Proceedings Paper:
Gleirscher, Mario orcid.org/0000-0002-9445-6863 (2020) Yap: Tool Support for Deriving Safety Controllers from Hazard Analysis and Risk Assessments. In: Second Workshop on Formal Methods for Autonomous Systems (FMAS2020). Electronic Proceedings in Theoretical Computer Science . Open Publishing Association , p. 31.

https://doi.org/10.4204/EPTCS.329.4

YAP: Tool Support for Deriving Safety Controllers from Hazard Analysis and Risk Assessments

Mario Gleirscher*
Dept. of Computer Science, University of York, York, U.K.
mario.gleirscher@york.ac.uk

Safety controllers are system or software components responsible for handling risk in many machine applications. This tool paper describes a use case and a workflow for YAP, a research tool for risk modelling and discrete-event safety controller design. The goal of this use case is to derive a safety controller from hazard analysis and risk assessment, to define a design space for this controller, and to select a verified optimal controller instance from this design space. We represent this design space as a stochastic model and use YAP for risk modelling and generation of parts of this stochastic model. For the controller verification and selection step, we use a stochastic model checker. The approach is illustrated by an example of a collaborative robot operated in a manufacturing work cell.

1 Introduction

To ensure their safe operation, machines, such as mobile robots or delivery drones, incorporate controllers responsible for the *handling of critical events (CEs)* while performing their tasks. We will refer to such controllers as *safety controllers*. CEs can be failures, human errors, or other hazards, any causes thereof and any consequences, such as incidents or accidents. CE handling can involve the anticipation and mitigation of hazards and the prevention and alleviation of accidents, for example, by switching a machine into a mode with lower risk, that is, a *safety mode*, by performing a *safety function* (e.g. a warning signal), or by changing the machine's activity. Therefore, safety controllers have to be carefully specified, designed, and verified in order to be deployed according to state-of-the-art regulations [11, 9].

This tool paper supplements our approach to the verified synthesis of safety controllers [6] with a hands-on guide to the research tool YAP *Against Perils* [5]. One objective of YAP is to support the steps required to transform results from hazard analysis into verifiable models of safety controllers. YAP seeks to bridge the gap between the identification of hazards, the formulation of safety goals, and the implementation of safety controllers. Although YAP might be more widely used, adaptive and autonomous cyber-physical systems with their highly automated and complex safety mechanisms [2] are in its focus.

Sect. 2 briefly revisits the example discussed in more detail in [6]. Sect. 3 describes preliminaries of YAP. Sect. 4 proposes a workflow to derive a safety controller. Sects. 5.1 to 5.5 detail this workflow in the format of a hands-on guide. Sects. 6 and 7 discuss directions for future work and conclude.

2 Running Example: A Collaborative Manufacturing Robot

We illustrate the proposed workflow by example of a human-robot collaboration (HRC) in a manufacturing work cell with a collaborative robot arm [6]. This work cell has a safeguarded workbench, which is manually supplied with work pieces to be processed by a robot arm and a welder within a safeguarded

*This research was funded by the Lloyd's Register Foundation under the AAIP grant CSI:Cobot.

M. Luckcuck, M. Farrell (Eds.): Second Workshop
on Formal Methods for Autonomous Systems (FMAS2020)
EPTCS 329, 2020, pp. 31–47, doi:10.4204/EPTCS.329.4

area next to the workbench. The robot moves to the workbench, grabs the work piece, and moves to the welder. The robot and the welder together perform a specific welding task on the work piece. After finishing this task, the robot arm returns the work piece to the workbench where the operator picks it up and supplies the robot with another work piece to repeat this cycle. The work cell is equipped with several safety modes (e.g. safety-rated monitored stop) and safety functions (e.g. a warning display) operated by a safety controller on occurrence of a CE (e.g. operator close to weld spot while welder and robot are working). This way, the safety controller works on top of this cyclic manufacturing process.

3 Overview of YAP: Modelling Concepts and Tool Features

YAP is a research tool for risk modelling, analysis, and design of safety controllers.[1] YAP's input language is a domain-specific language (DSL) providing a corresponding set of modelling primitives.

Activities provide a finite abstraction of the physical process of interest and can be useful for modelling the task structure of an application as well as for structuring a risk analysis accordingly. For example, the task of the robot arm exchanging a work piece can be separated from a welding task performed by the welder and the robot arm.

Risk factors (factors for short) describe the *life-cycle* and constituents of CEs potentially being observed when performing the activities, for example, the robot arm and the operator being on the workbench simultaneously. The hazard list can be modelled as a list of factors. *Factor dependencies* specify temporal or causal relations between risk factors (e.g. `requires`, `prevents`). For example, the fact that the robot arm touches the operator `requires` the operator to be in one of the safeguarded areas.

A factor is modelled by four life-cycle *phases* (i.e., inactive 0^f, active f, mitigated $\bar{f}$, and mishap $\underline{f}$) and five *events* (i.e., endangerment, mitigation, resumption, mishap, alleviation). Four of these events can be refined into *modes* and attributed with (quantitative) *parameters*. For example, the *endangerment* from an operator entering the workbench while the robot is handling a work piece there is *detected* by a light barrier and the robot position. This event can be *mitigated* by a safety-rated monitored stop (`srmst`) and signalling the operator to leave the workbench. After the operator has followed this advice, the robot can *resume* its work piece handling. In case of a detected mishap, a potential consequence could be *alleviated* by a complete shutdown of the work cell and an emergency call. Modes can be embodied by physical and logical *items*, particularly, *actors* (synonymously, agents), constituting the *application*. For example, the logic for the safety-rated monitored stop could be embodied by the robot arm.

These modelling primitives will be explained and used in Sect. 5. A more detailed description of YAP's DSL is, however, provided in [5]. Overall, YAP models can inform controller design by injection of a model of a safety controller into a process model of the application. However, apart from this use case, with YAP's DSL one can describe operational risk as an abstract state machine, explore its symbolic state space, that is, the *risk space*, shape its transition relation, and perform a light-weight symbolic simulation of CE occurrence and handling. Furthermore, one can calculate risk spaces and properties thereof (e.g. mitigation orders [7]), and generate minimal cut sequences [6, 5].

4 Overview of the Workflow

Figure 1 indicates the methodological context in which YAP can be used. There, the specification and synthesis of a safety controller consists of several work steps (1 to 12) in five stages.

[1] YAP, its manual, and the running example can be obtained from `http://www.gleirscher.de/yap/` or from the author.

