Formal analysis of production line systems by probabilistic model checking tools

Formal analysis of production line systems by probabilistic model checking tools

Paolo Ballarini
CentraleSupélec, Université Paris Saclay, France
paolo.ballarini@centralesupelec.fr

András Horváth
Dipartimento di Informatica, Università di Torino, Italy
horvath@di.unito.it

***Abstract*—Performance modelling is a fundamental part of the design and operation of reliable manufacturing systems. Based on the probabilistic model checking formalism [11] supported by the PRISM tool [13], in this paper we present a framework for automatic generation of 1) formally expressed discrete-state Markov chain (DTMC) models of production lines, and 2) of a number of related key performance indicators given in terms of temporal logic formulae. Since the framework is fully parametric it can straightforwardly be used for comparative analysis of different system configurations. In order to tackle scalability issues we present two alternative encodings of the DTMC model corresponding to a given production system and discuss how they compare. We demonstrate the effectiveness of the framework through a number of experiments.**

***Index Terms*—Manufacturing systems, Model checking, Production performance evaluation**

I. Introduction

The design of modern industrial production systems is strongly affected by product quality and delivery reliability requirements. The ability to guarantee that products are issued within given time deadlines and that they match given quality standards are essential factors throughout the design and maintenance of a manufacturing system.

In this paper we consider production line models composed of buffers and unreliable machines with Markov behavior. Building on previous works dealing with the same or similar models (see, e.g., [1], [2], [7]–[9]), in this paper we introduce an integrated framework for formal modeling and performance analysis of such systems. Our proposal is to describe the system with a well-known and widely applied model checking tool called PRISM [13] which allows for specifying the system in terms of a DTMC, expressing and calculating performance indices, and verifying complex temporal properties of the model [11]. The presentation in this paper is limited to two-state machines and linear production lines without branching or loops but extensions to overcome these assumptions are straightforward in the same framework. For what concerns the description of the model in PRISM, we implement two approaches. The first approach builds the DTMC corresponding exactly to the model but suffers from scalability problems and cannot be applied with more than 5 machines. The second one instead scales well with the number of machines in the system but results in a modified model with a slightly enlarged state space whose behavior however is easy to map to that of the original model.

The paper is organized as follows. Section II provides the description of the considered systems. In Section III a brief introduction to probabilistic model checking is given. Section IV shows how to deal with the considered type of production lines is PRISM. In Section V we experiment with the proposed ideas. Conclusions are drawn in Section VI.

II. System description

A linear production system (or production line for short) is a type of manufacturing system in which parts visit a number of workstations (called also machines) in a fixed order (see Figure 1) and following a single path (no branching). Machines process one part per time unit. In many industries there are production processes that can be described by such a simple model, for example food industry, automotive industry, paper industry, or semiconductor manufacturing.

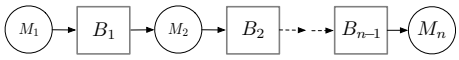

Fig. 1. A production line with n machines.

In this paper we assume two-state machines: a machine can be either up (or UP or U for short) or broken (or $DOWN$ or D for short). The state of a machine changes in a random fashion. In each time unit (called also slot) an operational machine can break with probability p_i, $1 \leq i \leq n$, where n is the number of machines. A broken down machine gets instead repaired with probability r_i, $1 \leq i \leq n$. Standalone availability of a given machine M_i, i.e., the probability that M_i is up assuming that it is not affected by the rest of the system, can be calculated simply as $r_i/(r_i + p_i)$.

Between two adjacent machines there is a buffer where parts can be stored. Accordingly, the number of buffers is $n - 1$. The capacity of the ith buffer, i.e., the number of parts it can hold, is denoted by n_i with $1 \leq i \leq n - 1$.

We refer to the ith machine by M_i and to the ith buffer by B_i. As for terminology, B_i is called the upstream buffer of M_{i+1} and the downstream buffer of M_i. The first machine does not have an upstream buffer and we assume that parts are always available to the first machine (it never gets starved). The last machine does not have a downstream buffer and we

978-1-7281-2989-1/21/$31.00 ©2021 IEEE

assume that it can always output processed parts (i.e., it never gets blocked because of its full downstream buffer).

A machine is operational if it is UP, its upstream buffer is not empty (i.e., it is not starved) and its downstream buffer is not full (i.e., it is not blocked). An important and natural assumption to keep in mind is that only operational machines can break down. In other words, a machine that is UP but starved and/or blocked cannot break down, it remains in its UP state.

Let us turn our attention to the dynamics of the model. As already mentioned we assume that things happen in slots, i.e., we have a discrete time model. We can think of the update of the state of the model in a time unit as a two-phase procedure: first, the states of all the machines are determined (probabilistically) then the buffer occupancies are changed accordingly (our second PRISM encodings will follow exactly this logic.) If machine M_i is operational and does not break down (which happens with probability $1 - p_i$) then it takes one part from its upstream buffer and puts one part in its downstream buffer. If it breaks down then it does not move any part. If it is UP but either starved or blocked or both then it remains UP with probability 1 and does not move any part. If machine M_i is $DOWN$ then it gets repaired with probability r_i and in the same slot it can move a part from its upstream buffer to its downstream buffer. If it remains $DOWN$ (with probability $1 - r_i$) then it does not move any part.

The state of the model is given by the state of the machines and the number of parts in the buffers, i.e. by a $(2n-1)$-tuple of the form

$$(m_1, b_1, m_2, b_2, \dots b_{n-1}, m_n)$$

where $m_i \in \{U, D\}$ is the state of the i-th machine and $b_i \in \{0, \dots n_i\}$ is the number of parts in buffer B_i. The stochastic process at hand is a discrete time Markov chain (DTMC).

