Reversible Computation *vs.* Runtime Adaptation in Industrial IoT Systems

Duncan Paul Attard, Keith Bugeja, Adrian Francalanza(✉), Marietta Galea, Gerard Tabone, and Gianluca Zahra

University of Malta, Msida, Malta
{duncan.attard,keith.bugeja,adrian.francalanza,marietta.galea,
gerard.tabone,gianluca.zahra.16}@um.edu.mt

Abstract. This paper presents a comparative study between two software engineering techniques, reversible computation and runtime adaptation, in the context of industrial IoT. We frame our comparison around a representative Industry 5.0 shop floor case study that focuses on the high-precision manufacturing of integrated circuits. The case study identifies four error scenarios that can arise in typical shop floor operations and evaluates how reversible computation and runtime adaptation address them, highlighting the strengths and limitations of each approach.

Keywords: reversible computation · runtime adaptation · industrial IoT · industry automation

1 Introduction

Industrial automation has seen significant recent advancements, driven by the robotisation developments of Industry 4.0 [48], and further extended to collaborative robots (or *cobots*) under Industry 5.0 [29]. These advancements have made production automation increasingly accessible to small and medium-sized enterprises (SMEs). This uptake has enabled production automation in low-volume and small-batch manufacturing, sectors that were previously considered beyond the reach of such technologies [28,78]. Indeed, achieving lean small-batch manufacturing through automated production is now considered central to the future of industrial development and competitiveness, particularly in high-precision and safety-critical domains [8], *e.g.*, microelectronics, automotive, and aerospace.

Industrial IoT (IIoT) plays a central role in enabling the *sensor-driven* infrastructure underpinning Industry 4.0 and 5.0 [9,63]. It facilitates real-time monitoring and control in robotised environments, where safety-critical feedback mechanisms govern human-robot interaction [11]. *Robotisation* fundamentally relies on the digitisation of the physical factory shop floor and assembly line. This process is typically governed by two requirements [11,64]:

Funded by the IPCEI-UM (No: E24LO17-01) project under Malta Enterprise.

C. A.Mezzina and A. Schmitt(Eds.): Jean-Bernard Stefani Festschrift, LNCS 16065, pp. 199–217, 2026.
https://doi.org/10.1007/978-3-031-99717-4_11

Automation robustness requires robotised manufacturing to anticipate, detect and avert uncertainties, errors, and abnormalities. Not meeting these conditions could lead to equipment damage, production defects, and in the case of cobots, human injuries and fatalities.

Automation flexibility requires manufacturing robots to be resilient to errors and unexpected situations. As much as possible, production should remain uninterrupted in order to reap the benefits of automation and maximise production gains.

There are various approaches that are used to attain high levels of automation robustness and flexibility [62]. At one end of the spectrum, *static* solutions augment the digital models of shop floors and assembly lines. Static solutions incorporate uncertainty and error occurrences via techniques such as Monte Carlo simulations [18] and machine learning [41]. The latter offline error prediction techniques then permit the design of robust production processes that tolerate uncertainty [23,52]. However, static approaches are still susceptible to break down whenever unexpected errors fall outside predicted scenarios [64]. In addition, static solutions often require vast amounts of resources to model all eventualities and work adequately (*e.g.* data gathered for machine learning purposes and computational power required to analyse the data). These prohibitive upfront costs can render them beyond the reach of SMEs [59]. At the other end of the spectrum are the sensor-based *dynamic* approaches, where set-ups such as vision-based control systems [71,87], often fused with other sensory inputs *e.g.* LiDARs, provide greater automation flexibility in handling uncertainty through dynamic intervention [27]. Yet, this enhanced flexibility comes at a higher cost, such as, installing additional (expensive) high-precision sensors [73,85], which can be prohibitive to many SMEs. Additionally, the runtime computational overhead required may exceed the capabilities of the robot hardware, where determining the right course of action on-the-fly under tight latency constraints may be intractable [59]. Due to these constraints, full-blown dynamic techniques might ultimately resort to more standard graceful degradation approaches [33] for a number of situations.

There are other techniques that strike a balance between static and dynamic approaches to maintain cost-efficiency, thereby extending production automation to a wider range of enterprises. At the same time, these techniques incorporate static information about the shop floor digital model to alleviate the need for high-precision hardware and minimise the runtime computation needed to effect interventions in a timely and precise manner [59]. Runtime Monitoring [17,37] and Reversible Computing [54,75] are two prominent techniques with these characteristics. This paper compares these two approaches in order to understand their commonalities, relative advantages, and limitations. We also investigate how the two techniques can be used to benefit one another. This comparison is given in the context of a representative Industry 5.0 high-precision shop floor case study that manufactures integrated circuits (ICs).