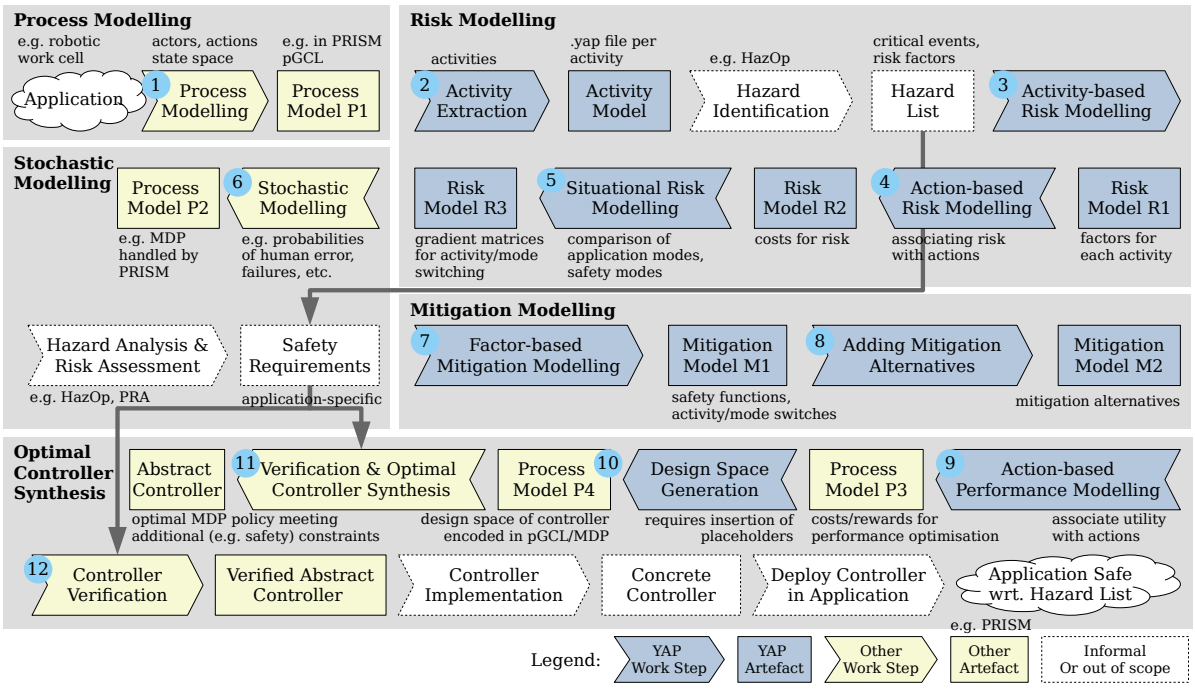

Figure 1: Intended workflow to be used with YAP and the stochastic model checker

Process Modelling. 1 One begins with constructing a behavioural model (P1) of the physical process of interest, focusing on actors, actions, and the state space these actions can modify (Sect. 5.1). For this use case of YAP, we encode this model in the probabilistic guarded command language (pGCL) of the PRISM stochastic model checker [15] for the analysis of Markov decision processes (MDPs). An MDP is a stochastic model, particularly well-suited for reasoning about processes with non-deterministic decisions over actions and probabilistic outcomes of these actions. pGCL offers a concise way of encoding MDPs. PRISM conveniently implements pGCL with parallel composition [8, 21].

Risk Modelling. 2 To structure the risk analysis, we decompose the process into activities (Sect. 5.2). 3 For each activity, we identify risk factors and incidents using an appropriate technique (e.g. the hazard identification stage of hazard operability studies (HazOp) [10]) and capture the results, that is, the hazard list, in a risk model (R1, Sect. 5.2). The practical reasoning and the domain expertise to apply when using a typical system hazard analysis and risk assessment (HARA) technique, are very well-documented in a rich corpus of literature (e.g. [18, 13, 3]). Hence, we touch HARA specifics only selectively. 4 We extend this risk model (R2) by associating to every action a number that encodes the risk of any of the captured incidents from the performance of this particular action (Sect. 5.2). 5 Furthermore, the risk model (R3) is extended by assigning to every transition between two activities or safety modes a number that encodes the increase or decrease in risk of any incident when taking this transition (Sect. 5.2). The result are two *gradient matrices*, one for activity changes and one for safety mode changes.

Stochastic Modelling. 6 We return to the process model and introduce probabilistic phenomena such as human error and sensor failure (Sect. 5.3). As a result, we get a process model (P2) amenable to HARA, for example, further steps of HazOp and, particularly, probabilistic risk assessment (PRA).

Mitigation Modelling. 7 Based on the process and risk models, we design mitigations for each of the identified risk factors (Sect. 5.4). An individual mitigation can perform a safety function, an activity change, and safety mode change, and will after removal of the corresponding risk factor return the process to a state where normal operation can continue. Within YAP, the mitigation model (M1) refines the risk model by an abstract state machine. This state machine is translated into the language used for the process model. 8 YAP allows the definition of alternatives for the mitigation of a single risk factor (Sect. 5.4). The result is a mitigation model (M2) with these alternatives creating a *controller design space*.

Verified Controller Synthesis. 9 Similar to the assignment of risk to process actions, we now associate other costs and rewards (e.g. nuisance of the operator, energy consumption, utility of a performed robot action) with these actions (Sect. 5.5). From this step, we obtain a reward-enhanced MDP model of the application (P3). 10 Next, we use YAP's pGCL generator to translate the risk and mitigation models into a set of pGCL fragments. These fragments fill placeholders easily inserted into the process model beforehand. The resulting process model (P4) is amenable to property verification and acts as a design space for controller synthesis (Sect. 5.5). This design space includes the set of choice resolutions of the MDP as well as further degrees of freedom stemming from other unfixed model parameters (e.g. probabilities 6, rewards 9). We can now verify properties of this design space. 11 Importantly, we use PRISM not only for property verification but also for the selection of a policy (also called adversary or strategy), that is, a particular choice resolution representing the controller, from this design space. Optimal MDP policies are artefacts of quantitative verification and can be represented as discrete-time Markov chains (DTMCs). Depending on the safety requirements, the chosen policy will have to meet certain safety constraints and be Pareto-optimal with respect to the considered performance criteria (Sect. 5.5). 12 We finally verify further safety properties of the policy (Sect. 5.5). Theoretically, 11 and 12 could be collapsed into one verification step carried out on the design space. However, the tooling for this use case requires us to separate Pareto optimisation and constraint verification.

5 Workflow for Controller Design

The following sub-sections provide a hands-on guide detailing the workflow outlined in Figure 1.

