. Such a process has been carried out using SysML and the meta-model described in Section III-A. The model has to encompass fundamental information about the production recipe and the quality properties that a stakeholder is interested in verifying and certifying. In particular, the model structures the knowledge on three fundamental pillars:

- the *production recipe*, in terms of task, dependencies between tasks and their sequential/parallel execution;
- the *KPIs* associated to a task or to the recipe itself;
- the *relations* between KPIs, tasks and recipe, and how their evaluation is carried out.

Fig. 3 depicts the case-study production recipe modeled through an *Activity Diagram* in SysML: it is composed of the

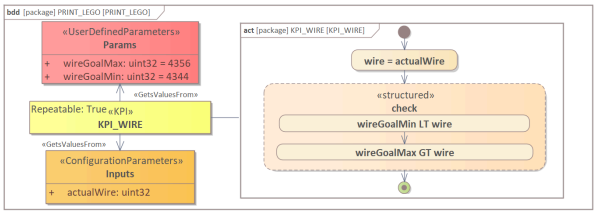

Figure 4: The BDD defining a KPI regarding the "PrintLego" task. It verifies that the plastic filament consumption of the 3D printer stays within a specific range (in millimeters).

necessary sequence of tasks to produce LEGO®-like toy. More specifically, tasks "PrintLego", "GetLego" and "GetLego2" are started in parallel at the beginning of the recipe. Therefore, the production line starts printing the *red* piece, while the other two bricks are collected from the warehouse and placed on two pallets. The dependencies between tasks simply define the sequence of the necessary operations and do not represent timing constraints. Thus, the "Assemble1" task execution can be fired once both the *green* and the *white* pieces are loaded on the main conveyor. Such a task transports the two pallets to the robotic cell and assembles those pieces using manipulators arms. To represent parallel execution, SysML uses a specific syntactic constructs to model fork-join: the black vertical bars in Fig. 3, which branches off the execution of the recipe in parallel (*e.g.*, "PrintLego", "GetLego" and "GetLego2" tasks) and merges tasks to resume sequential execution (*e.g.*, "GetLego" and "GetLego2" to "Assemble1").

Each task of the recipe is paired with a set of KPIs. Fig. 4 depicts an example of a BDD representing a KPI for the "PrintLego" task. Such a construct is in charge of checking that the length plastic filament consumed (*i.e.*, *wire* in the diagram) is within specific lower and upper boundaries. The definition of the diagram concerning the KPI is carried out referencing to the meta-model described in Section III-A. In particular, the BDD includes a `uint32` input value for the consumed filament length. It is contained in a block stereotyped by `ConfigurationParameter` (orange in the figure). In addition, the diagram defines the upper and lower filament consumption bounds in a `UserDefinedParameter` block (red in the figure). Furthermore, the evaluation of the KPI block (yellow in the figure) is on data-set instead of a single value. This is specified by setting the `Repeatable` block property to True. As per the meta-model, the KPI is connected with other the stereotyped blocks via the *GetsValuesFrom* relation. Finally, the mathematical formula is defined in an Activity Diagram, connected to the KPI block. In such a diagram, the local variable `wire` is defined. This is done by textually defining the assignment expression within an *Action* block. Then, the KPI mathematical formula is characterized in a *Structured Activity* node within the diagram. Such a construct allows defining multiple text-based expressions within multiple *Action* blocks. Each block is in a sequential order, to specify their order of evaluation. The mathematical formula is simple: it verifies that the `wire` value is between `wireGoalMin` and `wireGoalMax` parameters. Therefore, the *Structured Activity* consists of two sequential expressions:

```
1  contract LegoRecipe {
2    LegoProduct[] private products;
3    uint32 private productId; address private owner;
4    ...
5    function createProduct(address dataProvider) public {
6      require(msg.sender == owner);
7      LegoProduct product = new LegoProduct(productId, dataProvider);
8      products.push(product);
9      productId ++;
10   }
11 }
```

Listing 1: Solidity code snippet for the LEGO® recipe contract.

```
1  contract KPIWire is AbstractKPI {
2    uint32 private constant wireGoalMax = 4356, wireGoalMin = 4344;
3    uint32 private wire;
4    ...
5    function provideDataTask(uint32 actualWire) public validDataProvider
           isRepeatible incrementAfter {
6      wire = actualWire;
7    }
8    function checkSatisfaction() override public view validOwner returns (bool) {
9      return wireGoalMin < wire && wireGoalMax > wire;
10   }
11 }
```

Listing 2: Solidity code snippet for the *wire* KPI contract.

the first verifying that `wireGoalMin` is less than `wire`; the second checking that `wireGoalMax` is greater than `wire`.