In order to give an examples for a transition of the DTMC, consider a line with three machines being in the state $(U, 4, D, 0, U)$ with buffer sizes $n_1 = n_2 = 10$. Accordingly, the last machine is starved and cannot change state. The first machine either breaks down or remains up and the second either gets repaired or remains down. This means that there are four possible transitions with associated probabilities as follow

$$(U,4,D,0,U) \xrightarrow{(1-p_1)\cdot(1-r_2)} (U,5,D,0,U)$$
$$(U,4,D,0,U) \xrightarrow{p_1\cdot(1-r_2)} (D,4,D,0,U)$$
$$(U,4,D,0,U) \xrightarrow{(1-p_1)\cdot r_2} (U,4,U,1,U)$$
$$(U,4,D,0,U) \xrightarrow{(p_1\cdot r_2} (D,3,U,1,U)$$

Figure 2 shows the complete state-transition graph of the DTMC for a 2 machines line and assuming $n_1 = 4$ as the size of the single buffer (for the sake of space we denote $\overline{p} = 1 - p$). Assuming that the initial state is $(U, 0, U)$ the number of reachable states is 13. Note that the initial state is transient meaning that the system cannot reach it once it is left. The state space is irregular close to the boundaries (empty or full buffer). For example, having the buffer full is possible only with M_1 up and M_2 down.

III. Probabilistic model checking

The integrated framework for modelling and performance analysis of production lines we propose in this paper is based on probabilistic model checking (PMC) [3]–[5]. The overall idea behind model checking [6] is that of providing a modeller with algorithms for the automatic verification of properties formally expressed in terms of formulae of a given temporal logic. In the "classical" model checking settings models are state-transition graphs and the verification of a temporal logic property φ yields a boolean result: φ is either *satisfied* or *not satisfied* by the model. PMC extends classical model checking to the realm of probabilistic models, i.e., to "probabilistic" state-transition graphs (including DTMCs). Given a probabilistic model $\mathcal{M}$ the verification of property φ yields a probability value, denoted $P(\mathcal{M}, \varphi) = p \in [0, 1]$, which reads "formula φ is satisfied by $\mathcal{M}$ with probability p".

Because of the inherent *discrete-time* nature of the production lines we consider, our framework is based on the version of PMC that targets DTMC which is known as probabilistic computational tree logic (PCTL) model checking. We briefly summarise the basic elements (syntax and semantics) of PCTL model checking referring the reader to the literature for a complete treatment [10], [11].

DTMCs and model checking. In the context of PCTL model checking a DTMC is a k-dimensional ($k \in \mathbb{N}$) process whose set of state S is assumed to be $S \subseteq \mathbb{N}^k$, i.e. a state $s \in S$ is assimilated to a k-tuple of countable state variables, with s_i denoting the i-th state variable. Furthermore a subset of states can be specified through an *atomic proposition* μ (i.e. a statement of the propositional logic built on top of the DTMC state variables) with the following form $\bigvee_j(\bigwedge_{i\leq k} s_i \sim n_i)$, where s_i denotes the i-th state variable of the DTMC, $\bigvee$ the logical disjunction, $\bigwedge$ the logical conjunction, $\sim \in \{<, \leq, =, \neq, \geq, >\}$, $n_i \in \mathbb{N}$. For example, w.r.t. to the DTMC in Figure 2, atomic proposition $\mu_1 : (b_1 = 3)$ corresponds to subset of state $S_{\mu_1} : \{(U, 3, U), (U, 3, D), (D, 3, D)\}$, whereas $\mu_2 : (m_1 \neq D \wedge b_1 > 0 \wedge b_1 < 3) \vee (b_1 > 3)$ corresponds to subset of state $S_{\mu_1} = \{(U, 1, U), (U, 2, D), (U, 2, U), (U, 4, D)\}$. A path of a DTMC is an observable *execution* of the DTMC, i.e. a sequence of states $\sigma : s^1, s^2, \dots, s^i, \dots$ such that for each state s^i there is a non-null probability to jump to s^{i+1}, that is $\mathbf{P}(s^i, s^{i+1}) > 0$, where $\mathbf{P}$ is the transition probability matrix of the DTMC. A DTMC model inherently induces a probability space on the underlying set of paths such that the probability measure of an *event*, i.e., a set of infinite paths sharing a common finite prefix $\sigma_n =: s^1, s^2, \dots, s^n$, is defined as $Prob(\sigma_n) = \prod_{i=1}^{n-1} \mathbf{P}(s^i, s^{i+1})$.

PCTL syntax. The PCTL temporal logic allows for reasoning about the dynamics underlying a DTMC model by means of formulae that involve both classical operators (i.e., conjunction, negation) of the propositional logic as well as *temporal operators* (e.g., *next* and *until*). Specifically PCTL formulae are split into *state-formulae* (φ), i.e., for expressing