5.1 1 Process Modelling: The Physical World

We create a stochastic model (an MDP) of the manufacturing process, as described in Sect. 2, and employ the PRISM model checker for its analysis. We model *actors* (e.g. a robot arm, an operator), *activities* (e.g. exchWrkp for exchanging a work piece, welding a work piece), and (atomic) *actions* (e.g. the robot grabs the work piece, the operator enters the work cell). In PRISM, actors can be implemented as modules and actions as guarded commands of the form `[EVENT] GUARD → UPDATE`.

Listing 1 exemplifies the actor **robotArm**, participating in the two activities exchWrkp and welding with several actions. For example, the involvement of **robotArm** in exchWrkp is implemented by the three actions r_moveToTable, r_grabLeftWorkpiece, and r_placeWorkpieceRight. The structure of many of the modelled actions follows a specific pattern:

`[ACTOR_ACTION] !CYCLEEND & SM & ACTIVITY & CUSTOM → UPDATE .`

Action names carry prefixes to indicate the actor(s) performing these actions, that is, **r** for a robot action, **rw** for a compound action of the robot and the welder, **h** for a physical human action, **hi** for an internal

Listing 1: Process model fragment for the robot arm in PRISM

```
module robotArm
reffocc:   bool       init false; // is the grabber occupied?
wpfin:     bool       init false; // is the work piece finished?
rloc:      [atTable..atWeldSpot]  init inCell; // robot arm location
//<%
...
// exchWrkp: exchange a work piece between workbench and welder
[r_moveToTable] !CYCLEEND & (safmod=normal|safmod=ssmon|safmod=pflim) & ract=exchWrkp & !rloc
    =sharedTbl & ((wps!=right&reffocc)|wps=left&!reffocc) -> (rloc'=sharedTbl);
[r_grabLeftWorkpiece] ...;
[r_placeWorkpieceRight] ...; ...
// welding: carry out welding task together with welder
[r_moveToWelder] !CYCLEEND & (safmod=normal|safmod=ssmon|safmod=pflim) & ract=exchWrkp &
    reffocc & !wpfin -> (ract'=welding)&(rloc'=atWeldSpot);
[rw_weldStep] ...;
[rw_leaveWelder] ...; ...
endmodule
```

human decision, **s** for a synchronous action of the safety controller and other actors, **si** for an independent controller action. `CYCLEEND` is a predicate that, when `true`, terminates model execution. Each action has to be guarded by the activities (`ACTIVITY`) and safety modes (`SM`) it is allowed to be performed in. Action-specific guards (`CUSTOM`) and updates (`UPDATE`) are conjoined and specified. The **robotArm** does not contain stochastic phenomena whereas **humanOp** and other parts of the process model do. The use of probabilistic updates for human errors and other phenomena will be further discussed in Sect. 5.3.

5.2 Risk Modelling with YAP

2 Activity Modelling. We create a YAP file for each activity (e.g. generic task or sub-task) of the manufacturing process. For example, for the activity exchWrkp, the listing on the right specifies relationships to other activities. Particularly, exchWrkp inherits (`includes`) attributes (i.e., hazard, activity successors, etc.) from the generic activity moving and can be followed (`successor`) by either of the basic activities off, welding, or idle. With the following command, YAP identifies all activities reachable from the activity off in form of a labelled transition system (LTS). For inspection, this LTS is visualised as a graph in Figure 2.

```
Activity {
  include moving;
  successor welding;
  successor off;
  successor idle;
}
```

```
yapp --global-logging -m off.yap \
     -o output/off-act.dot --showmodel activities .
```

3 Activity-based Risk Modelling. We create a hazard list for each activity, for example, by performing a HazOp [10]. Then, we derive factor specifications from these lists. For instance, for the activity welding, we specify five factors, such as the factor "Human arm and Robot on shared Workbench" (HRW, Figure 3). Then, we specify factor dependencies, for example, HRW `requires` the factor "Human arm on Workbench" (HW), `prevents` the activation of factor "Human Close to weldspot" (HC), and HRW's mitigation prevents the mitigation (`mitPreventsMit`) of HC and "Human in Safeguarded area" (HS).

The inclined reader will recognise that HRW is not only a critical event in the process, it also contains the hazard "Robot on shared Workbench". As a whole, HRW can be seen as a latent cause of the incident

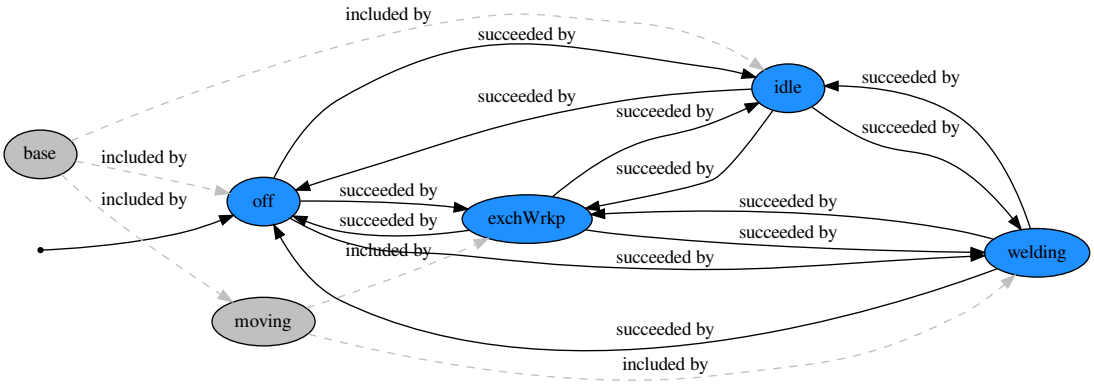

Figure 2: Activity graph showing all activities reachable from the activity off

```
HRW desc "(H)uman arm and (R)obot on
          shared (W)orkbench"
  requires (HW)
  prevents (HC)
  mitPreventsMit (HS,HC) ...
```

```
RT desc "(R)obot arm (T)ouches
         the operator"
  requiresNOf (1|HRW,HS,HC)
  mitPreventsMit (RC)
...
```

Figure 3: Specifications of the two risk factors HRW and RT

"Robot arm Touches the operator" (RT). Accordingly, the specification for RT in Figure 3 uses the constraint `requiresNOf (1|HRW,HS,HC)` to refer to its potential causes: at least one of HRW, HS, and HC.

4 Action-based Risk Modelling. In YAP, actions of both the controlled process and the safety controller can be characterised using multiple weights that order the choice alternatives when numerical solvers search the MDP policy space. YAP converts these weights into action reward structures used for quantitative verification in tools such as PRISM.