The paper is structured as follows. Section 2 outlines the main characteristics of our shop floor case study, and Sect. 3 summarises the key elements of

the runtime monitoring and reversibility techniques. Section 4 considers a selection of error situations that may arise in our shop floor case study and argues how these can be handled by the respective techniques. It also discusses how the two approaches can be combined to leverage their complementary strengths. Section 5 concludes. Our survey does not assume prior knowledge of the aforementioned software engineering techniques. Familiarity with industrial IoT and the challenges in Industry 4.0 and 5.0 is beneficial.

2 Industry 5.0 Factory Shop Floors

Industry 4.0 and 5.0 smart factories comprise fast-paced shop floors where self-driving vehicles and other machinery need to adapt to dynamic changes in real-time. Autonomous mobile robots (AMRs) meet this need by using various sensing devices, *e.g.* cameras, LiDARs, and inertial measurement units [77], to perceive their surroundings, plan optimal navigation routes, and execute real-time obstacle avoidance.

The fleet management system (FMS), or *fleet manager*, coordinates the interaction between different machines on a shop floor, *e.g.* AMRs, high-precision machines, and conveyor systems [43]. It acts as a central coordinator for route planning, movement synchronisation, task assignment, *etc.*, by issuing high-level commands to machines. For instance, the command *'move from A to B via waypoints* $1, 2, 3$*'* plans the route of an AMR between two stations; *'wait at waypoint 3 until the docking station at A becomes unoccupied'* synchronises its movement w.r.t. other machines; and *'pick up object at A'* tasks the AMR with moving objects around the shop floor. A machine interprets fleet manager commands via its onboard planning module that decomposes them into a detailed motion plan. The plan is subsequently converted into fine-grained control instructions, *e.g.* joint trajectories and velocity profiles, and fed to the motion controller, which manages machine actuators in real-time via continuous feedback loops. Modern FMSs function as coordinating hubs that expose APIs to enable integration with external systems, *e.g.* manufacturing execution systems (MESs), quality control, and digital twin platforms. These APIs offer capabilities such as robot coordination, task assignment, and real-time monitoring of mobile robots.

The diagnostics exposed by the FMS APIs may not always offer sufficient capabilities to external supervision systems wanting to provide added robustness, automation flexibility, quality control, and human safety on top of existing functionality. IIoT systems can be deployed as an *overlay network* [81] of bespoke devices attached to the FMS and factory shop floor equipment to collect specific or high-precision data about machinery and its operating environment. For instance, LiDAR sensors affixed to an AMR base can establish equipment safety perimeters; vibration sensors mounted on a robotic arm can detect excessive force during pick-and-place operations; and particulate sensors can reveal increased risks of contamination in sensitive industrial processes. Integrating sensor data through fusion algorithms enhances operational awareness by providing a multi-faceted view of system behaviour. This can enable faster anomaly detection and

more informed, context-aware decision-making [80]. Section 4 describes an external MES that manages the operation of a factory shop floor model (see Sect. 2.1) via a conceptual FMS API and IIoT sensor network overlay.

Factory shop floor designs follow one of three approaches [43]. The *fully-autonomous* approach relegates the fleet manager to a monitoring capacity. Each machine has a *local* planner that plans its routes between stations and resolves conflicts (*e.g.* movement synchronisation, collision avoidance) with other machines [30]. In the *semi-autonomous* approach, machines offload the route planning onto the fleet manager to optimise the overall movement across the shop floor but retain their conflict resolution capability [36]. The *centralised* approach utilises the FMS fully [20]. It computes a *global* route plan for every machine, inherently avoiding conflicts and streamlining navigation on the shop floor. This centralised approach relieves individual machines from autonomous decision-making but comes at the cost of additional computational and communication overhead when dynamic changes in the environment (*e.g.* unforeseen obstacles) oblige the FMS to recompute all routes.

2.1 Integrated Circuit Manufacturing Model

We focus on a subclass of Industry 5.0 manufacturing plants that specialise in automotive-grade ICs, MEMS, and microcontrollers, *e.g.* [72,79,82]. A plant receives wafers containing thousands of ICs for assembly, testing, and packaging. The process begins with wafer dicing, where each chip (or *die*) is separated from the wafer. After dicing, dies are individually mounted onto a package substrate using a die attach material, such as epoxy or solder. This is followed by wire or flip-chip *bonding*, which establishes electrical connections between the die and package substrate wiring. The chip is then *encased* in protective housing, *e.g.* epoxy moulding compound (EMC), to shield it from mechanical damage, moisture, and other contaminants, while also permitting efficient heat dissipation. Lastly, the encapsulated ICs undergo *deflashing* to remove the thin layer of excess epoxy that can bleed between connections during the die encasing process.