C. Smart Contract

The SysML recipe for the LEGO®-like toy is translated into the `LegoRecipe.sol` Solidity source code file; an excerpt of this file is reported in Listing 1. In particular, this contract maintains a list of "child" smart contracts, one for each production object. Indeed, when the production line starts the fabrication of a product, the method `createProduct(...)` is called and a new instance of the `LegoProduct` contract is created (parameterized with an unique identifier `productId` automatically generated). At line **10** of Listing 1 we check that only the recipe owner (i.e., the party that deployed the recipe contract) can create a product. During creation, the `LegoProduct` constructor parameter `dataProvider` sets the address of the party that is authorized to provide data to the KPIs of a particular product.

The `LegoProduct.sol` Solidity source file contains a list of all tasks specified in the activity diagram of Fig. 3 (and a list of recipe KPIs). When the contract is instantiated, the tasks list is created, instantiating a smart contract for each task. Tasks are initialized as explained in Section III: a predecessors count and a successor task are set. For instance, consider the code snippet of the `LegoProduct.sol` file in Listing 3. We can see that `CheckPrint` and `Assemble1` contracts have both `Assemble2` as successor (the fourth parameter of the constructor); and `Assemble2` has indeed 2 as predecessors count (the third parameter of the constructor). This is the translation of the parallel flow between "CheckPrint", "Assemble1" and "Assemble2" as depicted in the activity diagram of Fig. 3. Sequential flows are simpler: `Assemble2` has `CheckFP` as successor and `CheckFP` has 1 as predecessors count.

Finally, in Listing 2 we have an excerpt of *wire* KPI smart contract, translation of the BDD in Fig. 4. As we can see, KPI constraints are translated to constant variables (lines **5-6** of Listing 2) while the KPI input is translated to the public method `provideTaskData(...)` that assigns the (private) variable `wire` the value provided by the method parameter `actualWire`. The KPI condition is translated to a boolean expression (line **15**), encapsulated in the pub-

```
AbstractTask[tasksNumber] private tasks;
...
tasks[Tasks.CheckFP] =
  new CheckFP(productId, dataProvider, 1, addrStoreFP);
tasks[Tasks.Assemble2] =
  new Assemble2(productId, dataProvider, 2, addrCheckFP);
tasks[Tasks.CheckPrint] =
  new CheckPrint(productId, dataProvider, 1, addrAssemble2);
tasks[Tasks.Assemble1] =
  new Assemble1(productId, dataProvider, 2, addrAssemble2);
```

Listing 3: Solidity code snippet for the LEGO® product contract.

Table I: Generated bytecode size at different recipe complexity levels.

Recipe	Recipe KPIs	Tasks KPIs	Tasks	EVM bytecode
r8	2	6	8	85 822 byte
r16	4	12	16	160 415 byte
r32	8	24	32	309 601 byte

lic method `checkSatisfaction(...)`. The latter will be called by the corresponding task, to check the KPI satisfaction. Note that, the modifier `validDataProvider` ensures that only the authorized party can provide KPI data. The check assessing the non repeatability of a KPI is demanded to the `isRepeatible` modifier.

D. Automatic generation complexity

An important aspect to consider when dealing with smart contracts is the bytecode *size*, namely the space needed to store the compiled source code on the blockchain. Each blockchain transaction has a cost, proportional to the dimension of the transaction. Hence, the bigger is the smart contract and the higher is the cost for its deployment. As mentioned in Subsection III-B, we mitigate this problem by using side chains instead of the Ethereum Mainnet. Nevertheless, even in side chains transactions have a cost, hence minimizing the size of the compiled smart contracts is still important.