Like for the actions of the safety controller, one can specify such characteristics for the actions of any actor in the process. This is done in YAP by providing a `weight` structure (Listing 2). For action-based risk modelling, parameters prefixed with `risk_`, say `risk_f` for a factor f, are used to accumulate risk in reward calculations for the underlying MDP. The corresponding action rewards are guarded by f, requiring that f is active. For example, if `weights` provides a `risk_f` entry for the process action a then a is rewarded only if it is taken in a state where f holds [15].

Connecting the YAP and PRISM Models. Crafting the YAP model and generating a controller model to be integrated into a process model require the engineer to have substantial knowledge of the process model. For example, in order for YAP to inject valid synchronisation commands into existing PRISM

Listing 2: Structure for specifying risk values on a per-action basis

```
Weights rewards {   guard      risk_HC;
  // robotArm
r_moveToTable:      ""         "5";
...
  // welder
rw_weldStep:        ""         "10";
rw_leaveWelder:     ""         "5";
  // humanOp
...
h_approachWeldSpot: ""         "7";
h_exitCell:   "notif=leaveArea" "0"; }
```

Listing 3: Connecting YAP and PRISM models via YAP agents and PRISM modules

```
1  robotArm type AGENT
2    validActs="exchWrkp|welding|off";
3  welder type AGENT
4    validActs="welding|idle|off";
5  safetyCtr type CONTROLLER
6    notif="[ok..resetCtr] init ok"
7    notif_leaveWrkb="bool init false";
```

Listing 4: Two risk gradient matrices

```
1  Distances act {
2   off:      0;
3   idle:     1 0;
4   exchWrkp: 3 2 0;
5   welding:  5 4 2 0; }

6  Distances safmod {
7   normal:  0;
8   hguid:   -2  0;
9   ssmon:   -1  1    0;
10  pflim:   -2  0   -1  0;
11  srmst:   -3 -1   -2 -1  0;
12  stopped: -4 -2 -3 -2 -1 0; }
```

modules, one has to specify the activities the respective actors are involved in.[2] This is done by providing a `validActs` parameter taking values of the form `act1|act2|...|actn`. For example, in Listing 3, the **robotArm** is involved in the three activities exchWrkp, welding, and off, the **welder** in welding, idle, and off. If `validActs` is not provided then YAP assumes that the actor can be involved in any of the specified activities. Furthermore, additional module-specific (i.e., local) variables used by the safety controller can also be declared as part of an item specification.

5 **Situational Risk Modelling.** In YAP, we can not only model risk in terms of factors but also in terms of the behavioural modes the application or controlled process can be in. Examples of such modes are activities and safety modes (Sect. 5.4) as used and standardised in application domains such as HRC [11]. Of course, we allow behavioural modes to change during operation. Moreover, we often do not exactly know about the absolute risk level of a certain mode in a certain situation. Thus, YAP offers the possibility to define *risk gradient matrices* that only capture expected changes in the risk level when changing from one mode into another. We only consider symmetric changes and, thus, use skew-diagonal matrices reduced[3] to their lower left triangles. Examples of these matrices are shown in Listing 4.

To use such matrices for maximisation in YAP's pGCL generator,[4] we associate a positive gradient with an improvement of the risk level and, vice versa, a negative gradient with a worsening of the risk level.[5] For example, the act matrix tells us that a transition from the activity welding to the activity exchWrkp improves the risk level by 2. As a further example, the safmod matrix states that a transition from the safety mode srmst (i.e., safety-rated monitored stop) to the safety mode ssmon (i.e., speed and separation monitoring) worsens the risk level by -2.

YAP uses these matrices to calculate act- and safmod-updates of the set of guarded commands generated for the safety controller from the given factor specifications [6]. In YAP, risk can be evaluated both based on transitions through the risk space (i.e., from one risk state to another) and on a wider situational basis (i.e., based on transitions from one activity or safety mode to another). Note that while in step 4, actions are associated with an *absolute risk* value, in step 5, mode and activity changes are associated with a *risk gradient*, a relative measure stemming from a pair-wise comparison of modes and activities. Whereas for individual actions, the YAP user needs to agree on a single global scale for risk assessment,

[2] That could also be achieved by parsing the process model (here, the PRISM file) but is beyond YAP's current functionality.
[3] For readability, one can provide the symmetric upper right triangle as well. However, YAP internally mirrors the values from the lower left to the upper right to ensure skew-diagonality. [4] A description of YAP's algorithms is out of scope of this tool paper. [5] This choice should not be too counter-intuitive as one can associate negative numbers with something undesirable.

the higher complexity of risk assessment for modes and activities is taken account of by gradients.

5.3 6 Stochastic Modelling

Probabilistic choice in the process model set up in Sect. 5.1 (e.g. an MDP modelled in PRISM's pGCL) can capture a variety of adverse and critical stochastic events, such as human errors, sensor failures, actuator perturbation and failures, and mishaps.

Human Errors. We consider, for example, intrusion of **humanOp**, whether intentional or erroneous, into the work cell when not allowed:

```
[hi_mayEnterCell] !dntFlg_enterCell & !mntDone & !CYCLEEND
    // deontic flag and end−of−cycle check
   & wps!=empty & hact=idle & (hloc!=inCell & hloc!=atWeldSpot) & !wpfin
    // action physically possible / feasible / reasonable
   -> ((prm_enterCell&req_enterCell)?.9:.2):(dntFlg_enterCell'=true)
    // action enabled if (90) / ifnot (10) allowed / required
     +((prm_enterCell&req_enterCell)?.1:.8):true;
    // action notenabled if (10) / ifnot (90) allowed / required
```

This listing shows a two-staged[6] guarded command modelling the operator's internal decision, not their physical action. The flag `dntFlg_enterCell` is *deontic* in the sense that it *enables but not triggers* the action h_enterCell of the operator actually entering the work cell. The activation of this flag is handled by a *conditional probabilistic choice*: the condition `prm_enterCell` distinguishes between probabilistic intrusion in states where the human operator is *permitted* to enter the work cell and probabilistic intrusion where the operator is not allowed to enter. In case of permission, `dntFlg_enterCell` is set to `true` with a 90% chance and, in case of denial, only with a 20% chance. This is a way of saying that the operator commits a human error in 20% of the times *when they should not* enter the work cell. The predicate `req_enterCell` is not used in this particular model. It just indicates another potentially useful state selection mechanism. The condition and the deontic flag can be omitted if the probabilities are universal and we do not need to separate a *physical* action (e.g. movement of the operator) from a *logical* action (e.g. change of the operator's mind). In this model, we treat operating errors and malicious misuse in the same way, however, a distinction can be necessary in other contexts.