Figure 1 depicts a fragment of the manufacturing process described above for a particular shop floor [69]. It consists of *high-precision* machines that perform the final steps of the process, namely wire bonding, EMC die encasing, and deflashing. Stockers are specialised clean storage units that accommodate *die trays* containing batches of dies in various stages of completion. High-precision machines and Stockers have docking stations, which are designated interfaces where die trays are deposited for processing and retrieved after completion. Mobile manipulators (MMs), which are hybrid machines comprising a robotic arm for picking die trays mounted on an AMR base, transport die trays between the high-precision machines and stockers. The FMS orchestrates the operation of the entire shop floor, adopting the centralised approach as described in Sect. 2. Human intervention on the shop floor is limited to specific tasks, such as material handling and replenishment, quality control and inspection, and machine servicing and fault repair. Figure 1 shows machines outfitted with IoT sensors. This use case employs (i) *LiDAR* sensors affixed to MM bases to establish safety

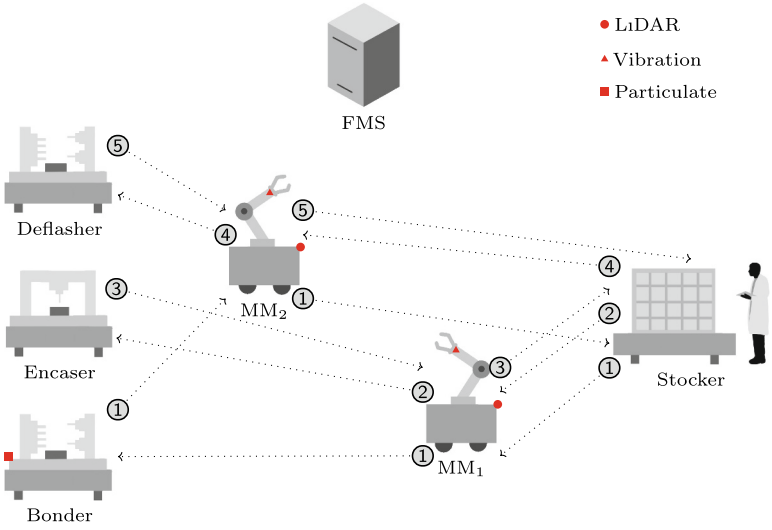

Fig. 1. A typical factory shop floor at an IC manufacturing plant

perimeters around human operators; (ii) *vibration* sensors mounted on robotic arm grippers to detect excessive force during pick-and-place manoeuvres; and (iii) *particulate* sensors to monitor airborne contamination on the shop floor.

Example 1. Figure 1 captures one hypothetical workflow computed by the FMS:

(1a) MM_1 picks a tray of diced dies from the Stocker and delivers it to the Bonder.
(1b) *Simultaneously*, MM_2 retrieves a tray of bonded dies from the Bonder and deposits it into the Stocker.
(2) MM_1 picks the die tray handled by MM_2 in step (1) from the Stocker and delivers it to the Encaser.
(3) MM_1 retrieves the processed tray of encased dies from the Encaser and deposits it into the Stocker.
(4) MM_2 picks the die tray handled by MM_1 in step (3) from the Stocker and delivers it to the Deflasher.
(5) MM_2 picks the processed tray of deflashed dies and deposits it into the Stocker. ■

3 Techniques for Modifying System Behaviour at Runtime

In a non-automated factory shop floor, it is routine for human operators to either report issues or intervene as they arise. Simply replacing humans with their corresponding automated production equivalent robs the shop floor of this

implicit but essential supervision. Thus, to attain an automated shop floor with functionality comparable to the non-automated one, one also needs *automated* issue detection and remediation. Section 1 argues that IIoT systems, such as those described in Sect. 2, are often hard to model fully (*e.g.* due to partial knowledge about the environment or phenomena, such as dust particles, that are too complex to model) or maintain continually (*e.g.* prototype refinement over iterations). More critically, certain system attributes cannot be *analysed* (*e.g.* profiling [42]) or *modified* (*e.g.* restarting failed components [53]) unless the system is running. Runtime monitoring and reversible computation are two techniques that enable the runtime analysis and modification of IIoT systems such as the one presented in Fig. 1.