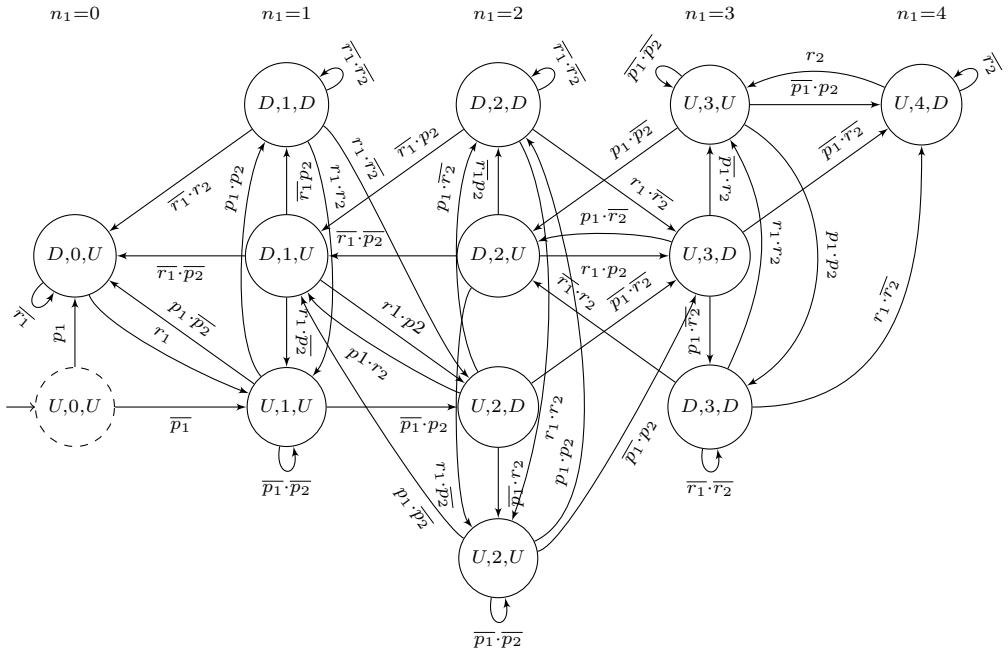

Fig. 2. The DTMC for the MBM system with buffer size $n_1 = 4$ consists of 13 states.

conditions whose truth is established w.r.t. states of a DTMC, as opposed to *path-formulae* (ϕ), i.e., for conditions whose truth is established w.r.t. possible evolutions that can be observed from a given state of the DTMC. Formally PCTL formulae are terms of the following grammar:

$$\varphi \ ::= \ \top \mid \mu \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid P_{\sim p}[\phi]$$
$$\phi \ ::= \mathbf{X}\varphi \mid \varphi_1 \ \mathbf{U}^{\leq k} \ \varphi_2 \mid \mathbf{F}^{\leq k}\varphi$$

where $\top$ stands for the `true` formula, μ denotes an atomic proposition (i.e., an inequality involving a model's state-variables), $\neg$ and $\wedge$ are the basic negation and conjunction connectives of propositional logic and $\mathbf{X}$ is the *next* temporal operator, $\mathbf{U}^{\leq k}$ is the time-bounded *until* temporal operator with $k \in \mathbb{N}$ being the bounding interval and $\mathbf{F}^{\leq k}$ is the (time bounded) *sometime in the future* operator, which is a special case of the *until* where $\phi \equiv \top$.

PCTL semantics. PCTL formulae are interpreted w.r.t. the states of a DTMC. To specify a property of a DTMC we always use a state formula φ. Intuitively, a state s satisfies a formula φ, denoted $s \models \varphi$, that do not involve the probabilistic operator $P_{\sim}[\cdot]$ if by substituting the state variables present in φ with the corresponding values of s the formula evaluates to true. Conversely a formula $P_{\sim p}[\phi]$ is satisfied in s if from s the probability of taking a path that satisfies ϕ falls in the interval corresponding with $\sim p$.

For path formulae ϕ we consider two basic temporal operators $\mathbf{X}$ (next) and $\mathbf{U}^{[t_1,t_2]}$ (time-bounded until). Path formulae ϕ are interpreted w.r.t. a path σ. A next formula $\mathbf{X}\varphi$ is satisfied by σ if φ is satisfied in the state which is next to the initial state of σ, i.e., $\sigma[1] \models \varphi$. A (time-bounded) until formula $\varphi_1 \ \mathbf{U}^{[t_1,t_2]} \ \varphi_2$ is satisfied by σ if $varphi_2$ is true within $k \in [t_1, t_2]$ transitions from the initial state $\sigma[0]$ and φ_1 is true up until this point.

Example 1: Referring to the DTMC model of Figure 2 let us consider the following example of informally stated temporal property together with its corresponding PCTL encoding. ϕ_1: *" there is at least 25% probability that, within 2 time units, the buffer gets empty and the second machine never breaks down ever since."*. The PCTL encoding of ϕ_1 is as follows:

$$\phi_1 \equiv P_{\geq 0.25}[(m_2 = U)\mathbf{U}^{\leq 2}(b_1 = 0)]$$

Let us suppose we want to verify whether ϕ_1 is satisfied in state $(D, 2, U)$, i.e., the state where the first machine is broken, the second machine is working and the buffer contains 2 items. Since the DTMC contains only one reachable state corresponding to the buffer being empty $(D, 0, U)$ we have to consider the probability measure of those paths σ that starting from $(D, 2, U)$ (i.e., $\sigma[0] = (D, 2, U)$) lead us to $(D, 0, U)$ with at most 2 transitions. From the state-transition graph of Figure 2 it is straightforward to see that there exists a single a finite path $\sigma_1 : (D, 2, U), (D, 1, U), (D, 0, U)$ that satisfies the until formula $(m_2 = U)\mathbf{U}^{\leq 2}(b_1 = 0)$ (in fact σ_1 has length ≤ 2, and condition $b_1 = 0$ is matched in the last state of σ_1 while $m_2 = U$ is matched all along. Therefore the probability to satisfy $(m_2 = U)\mathbf{U}^{\leq 2}(b_1 = 0)$ in state $(D, 2, U)$ corresponds to the probability measure of σ_1, i.e., $Prob(\sigma_1) = (\overline{r_1} \cdot \overline{p_2})^2$ hence $(U, 2, U) \models \phi_1$ as long as $(\overline{r_1} \cdot \overline{p_2})^2 \geq 0.25$.