Sensor Failures. The separation of physical and logical human actions allows us to synchronise sensor actions with physical actions, for example, to model a range detector with a 5% chance of failure:

```
[h_enterCell] true -> .95:(rngDet'=near)+.05:true;
```

h_enterCell is an event with two *synchronous* actions, the physical action of **humanOp** entering the work cell and the logical action of the range finder in **sensorUnit**. Synchronisation establishes real-time behaviour despite the deontic nature of pGCL. This sensor failure is modelled by the range finder signalling `near` to **safetyCtr** only in 95% of the cases where **humanOp** actually enters the cell.

Actuator Perturbation and Failures. We have not modelled any perturbations in the HRC case study. However, an obvious entry point for such phenomena would be the actions of **robotArm**. For example,

[6] Such commands do not increase the expressiveness of pGCL as they can be expressed by a set of ordinary commands. However, they increase convenience by allowing what can be called conditional probabilistic choice.

the action r_grabLeftWorkpiece could be extended by a probabilistic update modelling the fact that grabbing a work piece fails in a certain fraction of trials with a work piece not in the grabber and/or still in the work piece support, or whatever outcome seems realistic.

Mishaps after Critical Events. Based on conditional probabilistic choice, similar to human error modelling, we use probabilities to capture the fact that from activated critical events and certain actions, a transition into a mishap state is possible. Consider the example:

```
HC desc "(H)uman (C)lose to active welder and robot working"
  ...
  mis="h_exitCell" // mishap possibly initiated by the action h_exitCell
  prob=0.05 // probability of mishap is 5 percent if unrecognised OR active and not mitigated
  sev=5; // severity of the mishap is of class 5
```

The parameter `prop=0.05` defines the probability of a `mis`hap from the action h_exitCell in case of an activated HC to be 5%. The parameter `sev=5` associates the impact from such a mishap to the exemplary impact class 5, which can, for example, mean "high". In other words, if **humanOp** wants to perform the action h_exitCell if the factor HC is active (i.e., *HC*) then, with a 5% chance, the outcome will be the mishap *HC*, a specimen of the `MISHAP` state. The resulting pGCL fragment generated by YAP:

```
[h_exitCell] true ->
   ((!HCp=mis & (CE_HC | RCE_HC))?0.05:0):(HCp'=mis)
   +((!HCp=mis & (CE_HC | RCE_HC))?0.95:1):true;
[h_placeWorkpieceLeft] true ->
   ((!HRWp=mis & (CE_HRW | RCE_HRW))?0.01:0):(HRWp'=mis)
   +((!HRWp=mis & (CE_HRW | RCE_HRW))?0.99:1):true;
```

HC can, for instance, be a welding spark injuring the operator, or the **robotArm** hitting them, both encoded in a corresponding risk/severity reward generated by YAP:

```
[h_exitCell] (!HCp=mis & (CE_HC | RCE_HC)) : 5.0;
```

One downside of this way of reward modelling in PRISM is that the reward is paid independent of the outcome of h_exitCell. A solution not chosen here would be to introduce an intermediate state with the disadvantage of doubling the state space each time such a construction is chosen.

5.4 Mitigation Modelling with YAP

7 **Factor-based Mitigation Modelling.** The basic factor model allows mitigation in one (direct) or two stages (indirect). For controller synthesis, we focus on the two-staged approach, which suggests the specification of modes for detection, mitigation, and resumption.[7] For example, the modes for the factor HRW are referred to from its specification:

```
HRW desc "(H)uman arm and (R)obot on shared (W)orkbench" ...
  guard "hACT_WORKING & rloc=sharedTbl & hloc=sharedTbl"
  detectedBy (SHARE.HRWdet)
  mitigatedBy (PREVENT.HRWmit)
  resumedBy (.HRWres) ...
```

and detailed in the `Application` fragment:

```
mode HRWdet
  guard "hACT_WORKING & rloc=sharedTbl & lgtBar=true"
  embodiedBy cellObSys;
```

[7] We omit alleviation modes for the sake of simplicity of this guide.

```
4  mode HRWmit desc "safety-rated monitored stop"
5    update "(notif_leaveWrkb'=true)" // safety function on
6    target (safmod=srmst) // safety mode on
7    embodiedBy robotArm
8    disruption=5 nuisance=1 effort=0.5;
9  mode HRWres
10   guard "hloc!=sharedTbl" // checking for hazard removal
11   update "(notif_leaveWrkb'=false)" // safety function off
12   target (safmod=normal); // safety mode off
```

Detection, mitigation, and resumption modes make use of the process model. For example, HRWdet uses the formula `hACT_WORKING` (explained below) and the module variables `rloc` and `hloc` (the locations of the robot arm and the operator) in an embedded `guard` expression. `update` in HRWmit specifies the assignment of `true` to the module variable `notif_leaveWrkb` to notify the operator to leave the workbench. Being a generic form of update, `target` specifies mode switching preferences that, when unsafe, will be overridden by YAP using the gradient matrices (Listing 4).

For example, HRWmit switches to the mode "safety-rated monitored stop" (`smrst`) and HRWres back to the `normal` mode. However, **safetyCtr** would only switch to the `normal` mode if this is acceptable in the risk state to be reached. Hence, each of these modes will be enhanced by information about activities and risk states and translated into one or more guarded commands for being integrated into the process model. Note that there are also *embodiment* references to the items **robotArm** and **cellObSys**, the latter being the overall sensor system of the work cell.

Connecting the YAP and PRISM Models. It is convenient to reuse formulas in several places. For this purpose, YAP allows their definition in the `Application` sections of the YAP model:

```
1 Application cobot {
2   hACT_WORKING = "(ract=exchWrkp | ract=welding | wact=welding) & safmod=normal";
3   ...
4   hST_HOinSGA = "hloc=inCell | hloc=atWeldSpot";
5   hFINAL_CUSTOM = "wpfin & wps=empty & !reffocc & mntDone"; ... }
```

For example, the predicate `hACT_WORKING` includes activities where actors are effectively working. `hST_HOinSGA` is a shortcut specifying states where the operator is in the safeguarded area. `hFINAL_CUSTOM` refines `CYCLEEND` (Sect. 5.1), the termination of a process cycle, in our example, the end of a manufacturing cycle in the work cell. This predicate is used in the reduction of cyclic end components in order for optimal MDP policy search algorithms to work correctly [15].