3.1 Runtime Monitoring and Adaptation

Runtime monitoring (RM) is a family of dynamic analysis techniques [17,40]. It uses *monitors*: machines that incrementally analyse a *finite* prefix of the *execution* (or *trace*) exhibited by the system under scrutiny (SuS) to check whether its behaviour meets prescribed criteria. In RM, these criteria take the form of correctness *properties* [3,4], often expressed in high-level formalisms such as temporal logics [13,17]. Properties encode *prior* knowledge about the SuS as correct behaviour that the system is expected to exhibit. Monitors are synthesised from correctness properties (*e.g.* see [2,12,14,70]) and *instrumented* with the SuS to gather trace events (*e.g.* sensor data) during execution. The resulting set-up captures an *unfolding* model of the SuS [5] compared to full (or complete) models used in techniques such as Model Checking [47].

RM is well-suited to IIoT scenarios where human supervision is *impractical*, *e.g.* cognitive fatigue [84] or continuous manual oversight and intervention [51], or *infeasible*, *e.g.* autonomous warehouses [16] or lights-out manufacturing [74]. It can analyse black-box (proprietary) components (*e.g.* AMRs) without needing access to their internals. Moreover, IIoT systems encompass aspects, such as mechanical wear and tear or human negligence, that are complex to predict and model statically. More importantly, control software often needs to intervene in the execution of the cyber-physical processes (in case of safety violations).

There are two main RM intervention methods [25,35]. Runtime enforcement (RE) ensures the execution of the SuS meets correctness properties [6,7]. Enforcement monitors take *preventive* action, steering the operation of the system within the bounds of properties' specifications. RE assumes the SuS to be highly instrumentable, as monitors must enforce its execution continually. This prerequisite is often too stringent for IIoT systems comprising black-box or proprietary components. Runtime adaptation (RA) is a less stringent alternative to RE. RA takes remedial actions (also called adaptations) *after* monitors detect violations to a correctness property [24,49]. This RM variant is less invasive than RE since it does not need to instrument the SuS to the same degree. Adaptation monitors aim to *restore* the SuS to a *good state* from where the intended execution can potentially continue. This fits the black-box nature of industrial machines (such as the ones of Fig. 1) since monitors reason about and react to

the *observable behaviour* of components rather than their internal states. Moreover, since monitor and system components can run in *isolation* [5,15,40], RA can offer better safeguards against inadvertently affecting the SuS execution.

RA relies on two forms of prior knowledge on the SuS. The first is an abstract notion of states the SuS can be in, together with a set of observable *events* that capture the transitions between these states. The second kind of SuS knowledge assumed is the correctness behaviour defined over these states, defined in terms of either execution graphs or traces labelled by events. The latter identify a range of states that should *not* be reached during execution, and trigger adaptation monitors to administer *predetermined* remedial actions to steer the SuS back to a non-violating state. Note that the state reached following a series of remedial actions need not be a state visited prior to the violation. For instance, an AMR that ends up off-track on the shop floor can be directed back to its starting point by a RA, but an AMR that develops a mechanical fault may need to be powered off, which constitutes an *unvisited* state that permits graceful degradation without compromising safety.

3.2 Reversible Computation

Reversible computation (RC) is a software engineering paradigm where computations are partitioned as either executing *forwards* or *backwards* [10,75]. This technique allows program executions to be undone whenever a number of backward computation steps are the reverse of forward steps, providing an *efficient* mechanism for error recovery and fault-tolerance. RC was originally motivated by the need for low-energy computing and heat dissipation reduction [56]. It has since been applied to other areas, including programming languages [46,86] and debugging [34,45,55], modelling and algorithm design [19,31,32,54,65,66], as well as to robotics [61,62]. In IIoT settings such as the one outlined in Fig. 1, RC is useful because many physical operations have a natural (or *direct*) reverse counterpart that undoes the effect of the original operation, *e.g.* a MM moving to the left by one unit can be reversed by moving to the right by the same unit. This provides an opportunity to create a layer of abstraction that facilitates programming and enables code reuse. The relationship between forward operations and their reversible counterpart is not always direct, particularly in cases where operations have external dependencies or side effects. In such cases, reversibility may be achieved by traversing reversible computational paths that differ from the original computation. One way to achieve this *indirect* reversibility makes use of checkpoints [38,57,68,83] that allow systems to rollback to a known reversible state, or else, by explicitly creating separate functions that perform the reverse operations. In some cases, auxiliary mechanisms (*e.g.*, memory for execution histories) are employed to reconstruct past states and enable reversibility [21,60,67]. Operations that cannot be directly or indirectly reversed are considered *irreversible* [62].