IV. Production lines in PRISM

A. PRISM modelling language

PRISM [12] is a probabilistic model checking tool suitable for the verification of different kinds of probabilistic models,

including discrete-time and continuous-time Markov chains. It provides the user with a simple state-based modelling language, called *reactive modules*, which uses (bounded) discrete-valued variables to characterise the states of the system and *guarded commands* to characterise the (probabilistic) state-transitions. A PRISM program consists of a collection of *modules* each of which represents the behaviour of a part of the modelled system. Notice that in many cases a PRISM model can be conveniently defined in terms of a single module, in others, it can be decomposed in a number of (synchronising) modules. Each PRISM module consists of a list of guarded-commands where a guarded-command is a program instruction with the following format:

[action_name] **guard** $\rightarrow$ **probabilistic_updates**;

where

- **action_name** is an optional label that can be used to synchronise commands of different modules,
- **guard** is a boolean condition on state variables
- **probabilistic_updates** is a '+' separated list of updates with the following form

  $p_1 : (\textbf{update}_1) + ... + (p_n) : (\textbf{update}_n)$

  where $p_i \in [0,1]$ and $\sum p_i = 1$ and $\textbf{update}_i$ is a '&' separated list of single variable updates with the following form:

  $(var_1' = \textbf{exp}_1)\&...\&(var_m' = \textbf{exp}_m)$

  where $\textbf{exp}_i$ is an integer-valued expression (built according to a specific syntax, see [13]).

```
dtmc
const double p1=0.01;  // fail probability
const double r1=0.1;  // repair probability
const int m1init=1;  // initial state
module M1 // a faulty/repairable machine
  m1  :  [0..1] init m1init;
  [fail] m1=1   -> p1 : (m1'=0) + (1 - p1) : (m1'=1);
  [repair] m1=0   -> r1 : (m1'=1) + (1 - r1) : (m1'=0);
endmodule
```

Fig. 3. PRISM encoding of a DTMC model for a faulty/repairable machine

Example 2: Figure 3 shows a simple example of PRISM code for a 2-states DTMC model representing a faulty/repairable production machine. The PRISM model consists of a single **module** named M1, which uses a single binary (state) variable named m1. The model depends on 2 real-valued parameters that represent the probability that, at a given instant of time, a fault occurs on the machine while it is operational (p1), the probability that the machine gets repaired (when it is broken) (r1) plus an integer-valued parameter representing the initial state of the machine (initM1). Module M1 consists of 2 guarded commands. Command labelled [fail] is enabled when the machine is operational (i.e. guard m1=1), and in this case with probability p1 variable m1 is updated to M1'=0, whereas with probability (1-p1) variable m1 is left unchanged. Conversely command labelled [repair] is enabled when the machine is broken (i.e. m1=1), and in this case with probability r1 m1 is set to m1'=1, whereas with probability (1-r1) variable m1 is left unchanged.

PRISM synchronised modules. PRISM supports modular modelling, i.e. a PRISM model may consists of a set of *synchronising* **modules**. In practice synchronisation is obtained through equally labelled commands spread between different modules (see Example 3). If in a given state s, a command labelled [label1] of a module M1 is enabled then it is going to synchronise with any other command labelled [label1] of any another module which also happens to be enabled in s. The **update** corresponding to the synchronisation of several commands is, by definition, given by the *product* of the **updates** of each synchronising command. For example let us consider the synchronisation of the 2 commands labelled [action1] of modules M1 and M2 in the code snippet below.

```
module M1
  [action1] guard1 -> p1 : (updt1,1) + ... + (pn) : (updtn,1)
endmodule
module M2
  [action1] guard2 -> q1 : (updt1,2) + ... + (qm) : (updtm,2)
endmodule
```

The update corresponding to the synchronisation of commands [action1] is:

$p_1 q_1 : (\textbf{updt}_{1,1})\&(\textbf{updt}_{1,2}) + ... + p_n q_m : (\textbf{updt}_{n,1})\&(\textbf{updt}_{m,2})$

```
dtmc
const double p1=0.01;  // fail probability
const double r1=0.1;  // repair probability
const int m1init=1;  // machine initial state
const int b1init=0;  // buffer initial state
const int n1=5;  // buffer size
module M1 // a faulty/repairable machine
  m1  :  [0..1] init m1init;
  [tic] m1=1   -> p1 : (m1'=0) + (1 - p1) : (m1'=1);
  [tic] m1=0   -> r1 : (m1'=1) + (1 - r1) : (m1'=0);
endmodule
module B1 // a finite-size buffer
  b1  :  [0..n1] init b1init;
  [tic] m1=1   -> 1 : (b1'=min(b1 + 1, n1));
  [tic] m1=0   -> 1 : (b1'=max(b1 - 1, 0));
endmodule
```

Fig. 4. PRISM encoding of a DTMC model for a faulty/repairable machine

Example 3: Figure 4 shows an extension of Example 2 where the faulty machine M1 is now coupled with a finite-size buffer B1. The machine behavior is unchanged however it now synchronises with the buffer (action [tic]) reproducing the following behaviour: when the machine is operational an item is added into the buffer, conversely, when the machine is broken an element is removed from the buffer.