8 Adding Mitigation Alternatives. Factor specifications allow one to provide several mitigation and resumption options and characterise their properties using *risk and performance* parameters. For example, the factor HS (Human in Safeguarded area while robot working or welding) refers to three mitigation options in its `mitigatedBy` directive:

```
1   HS desc "(H)uman in (S)afeguarded area while robot working or welding"
2     guard "hACT_WORKING & (hloc=inCell | hloc=atWeldSpot)"
3     detectedBy (SHARE.HSdet)
4     mitigatedBy (.ssmon,.srmst,.stopped)
5     resumedBy (.HSres)
```

We model the options ssmon, srmst, and stopped in more detail in Listing 5.

Listing 5: All modes including for the factor HS three mitigation options

```
mode HSdet desc "range detector"
  guard "hACT_WORKING & (rngDet=near |
      rngDet=close)";
mode ssmon
  desc "speed/separation monitoring"
  target (safmod=ssmon)
  disruption=9 nuisance=9 effort=8;
mode srmst
  desc "safety/rated monitored stop"
  target (safmod=srmst)
  disruption=5 nuisance=9 effort=5;
mode stopped
  desc "protective emergency stop"
  target (safmod=stopped)
  disruption=2 nuisance=6 effort=3;
mode HSres
  guard "!hST_HOinSGA" // check hazard removal
  target (safmod=normal);
```

Table 1: Placeholders recognised by YAP and to be inserted into the process model for substitution

Placeholder	Description
`<%YAP#TYPES>`	Inject global type declarations
`<%YAP#PREDICATES>`	Inject global definitions
`<%YAP#CONTROLLER>`	Inject controller module
`<%YAP#REWARDS>`	Inject reward structures
`<%YAP#MODULEHOOK(m)>`	Add data and command definitions to application module `m`

5.5 Verified Controller Synthesis

9 Action-based Performance Modelling. Along with the three mitigation options specified in Listing 5, we provide estimates for their *disruption* of the manufacturing process, for their *nuisance* of operators, and for the *effort* required for their execution. In addition, for each action in the process model (Sect. 5.1), one can provide a *guard* and several columns of optimisation parameters (e.g. `prod`, `eff_process_time`, `risk_HC`). Each such column is converted into an action reward structure.

```
Weights rewards {
          guard   prod eff_process_time;
  // robotArm
  r_moveToWelder: ""       "h"    "2*macro";
  ...
  // welder
  rw_weldStep:    ""       "h"    "3*macro";
  rw_leaveWelder: ""       "h"    "macro";
  // humanOp
  h_start:    ""       "l"    "macro";
  ...
  h_enterCell:    ""       "none" "none";
  h_exitCell: "notif=leaveArea""none""none";
}
```

Moreover, as shown in these two examples, one can also use parameters defined elsewhere (e.g. `macro`, `h`, `none`) instead of using literal numbers in place. One may provide several `weights` structures across the activity model. However, they will all be merged into one central "database" containing all columns found in the given structures.

10 Design Space Generation. The process model has to be instrumented with *placeholders* to be substituted with model fragments generated by YAP. Such placeholders take the form `<%YAP#X>` where *X* is the placeholder name.[8] Table 1 lists placeholders currently supported by YAP.

The output of YAP, for this use case, is an MDP, with its non-deterministic choice representing the decision space of all actors in the work cell. We call the decision space of the safety controller—as one

[8] The placeholders will need to be commented in order to not interfere with the semantics of the process modelling language (cf. Javadoc in Java). In PRISM, we therefore use `//<%YAP#X>`.

of these actors—the *design space*. This design space and the decision space of the other actors are used for optimal controller synthesis.

With the model constructed according to the steps 1 to 9 in Figure 1, we use YAP to generate and inject the controller into the *process model* (Sect. 5.1):

```
yapp -m model.yap -t target-template.xyz -o output/model.xyz \
     -f prism -d multi-event-concurrent --synthesise controller
```

With the output format switch `-f prism`, YAP creates three artefacts in this step:

1. An MDP in PRISM's pGCL used as the *design space* for PRISM's search for an *optimal policy*—a DTMC—representing the abstract controller (file `model.prism`).
2. A list of probabilistic computation tree logic (PCTL) properties to be verified of the design space by PRISM in step 11 (Sect. 5.5, file `model.props`).
3. A list of PCTL properties to be verified in step 12 (Sect. 5.5) of any policy found by PRISM (file `model_pol.props`).

The following pGCL fragment, generated by YAP, shows the design space for *switching into a safety mode* triggered if a particular hazard (e.g. HC, HRW, HS) has been detected:

```
1 [si_HCSrmstIdleVissafmod] !CYCLEEND & safmod=normal & HCp=act -> (safmod'=srmst);
2 [si_HCStOffAudsafmod] !CYCLEEND & safmod=normal & HCp=act -> (safmod'=stopped);
3 [si_HCStOffVissafmod] !CYCLEEND & safmod=normal & HCp=act -> (safmod'=stopped);
4 [si_HRWmitsafmod] !CYCLEEND & safmod=normal & HRWp=act -> (safmod'=srmst);
5 [si_srmstsafmod] !CYCLEEND & safmod=normal & HSp=act -> (safmod'=srmst);
```

The following pGCL fragment, generated by YAP, highlights the part of the controller design space allowing the controller to *switch off mode-specific safety functions* (lines 1-2) and *resume to a less restrictive safety mode* (lines 3-4):

```
1 [si_HCres2fun] !CYCLEEND & HCp=mit & !CE_HC & notif=leaveArea & !hST_HOinSGA -> (notif'=ok);
2 [si_HCresfun] !CYCLEEND & HCp=mit & !CE_HC & notif=leaveArea & !hST_HOinSGA -> (notif'=ok);
3 [si_HRWressafmod] !CYCLEEND & safmod=normal & HCp=inact & HSp=act & WSp=inact & HWp=inact &
      RCp=inact & (RTp=mit | RTp=sfd) & HRWp=mit & hloc!=sharedTbl & (notif_leaveWrkb=false)
4      -> (safmod'=ssmon)&(HRWp'=sfd);
```

11 Design Space Verification and Optimal Controller Synthesis. In this example, we use PRISM for the verification of the MDP and the synthesis of an MDP policy. The use of PRISM and the processing of its output is out of scope of this paper and, therefore, not explained here. Particularly, the formulas presented below contain PCTL operators [14] and other PRISM query language [21] primitives that are assumed to be familiar to the YAP user interested in PRISM-based controller synthesis with YAP.