The structure of the smart contracts we adopt to model recipes allows us to generate "lightweight" contracts: the generated bytecode size is proportional to number of tasks/KPIs of a recipe. Indeed, most of the code is encapsulated in the abstract tasks/KPIs contracts, libraries and the recipe contract, which are not strictly dependent on the recipe business logic. To test our automatic approach, we modeled three recipes at different levels of complexity: *r8*, equipped with 2 recipe KPIs and 6 KPIs shared by 8 tasks (8 KPIs and 8 tasks, in total); *r16*, equipped with 4 recipe KPIs and 12 KPIs shared by 16 tasks (16 KPIs and 16 tasks, in total); and *r32*, equipped with 8 recipe KPIs and 24 KPIs shared by 31 tasks (32 KPIs and 32 tasks, in total). For each recipe we have generated the Solidity smart contracts and compiled them with one of the latest version of the Solidity compiler[1], without optimizations. The results are shown in Table I. As we can see, doubling the complexity of the recipe (in the number of tasks/KPIs) results in a (little less) double size of the generated bytecode.

V. Conclusion and Considerations

In this paper we have introduced an innovative methodology for monitoring and assessing the quality of a production line. Starting from a SysML diagram, serving as an *abstract* representation of a production recipe, we automatically generate a set of smart contracts, serving as a *concrete* representation of the production recipe, and we deploy them on the Ethereum blockchain. A module extending *Automation Manager*, in charge of collecting production data and forwarding them to the smart contracts, is automatically generated as well. Due to the immutability of the blockchain, the provided data are inherently certified and they are used by the smart contracts to compute recipe KPIs, assessing production quality.

Having a blockchain-based *quality certification* attached to the produced objects increases the market value of the products. Indeed, end users and/or stakeholder can interact with the smart contracts to check whether a given product adheres with the original recipe and whether the product fulfills particular quality requirements. Unfortunately, writing smart contracts is a complex and error-prone task. To mitigate the problem, our approach dispenses developers from blockchain-related implementation details, since the smart contracts are generated from the SysML recipe without any manual intervention. Furthermore, the generated smart contracts dimension (in bytecode) scales linearly with the recipe complexity; this is crucial to deploy efficient blockchain-based solutions.

Smart contracts are not free of programming defects, and bugs in smart contracts very often result in a leak of funds. For this reasons, an automated generation approach may help in lowering the number of vulnerabilities present in the deployed smart contracts. In this respect, we may integrate the generation process with *secure compilation* techniques, in order to synthesize secure-by-construction smart contracts.

Finally, we can imagine to exploit the underling smart contracts *crypto-currency* to automatically fire payments, when quality requirements are met by the produced objects.

[1] `solc` version `0.8.0+commit.c7dfd78e.Linux.g++`.

References

[1] L. Ouyang, Y. Yuan, and F.-Y. Wang, "A blockchain-based framework for collaborative production in distributed and social manufacturing," in *2019 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, 2019, pp. 76–81.

[2] S. Geiger, D. Schall, s. meixner, and a. egger, "Process traceability in distributed manufacturing using blockchains," 04 2019.

[3] W. Alkhader, N. Alkaabi, K. Salah, R. Jayaraman, J. Arshad, and M. Omar, "Blockchain-based traceability and management for additive manufacturing," *IEEE Access*, vol. 8, pp. 188 363–188 377, 2020.

[4] M. Jurgelaitis, L. čeponienė, and R. Butkienė, "Solidity code generation from uml state machines in model-driven smart contract development," *IEEE Access*, vol. 10, pp. 33 465–33 481, 2022.

[5] A. Barišić, E. Zhu, and F. Mallet, "Model-driven approach for the design of multi-chain smart contracts," in *2021 3rd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*, 2021, pp. 37–38.

[6] S. Gaiardelli, S. Spellini, M. Lora, and F. Fummi, "A hierarchical modeling approach to improve scheduling of manufacturing processes," in *2022 IEEE 31th International Symposium on Industrial Electronics (ISIE)*, 2022, pp. 1–8.

[7] A. Antonopoulos, G. Wood, and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018. [Online]. Available: https://books.google.it/books?id=SedSMQAACAAJ

[8] Ethereum. Sidechains. [accessed: 2022-04-21]. [Online]. Available: https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/sidechains/

[9] "Industrial Computer Engineering (ICE) Lab," https://www.icelab.di.univr.it/.

[10] T. Lojka, M. Bundzel, and I. Zolotová, "Service-oriented architecture and cloud manufacturing," *Acta polytechnica hungarica*, vol. 13, no. 6, pp. 25–44, 2016.

[11] "OPC Unified Architecture specification – Part 1: Overview and concepts release 1.04 OPC Foundation," 2017.

[12] S. Gaiardelli, S. Spellini, M. Panato, M. Lora, and F. Fummi, "A software architecture to control service-oriented manufacturing systems," 2022, pp. 1–4.