B. PRISM models of production lines

The dynamics of the production line systems (Section II) is of a *fully synchronous* nature: the state of the machines

```
dtmc
module ProductionLine
m1 : [0..1] init m1init;
m2 : [0..1] init m2init;
m3 : [0..1] init m3init;
b1 : [0..n1] init b1init;
b2 : [0..n2] init b2init;
[] (m1=0)&(m2=0)&(m3=0)&(b1=0)&(b2=0) →
    (1 − r1) ∗ (1 − r2) ∗ (1 − r3) : true+
    (1 − r1) ∗ (1 − r2) ∗ r3 : (m3'=1)+
    (1 − r1) ∗ r2 ∗ (1 − r3) : (m2'=1)+
    (1 − r1) ∗ r2 ∗ r3 : (m2'=1)&(m3'=1)+
    r1 ∗ (1 − r2) ∗ (1 − r3) : (m1'=1)&(b1'=b1 + 1)+
    r1 ∗ (1 − r2) ∗ r3 : (m1'=1)&(m3'=1)&(b1'=b1 + 1)+
    r1 ∗ r2 ∗ (1 − r3) : (m1'=1)&(m2'=1)&(b1'=b1 + 1)+
    r1 ∗ r2 ∗ r3 : (m1'=1)&(m2'=1)&(m3'=1)&(b1'=b1 + 1);
[] (m1=0)&(m2=0)&(m3=0)&(b1=0)&(0<b2)&(b2<n2) →
    (1 − r1) ∗ (1 − r2) ∗ (1 − r3) : true+
    (1 − r1) ∗ (1 − r2) ∗ r3 : (m3'=1)&(b2'=b2 − 1)+
    (1 − r1) ∗ r2 ∗ (1 − r3) : (m2'=1)+
    (1 − r1) ∗ r2 ∗ r3 : (m2'=1)&(m3'=1)&(b2'=b2 − 1)+
    r1 ∗ (1 − r2) ∗ (1 − r3) : (m1'=1)&(b1'=b1 + 1)+
    r1 ∗ (1 − r2) ∗ r3 : (m1'=1)&(m3'=1)&(b1'=b1 + 1)&(b2'=b2 − 1)+
    r1 ∗ r2 ∗ (1 − r3) : (m1'=1)&(m2'=1)&(b1'=b1 + 1)+
    r1 ∗ r2 ∗ r3 : (m1'=1)&(m2'=1)&(m3'=1)&(b1'=b1 + 1)&(b2'=b2 − 1);
[] (m1=0)&(m2=0)&(m3=0)&(b1=0)&(b2=n2) →
...
module
```

Fig. 5. PRISM code snippet for the monolithic 1-slot model of a 2-machines production line.

```
dtmc
global b0:[0..1] init 1; // fictitious start buffer
global b3:[0..1] init 1; // fictitious end buffer
module Phase
ph : [1..2] init 1;
[tic] ph=1 → 1 : (ph'=2); // first slot: update machines state
[toc] ph=2 → 1 : (ph'=1); // second slot: update buffer state
endmodule
module M1 // intermediate machine, b0: upstream buffer, b1: downstream buffer
  m1 : [0..1] init m1init;
  [tic] m1=1 → p1 : (m1'=(b1=n1|b0=0)?1 : 0) + (1 − p1) : (m1'=1);
  [tic] m1=0 → r1 : (m1'=1) + (1 − r1) : (m1'=0);
  [toc] true → 1 : true;
endmodule
module B1 // intermediate buffer, m1: upstream machine, m2: downstream machine,
  b1 : [0..n1] init b1init;
  [tic] true → 1 : true;
  [toc] ((m1=1 & b0>0 & b1<n1) & !(m2=1 & b1>0 & b2<n2)) →
    ; 1 : (b1'=b1 + 1); // increase buffer occupation
  [toc] (!(m1=1 & b0>0 & b1<n1) & (m2=1 & b1>0 & b2<n2)) →
    1 : (b1'=b1 − 1); // decrease buffer occupation
  [toc] ( (!(m1=1 & b0>0 & b1<n1) & !(m2=1 & b1>0 & b2<n2)) |
          ((m1=1 & b0>0 & b1<n1) & (m2=1 & b1>0 & b2<n2)) ) →
            1 : (b1'=b1); // buffer unchanged
endmodule
module M2= M1[m1=m2, b0=b1, b1=b2, p1=p2, r1=r2, n1=n2,
  m1init=m2init]
endmodule
module B2= B1[b1=b2, m1=m2, m2=m3, b0=b1, b2=b3, n1=n2, n2=n3,
  b1init=b2init]
endmodule
module M3= M2[m2=m3, b1=b2, b2=b3, p2=p3, r2=r3, n2=n3,
  m2init=m3init]
endmodule
```

Fig. 6. PRISM code snippet of the 2-slot DTMC model for a 3 machines production line

as well as that of the buffers are updated synchronously. Such characteristic affects the definition of PRISM code for modelling of production lines.

In the remainder we present two distinct, yet semantically equivalent, approaches for modelling production lines through PRISM, which we refer to as *1-slot*, respectively *2-slot*, models[1]. Models of the *1-slot* type exactly reproduces the actual fully synchronous behaviour of a production line, i.e. each transition of the underlying DTMC corresponds with a transition of the actual DTMC of the considered system. On the other hand models of the *2-slot* family split the fully synchronous behaviour, by introducing a fictitious transition through which the determination of the successor of a state is achieved by splitting the updates of the state variables in two steps: in the first step ([tic]) the state of the machines is updated, in the second step ([tic]) the state of the buffers is updated consequently.

In terms of PRISM code the two families of models compare as follows: models of *1-slot* type consist of a single **module** made up with a large number of commands (since each PRISM command must consider the "global" configuration of the system, hence resulting in a combinatorial explosion of conditions to be considered by each command's guard), while models of *2-slot* kind, are based on a modular approach (i.e. a *2-slot* model consists of collection of **modules**, one for each machine plus one for each buffer) result in a much reduced code.

1) Monolithic 1-slot *type of mode:* The main aspect that affect the definition of a synchronised model is that the transitions depend on both the current state of each workstation as well as the current occupation level of each buffer. In this respect it is worth noticing that if a workstation's state can be *up* or *down*, for the buffers we need to distinguish 3 possibilities: *empty* (E), *full* (F), and *some* (S) (i.e., neither empty nor full). This means that for a system consisting of n workstations there are $2^n \cdot 3^{n-1}$ combinations that has to be taken into account (e.g., with $n = 2$ machines there are 12 possibilities, with $n = 3$ machines there are 72, and so on). In terms of PRISM code therefore this implies that (at least) $2^n \cdot 3^{n-1}$ *guarded-commands* are needed (with guards corresponding to relevant combinations of state conditions), which is something that undermines the scalability of PRISM modelling.