Synthesising Optimal Policies from an MDP. For optimal policy synthesis (here, the synthesis of DTMCs), we use the command

```
prism output/model.prism -pctl '<query>' -s \
     -exportadvmdp poloutdir/model-adv.tra \
     -exportstates poloutdir/model-adv.sta \
     -exportprodstates poloutdir/model-adv.pst \
     -exportlabels poloutdir/model-adv.lab .
```

Table 2: Formulas generated by YAP for custom property specification

Formula	Description
E	"detector" predicate for the critical event *E*
RCE_*E*	"ground truth or reality" predicate for the critical event *E*
ANYOCC (OCE)	*universal* detector predicate, true if *any* critical event is true
ANYREC (RCE)	universal ground truth counterpart of ANYOCC
ANY (CE)	true if any CE has occurred whether or not detected
ACCIDENT	true if any factor f is in its mishap phase *f*
MISHAP	true if ACCIDENT or if any final factor (e.g. an incident) is activated (i.e., in phase *f*)
SAFE	true of any state that is neither a CE nor a mishap
FINAL (CYCLEEND)	true if hFINAL_CUSTOM (Sect. 5.4) is reached

PRISM will generate one or more policy files (with names and suffixes according to `model-adv[1-n].pst,sta,tra,lab`) from the file `output/model.prism` and compliant with the optimisation query `<query>`, for example,

```
multi(R{"effort"}max=? [ C ], R{"nuisance"}max=? [ C ]) ,
```

and places these files in `poloutdir/`. This query searches for all policies that Pareto-maximise effort and nuisance (`Rmax[C]`) as explained in Sect. 5.5. PRISM enumerates such policies as a list of value pairs holding the results of the cost function defined by this query. Within PRISM's GUI, one can visualise these pairs as a Pareto front. We calculated a Pareto front for the present example in [6].

To include the verification of safety properties at this stage in the procedure, we can use a combination of a single optimisation query and several constraints, for example,

```
multi(R{"prod"}max=? [ C ], R{"risk_sev"}<=s [ C ]) and
multi(R{"risk_sev"}<=s [ C<=t ], P<=p [ F "ANY" ]) .
```

The first property maximises the productivity of the work cell (`Rmax[C]`) as long as the accumulative bound on action rewards (`R<=s[C]`) for `risk_sev` stays below a user-defined level *s*. The second property combines a time-bounded version (`C<=t`) of the latter constraint with the probability-bounded reachability (`P<=p[F]`) of ANY of the modelled factors, for user-defined bounds *t* and *p*. Beyond ANY, YAP generates further shortcut formulas (into the file `model.prism`) that can be used in PCTL properties as shown above. These state formulas are listed in Table 2.

12 Controller Verification: Checking the Generated Policies (DTMCs). As already mentioned in Sect. 4, the separation into the two verification steps 11 and 12 and the corresponding property lists is due to restrictions in the combinations of properties that can be checked by PRISM in one go. The policies are given in the form of DTMCs and can, thus, be further checked with the command

```
prism -importstates poloutdir/model-adv.sta \
      -importlabels poloutdir/model-adv.lab \
      -importtrans poloutdir/model-adv1.tra -dtmc \
      -pctl "<prop>" -gs >poloutdir/model-adv-checks.txt .
```

For example, for `<prop>` we applied

```
filter(avg, P=? [ !"ACCIDENT" W "SAFE" ], "ANYREC" & !"MISHAP")
```

to determine the average probability (`filter(avg, P=? [...], ...)`) of accident freedom until reaching a safe state (`!"ACCIDENT" W "SAFE"`) when starting from any reachable hazardous state (`"ANYREC" \& !"MISHAP"`). As already mentioned in Sect. 5.5, with the file `model.props`, YAP suggests a range of properties to be checked of a policy. See [6, Tab. III] for a selection of properties.

Further Processing of the MDP and DTMC. The abstract controller consists of the list of states of the MDP respectively the DTMC, for example,

```
State:(...  HRWp, notif_leaveWrkb, ...)
510:  (...  1, false, ...)
511:  (...  1, true, ...)
```

and all transitions describing the decisions (accordingly, with probabilistic outcomes), for example,

```
510 511 1 si_HRWmitfun
```

This transition, going from state 510 to 511 and labelled with the action `si_HRWmitfun`, notifies the operator, with probability 1,[9] to leave the workbench in case of an activated factor HRW.

For visualisation and model debugging, the MDP can be converted into a `dot` file with

```
prism output/model.prism -exporttransdotstates rel.dot .
```

This file can be used with GraphViz tools such as `dot`.[10] Based on GraphViz, YAP provides rudimentary facilities for the visualisation of the generated policy. Such a visualisation can be useful in the direct debugging of DTMCs with a state space of up to a size of around 1000 states.

6 Discussion and Outlook

From Abstract to Concrete Policies. The safety controller in its abstract form is represented by the calculated policy, a DTMC with state space Σ and action set A. Σ and A are results of combining the risk state space, generated by YAP from the factor set F, and the mitigation actions, filed in the YAP model, with the process model. Each state-based memory-less deterministic policy π can then be represented by a map $\pi\colon \Sigma \to A$. As shown before, the transition relation of the policy is provided by PRISM as a list of $(state, action, probability, state)$-tuples (cf. Sect. 5.5). The safety controller, part of such a policy, is a list of transitions that, at the concrete level, would again be guarded commands of the form:

$$\underbrace{[\text{controller action}]}_{\text{event}}\ \underbrace{\text{process \& risk state}}_{\text{guard}} \to \underbrace{\text{mode \& activity switch, safety function}}_{\text{update}}$$

PRISM's output can be used to translate this abstract policy representing the discrete-event controller into a concrete policy. This translation involves two essential steps:

- The translation of the abstract states into concrete guard conditions, and
- the translation of the updates into low-level procedures generating control inputs to the process.

Part of ongoing research is the corresponding refinement of this transition relation into an automaton that can run on, for example, an autonomous machine platform or the robot operating system[11] (ROS). We will investigate how environments such as Isabelle/UTP [4] and ROBOTOOL [20] can be used to verify and deploy safety controllers derived with the help of YAP. Isabelle/UTP provides a generic framework for model verification and ROBOTOOL an environment for rigorous robotic software development.

[9] The controller in this example is fully deterministic but, conceptually, we can also design randomised controllers with our approach. [10] See `https://graphviz.org`. [11] See `https://www.ros.org`.

Safety Properties and Safety Controller. A safety property states that "something will *not* happen" and, thus, is a property whose violation can be observed in finite time [16]. In many applications, "something" refers to a CE that cannot be avoided by a careful redesign of the process and, therefore, violations have to be accepted to a certain extent. A safety controller typically includes a safety monitor responsible for detecting such violations at run-time [17] and an active component influencing the monitored process in a way that the safety property is established again. In other words, the "violation counter" is restored. In such applications, we therefore substitute the verification of the original safety property by

- a response property [19] (formalising successful mitigation and resumption as a finite response to the detection of an endangerment) to be verified of the process integrated with the controller design space (cf. 11), and
- another safety property (cf. 12, formalising the absence of undesired consequences of the aforementioned violations) whose probability of being violated must not exceed a certain bound, by virtue of the safety controller when working correctly.