Figure 5 shows a portion of PRSIM code of 1-slot type generated for a 3-machines system: it consists of a single **module** equipped with 5 state variables (3 for the machines and 2 for the buffers). The code in Figure 5 only depicts the first 2 (out of 72) guarded commands needed for capturing the dynamics of a 3-machines system. The first command encodes the possible evolution of the system when all machines are down and buffers are empty ($(m1 = 0)\&(m2 = 0)\&(m3 = 0)\&(b1 = 0)\&(b2 = 0)$), the second command concerns the behaviour for when the machines are all down the first buffer is empty while the second is neither empty nor full.

2) Modular 2-slot *type of model:* As discussed above the straightforward PRISM encoding of a fully-synchronous

[1]PRISM models are available at https://gitlab-research.centralesupelec.fr/2011ballarinp/PRISMProductionLines

production line system consisting of n workstations requires $2^n \cdot 3^{n-1}$ PRISM instructions. Here we present a different kind of PRISM encoding that allows for reducing the number of necessary PRISM instructions to $2^n \cdot$. The main idea behind this alternative PRISM encoding is that of removing the buffer's occupation from the conditions that should be accounted for the characterisation of the state-transitions. In this manner we characterise the system's dynamics only in function of the state of the workstations for which we have a total of 2^n possibilities. In practical terms this alternative encoding is obtained by removing the buffer-occupation conditions from the guard of a PRISM command and by letting the *update* part of the command depend on buffer's occupation (by means of *conditional expressions*).

Figure 6 shows the 2-slot type version of the PRSIM code generated for a 3-machines system. It consists of a **module** named Phase (responsible for driving the 2-slot dynamic) plus 3 modules machines and 2 modules buffer. Notice that only the code for one machine module and for one buffer module needs to be explicitly given: the code for the other machines and buffer is obtained through PRISM *module renaming* facility (which handly allows for making copies of a module by renaming of variables and parameters). Module Phase drives the synchronisation: on action [tic] each machine module synchronously update the state of the machine (while each buffer does not do anything through action [tic]), while on action [toc] each buffer module update its state (while each machine does not do anything through action [toc]). By comparing Figure 5 and Figure 6 it is evident that the 2-slot type of PRISM model is indeed a lot shorter than the 1-slot type. As discussed in SectionV-A the price to pay for such shorter PRISM code is in terms of the state-space of the underlying DTMC.

C. Sanity check properties

Since the definition of a PRISM model for this class of production systems is a cumbersome activity prone to errors, we identified a number of necessary conditions, named *sanity checks*, that a model must verify in order to be considered valid. If even just one of such properties is not verified then the model under construction is certainly not representing the correct behavior of the production line system. Below we give a list of informally described sanity checks (denoted SC_i) together with their formal encoding in terms of PCTL formulae (ϕ_{SC_i}). They refer to a generic n machines production line with m_i and b_i representing, the state of the i-th machine, respectively buffer, and n_i representing the size of i-th buffer:

SC_1. *"A machine cannot be broken and its downstream buffer being full"*

$$\phi_{SC_1} \equiv P_{\leq 0}[F\ (\bigvee_{i=1}^{n-1}(m_i = 0 \wedge b_i = n_i))]$$

SC_2. *"Two adjacent buffers that are both full cannot both become not-full in one time unit"*

$$\phi_{SC_2} \equiv \bigwedge_{i=1}^{n-2} \phi_{SC_2,i}$$

with

$$\phi_{SC_2,i} \equiv [(b_i = n_i) \wedge (b_{i+1} = n_{i+1})] \Rightarrow P_{\leq 0}[X\ ((b_i < n_i) \wedge (b_{i+1} < n_{i+1}))]$$

SC_3. *"An empty buffer cannot become non-empty in one time unit if its upstream buffer is empty."*

$$\phi_{SC_3} \equiv \bigwedge_{i=1}^{n-2}([(b_i = 0) \wedge (b_{i+1} = 0)] \Rightarrow P_{\leq 0}[X\ (b_{i+1} > 0))])$$

SC_4. *" A full buffer cannot stay full when its downstream machine gets repaired"*

$$\phi_{SC_4} \equiv \bigwedge_{i=1}^{n-1}((b_i = n_i) \Rightarrow P_{\leq 0}[(m_{i+1} = 0)\ U\ (m_{i+1} = 1 \wedge b_i < n_i))])$$

The PRISM models generated by our framework (both of 1-slot and 2-slot type) have been all validated w.r.t. the above listed the sanity checks.

V. Numerical experiments

A. State space and calculation of steady state

Based on PRISM facilities for model construction and steady-state computation we first compared the 1-slot and 2-slot approaches w.r.t. the state-space size, and run time for building the model and for computing the steady-state distribution[2]. Figure 7 depicts results of such comparison. Model parameters are the following. The number of machines is 3,4 or 5. The buffers have the same capacity. The machine parameters (p_i, r_i) are chosen randomly in such a way that standalone machine availability is in the interval [0.9,0.95] and repair probability is 20 times larger than break down probability. The number of states (1st sub-figure) is about doubled by the 2-slot approach. Interestingly, this increase almost disappears when it comes to the number of the transitions (2nd sub-plot). This is due to the fact that in the 2nd phase of the 2-slot approach every state has a single outgoing transition with probability one (corresponding to the update of the buffer occupation levels) and the number of transitions is dominated by the many outgoing arcs most states have in the 1st phase. For what concerns the execution times of constructing the representation of the state space (3rd sub-plot) the 2-slot approach performs better due to the minor complexity of the corresponding PRISM description. The larger the number of machines the higher the difference. When it comes to calculating the steady state distribution (number of iterations on the 4th sub-plot and total execution time in the 5th) the 1-slot approach outperforms the 2-slot encoding because the presence of the transitions with probability 1 makes the convergence toward steady state slower for the 2-slot approach.