Re-Interpretation of Activity Graphs for Synthesis. We may want to allow several actors to concurrently carry out actions in any of the activities of the process. Therefore, it seems useful to associate a coloured Petri net (CPN) [12] semantics to activity graphs (e.g. Figure 2). CPNs offer a more flexible way of modelling concurrency compared to the parallel composition [8] used in PRISM's pGCL. Specifically, in a CPN, the places could represent activities and the movable labels the actors. Then, a placement of these labels, that is, a marking, indicates the activities actors are performing at a point in time. A transition in a CPN can move any number of labels between the activities, meaning the actors involved in that transition concurrently finish their current activities and start new activities. However, a pGCL guarded command in PRISM can either move one label or as many labels as there are actors participating in a synchronous event. As a part of our future work, we will investigate how the explicit approach to concurrency in CPNs improves the usefulness and flexibility of the activity model.

7 Conclusions

This paper provides a hands-on and tool-focused guide to a novel approach to the design, verification, and synthesis of safety controllers from hazard analysis and risk assessment as previously published in [6]. We also discuss a range of modelling decisions (e.g. identifying parameters, decomposing behaviour, integrating probabilistic choice) to be made when devising such controllers. The proposed step-wise and tool-supported workflow aims at supporting verification engineers in transforming data from hazard analysis and risk assessment into a verifiable controller model and, thus, contributes to recent and practically relevant challenges (e.g. [1, Challenges OC1, OC2, and OC4]).

Among the next steps of technical research are the improvement of the synthesis facilities, the evaluation of alternatives to PRISM, and the development of an integration with robotic platforms (e.g. ROS, digital twin environments[12]) and tools (e.g. RoboChart[20]) to automate controller deployment.

Acknowledgements. This research was funded by the Lloyd's Register Foundation under the Assuring Autonomy International Programme grant CSI:Cobot. I am greatly indebted to Radu Calinescu for many inspiring discussions and for encouraging me to implement some of the described YAP enhancements.

[12] See, e.g. `https://github.com/douthwja01/CSI-cobotics`.

References

[1] Radu Calinescu, Javier Camara & Colin Paterson (2019): *Socio-Cyber-Physical Systems: Models, Opportunities, Open Challenges*. In: *5th ICSE Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, IEEE/ACM, pp. 1–6, doi:10.1109/sescps.2019.00008.

[2] Radu Calinescu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli & Tim Kelly (2018): *Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases*. *IEEE Transactions on Software Engineering* 44(11), pp. 1039–1069, doi:10.1109/tse.2017.2738640.

[3] Clifton A. Ericson (2015): *Hazard Analysis Techniques for System Safety*, 2 edition. Wiley.

[4] Simon Foster, Frank Zeyda & Jim Woodcock (2015): *Isabelle/UTP: A Mechanised Theory Engineering Framework*. In: *UTP*, Springer, pp. 21–41, doi:10.1007/978-3-319-14806-9_2.

[5] Mario Gleirscher (2020): YAP *Against Perils: Application Guide and User's Manual*. University of York and Technical University of Munich. Available at `http://gleirscher.de/yap/`.

[6] Mario Gleirscher & Radu Calinescu (2020): *Safety Controller Synthesis for Collaborative Robots*. In: *Engineering of Complex Computer Systems, 25th International Conference (ICECCS), 28 - 31 October 2020, Singapore*, pp. 1–12. Available at `https://arxiv.org/abs/2007.03340`. In press.

[7] Mario Gleirscher, Radu Calinescu & Jim Woodcock (2020): *Risk Structures: A Design Algebra for Risk-Aware Machines*. Working paper, Department of Computer Science, University of York, York, UK. Available at `https://arxiv.org/abs/1904.10386`.

[8] Charles A. R. Hoare (1985): *Communicating Sequential Processes*. Int. Series in Comp. Sci., Prentice-Hall. Available at `http://www.usingcsp.com`.

[9] IEC 61508 (2011): *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. Standard, The 61508 Association. Available at `http://www.61508.org/`.

[10] IEC 61882 (2016): *Hazard and operability studies – Application guide*. Standard 61882, IEC. Available at `https://webstore.iec.ch/publication/24321`.

[11] ISO/TS 15066 (2016): *Robots and robotic devices – Collaborative robots*. Standard, Robotic Industries Association (RIA). Available at `https://www.iso.org/standard/62996.html`.

[12] Kurt Jensen & Lars M. Kristensen (2009): *Coloured Petri Nets*. Springer, Berlin Heidelberg, doi:10.1007/b95112.

[13] John Knight (2012): *Fundamentals of Dependable Computing for Software Engineers*. Chapman and Hall/CRC, doi:10.1201/b11667.

[14] Marta Kwiatkowska, Gethin Norman & David Parker (2007): *Stochastic Model Checking*. In M. Bernardo & J. Hillston, editors: *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM)*, *LNCS* 4486, Springer, pp. 220–70, doi:10.1007/978-3-540-72522-0_6.

[15] Marta Kwiatkowska, Gethin Norman & David Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-time Systems*. In G. Gopalakrishnan & S. Qadeer, editors: *23rd International Conference on Computer Aided Verification (CAV)*, *LNCS* 6806, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_47.

[16] Leslie Lamport (1977): *Proving the Correctness of Multiprocess Programs*. *IEEE Trans. Software Eng.* 3(2), pp. 125–43, doi:10.1109/TSE.1977.229904.

[17] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. *Journal of Logic and Algebraic Programming* 78(5), pp. 293–303, doi:10.1016/j.jlap.2008.08.004.

[18] Nancy G. Leveson (2012): *Engineering a Safer World: Systems Thinking Applied to Safety*. Engineering Systems, MIT Press, Cambridge, Mass., doi:10.7551/mitpress/8179.001.0001.

[19] Zohar Manna & Amir Pnueli (1995): *Temporal Verification of Reactive Systems: Safety*. Springer, doi:10.1007/978-1-4612-4222-2.

[20] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis & Jim Woodcock (2019): *RoboChart: modelling and verification of the functional behaviour of robotic applications*. Software & Systems Modeling, doi:10.1007/s10270-018-00710-z.

[21] Dave Parker, Gethin Norman & Marta Kwiatkowska (2019): *PRISM Model Checker*. Available at `http://www.prismmodelchecker.org/manual/`.