It happens however that models that cannot be managed with the 1-slot approach can be solved with the 2-slot encoding. 1-slot models with more than 5 machines cannot be solved on an ordinary portable computer because of the huge size of

[2]Experiments were run on standard portable computers

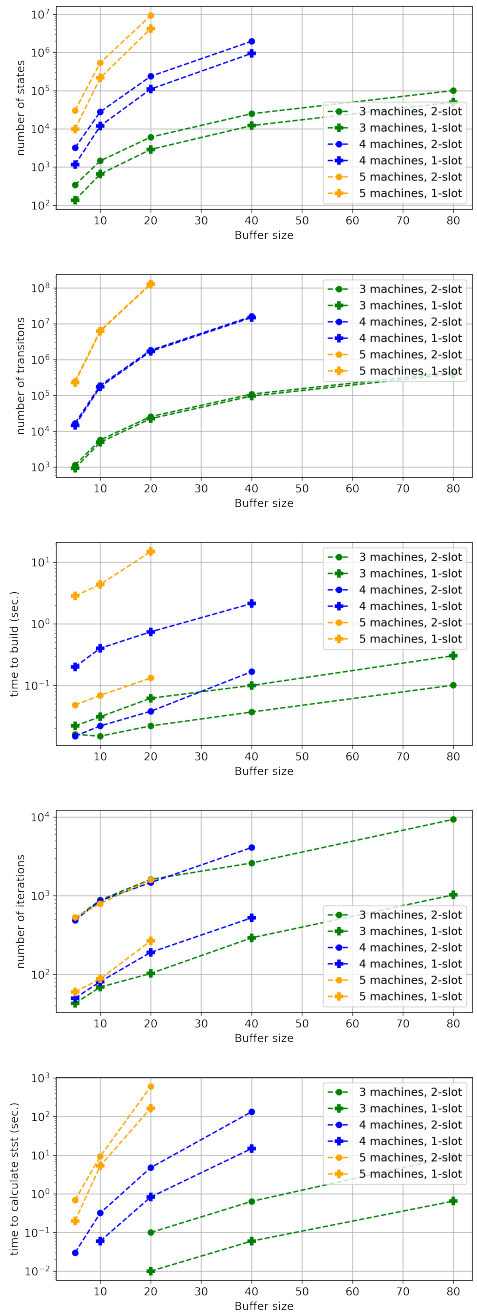

Fig. 7. Characteristics of the state space and corresponding calculations

the 1-slot encoding. With the 2-slot approach and relatively small buffer sizes, 6 or 7 machines can be dealt with. In the rest of the paper we report results obtained with the 1-slot approach.

To conclude we point out that (for this kind of system) the computation of the throughput (denoted by E) coincides with a steady-state measure. This is because within a linear, loops-free, layout, parts cannot be lost and follow the same path, therefore the throughput of any single machine equals the throughput of the whole system, hence any machine can be used to determine the throughput. In particular, the throughput equals the steady-state probability (denoted by S) that a given machine is up and it is neither blocked nor starved. Hence we have

$$\begin{aligned} E = & S(m_1 = U \wedge b_1 < n_1) = \\ & S(m_2 = U \wedge b_1 > 0 \wedge b_2 < n_2) = \\ & S(m_{n-1} = U \wedge b_{n-2} > 0 \wedge b_{n-1} < n_{n-1}) = \ldots = \\ & S(m_n = U \wedge b_{n-2} > 0) \end{aligned}$$

B. Lead time of specific parts

Next we show that the PRISM encoding allows for specifying and calculating lead time distribution of a part that enters the system in a specific situation. In other words, knowing the state of the system when a given part enters, we can calculate the probability that it will be ready (exits the system) in a given number of time units. Also in [9] a way to calculate lead time distributions is proposed. In that work however the lead time of a random part is determined assuming no knowledge on the state of the system when the part enters.

The calculations are carried out the following way. We set the initial state of the model (both machine and buffer states) to the state of the system when the considered part enters the first buffer. We modify the initial state of M_1 to down and make it irreparable by setting r_1 to 0. In this way no more parts can enter the system and the moment in which the considered part is ready coincides with the moment in which all buffers become empty. Accordingly, the lead time distribution of the considered part is equal to the distribution of the time that is needed to reach a state in which all buffers are empty.

In PRISM the distribution of time to empty buffers can be formulated using the property

$$P_{=?}[\mathbf{F}^{\leq T}(b1 = 0 \ \wedge \ b2 = 0 \ \wedge \ b3 = 0)]$$

where we assumed to have three buffers in the model and used $P_{=?}$ to indicate that we want PRISM to return the probability and not only a yes or no answer. As a result, PRISM calculates the probability that the considered part is ready after T time units and by varying T the lead time distribution is obtained.

Some experiments are reported in Figure 8. The number of machines is 4, all buffers are with capacity equal to 20. We vary the initial buffer occupancy levels but keep the initial total number of parts identical. The initial state of M_1 is down (and it remains down as explained above). The initial state of M_2 is up for all experiments. The initial state of M_3 and M_4 is instead varied. In all cases, since there are 30 parts initially in the system, at least 30 time units are necessary to reach empty buffers. Another common feature is that the cumulative distribution function tends to 1 because eventually the system gets empty with probability 1. As expected the fastest convergence to 1 is with the cases when M_3 and M_4 are up initially (upper three curves of the upper sub-plot). Among these cases, the more parts are in the last buffer initially the shorter the lead time will be. The other three curves of the same sub-plot are with M_3 and M_4 down initially. Also in this case the more parts are in the last buffer the better. Not

surprisingly, in this case the probability that the system gets empty in 30 time units is much lower. On the lower sub-plot we have the cases in which M_3 is up and M_4 is down initially, or vice-versa. It is interesting to note that in the case M_3 up and M_4 down the initial buffer state does not affect much the results. This is due to the fact that with M_4 down the last buffer can get full with high probability in all cases. On the other hand, with M_3 down and M_4 up initially, the results are highly affected by the initial occupation levels.

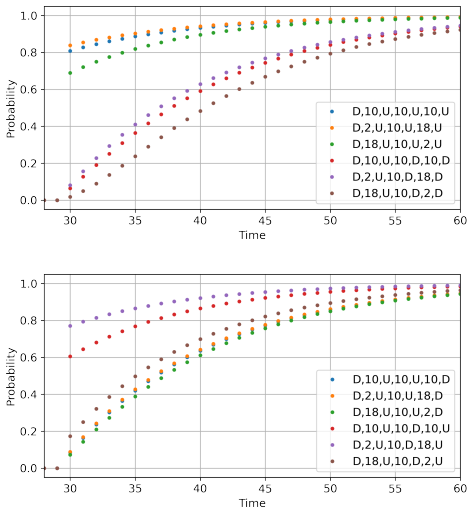

Fig. 8. Cumulative lead time distribution of a part entering the system in a specific state

With a property containing the **U** operator, it is possible to calculate the lead time distribution under some condition. Consider

$$P_{=?}[(m2 = 1)\mathbf{U}^{\leq T}(b1 = 0 \ \wedge \ b2 = 0 \ \wedge \ b3 = 0)]$$

which can be used to determine the distribution of time to empty buffers conditioned by machine M_2 never breaking. We performed the same experiments as before and the results are shown in Figure 9. The curves do not tend to 1 because, while it is guaranteed that the system gets empty, it cannot be ensured that M_2 does not break down in the meantime. Accordingly, every curve in Figure 9 is under the corresponding curve of Figure 8 for any value of the x-axis. The curves with the same number of parts initially in B_1 tend to the same value. This is due to the fact that these share the number of parts that M_2 has to process and hence share also the number of time units in which M_2 can break down. As expected, the potential vulnerabilities of M_2 are more amplified when it has to process more parts. For example, the curves with 18 parts initially in B_1 tend to a lower value than those with 10.

VI. Conclusion

In this paper we described a framework that allows for both the definition of DTMCs describing production systems and their analysis. The analysis can concern classical performance indices (throughput, probability of empty or full buffer, etc.)

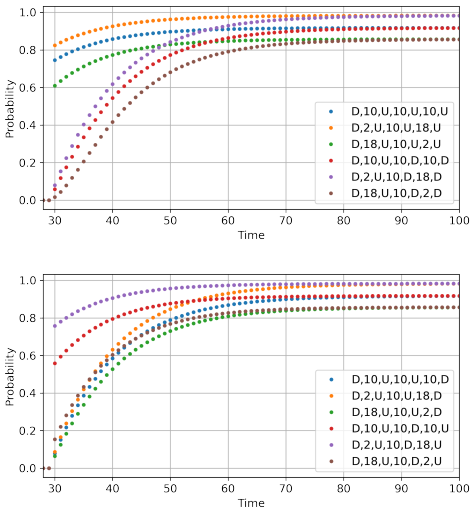

Fig. 9. Conditioned cumulative lead time distribution of a part entering the system in a specific state

and also more complex temporal behavior of the system (like lead time distributions). The framework was implemented using PRISM, a probabilistic model checker, assuming two-state machines and linear production lines but can easily be extended to more general machine models and system layout.

References

[1] A. Angius, A. Horváth, and M. Colledani. Moments of accumulated reward and completion time in Markovian models with application to unreliable manufacturing systems. *Performance Evaluation*, 75:69–88, 2014.

[2] Alessio Angius, Marcello Colledani, András Horváth, and Stanley B Gershwin. Analysis of the lead time distribution in closed loop manufacturing systems. *IFAC-PapersOnLine*, 49(12):307–312, 2016.

[3] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking CTMCs. *ACM Trans. on Computational Logic*, 1(1), 2000.

[4] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for CTMCs. *IEEE Trans. on Software Eng.*, 29(6), 2003.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[7] M. Colledani, A. Horváth, and Alessio Angius. Production quality performance in manufacturing systems processing deteriorating products. *Cirp Annals-manufacturing Technology*, 64:431–434, 2015.

[8] M. Colledani and T. Tolio. Integrated quality, production logistics and maintenance analysis of multi-stage asynchronous manufacturing systems with degrading machines. *CIRP Annals-Manufacturing Technology*, 61/1:455–458, 2012.

[9] Marcello Colledani, Alessio Angius, and András Horváth. Lead time distribution in unreliable production lines processing perishable products. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, 2014.

[10] H. A. Hansson and B. Jonsson. A framework for reasoning about time and reliability. In *Proc. 10th IEEE Real -Time Systems Symposium*, pages 102–111, Santa Monica, Ca., 1989. IEEE Computer Society Press.

[11] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation*, volume 4486 of *LNCS*, pages 220–270. Springer, 2007.

[12] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd Int. Conf. on Computer Aided Verification (CAV'11)*, pages 585–591, 2011.

[13] Prism home page. http://www.prismmodelchecker.org.