The above concepts extend naturally to industrial automation settings, such as the factory shop floor discussed in Sect. 2, where computation corresponds to sequences of physical actions [62]. Every operation has its own reversibility

characteristics defined by its intrinsic nature and its effect on the shop floor workflow. Some operations are directly reversible due to their confined side effect on other entities on the shop floor, such as a MM retracing its path or recharging the MM batteries. Others require indirect reversal. For example, a MM pushing a batch into a Stocker must instead grasp the batch before pulling it back. Picking the last die from the Stocker in Fig. 1 is also indirectly reversible when the resulting empty Stocker affects the operation of the other MMs. This is the case since in addition to returning the die to the Stocker, reversing the operation needs to inform the other MMs that the Stocker is no longer empty. There are also operations like soldering or gluing, which are inherently irreversible because there are no obvious recovery operations that apply. A RC framework may also lift the terms directly reversible, indirectly reversible, and irreversible to *strategies*. Strategies are defined as sequences of operations. A strategy is a *forward* strategy when it consists exclusively of forward operations; backward strategies are defined analogously. Forward strategies are reversible whenever there exists a backward strategy that restores the computation to the original state. A forward strategy is *directly reversible* when its backward strategy is composed of the inverse of its forward operations in reverse order. Conversely, an *indirectly* reversible strategy is one where its backward strategy does not satisfy the above condition [61,62].

3.3 RA and RC Side-by-Side

There are a few key distinguishing aspects of RA, discussed in Sect. 3.1, and RC, discussed in Sect. 3.2. The first one lies in how error states are defined and handled. In RC, there is a fixed delineation between good and error states, where backward computation is triggered only when the program transitions to an error state [58]. By contrast, error states in RA are captured implicitly by the correctness property being considered: an erroneous state for one property may be a good state for a different property. In certain cases, RA and RC may converge when a particular state is inherently erroneous for any correctness property (*e.g.* a damaged robotic arm). A second, but related, differentiating aspect is that correct behaviour is more explicit in RA, formally stated in terms of a (declarative) specification logic [37]. These specifications then permit the automated synthesis of the (algorithmic) runtime adaptation procedure in terms of monitors. The expected correct behaviour is typically left implicit in the case of RC. Leaving this expected behaviour implicit rules out the possibility of applying automated program synthesis techniques, where the reversible program generally needs to be hand-coded by the software engineer. The third distinguishing feature is that in RC, reversible operations *guarantee* a return to a specific previous state in the (forward) computation or an equivalent state for some notion of state equivalence [10,32,60], *e.g.* causal-consistent reversibility where two states are equivalent modulo the order of concurrent actions [59]. RA offers no such guarantees. This permits the formulation of trial-and-error repetition strategies to deal with unpredictable (minor) variations or extraneous conditions on the shop floor that are not captured via high-precision sensors. For

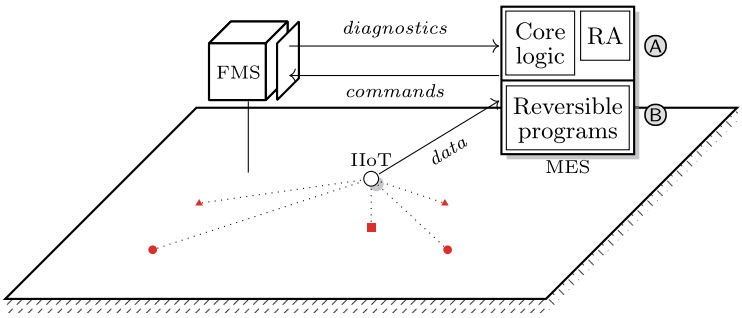

Fig. 2. MES monitoring and controlling shop floor via FMS and IIoT sensor network.

example, a MM that misses the docking station when moving forward (possibly due to tiny obstructing objects), can be reattempted after moving backwards. This reduces reliance on manual tuning and calibration to handle special cases, thereby improving the efficiency and adaptability of automated production lines. In practice, repetition strategies are limited. For example, a MM retracing the same route consumes battery power and contributes to mechanical wear, making unbounded repeated attempts costly and unsustainable. Such operations are considered *partially repeatable*, as they can only be performed a limited number of times. Schultz *et al.* [76] address this by introducing upper bounds on repetitions, for instance by associating retries with a finite number of tokens, enabling more predictable and resource-aware reversible strategies.

4 Recovering from Issues on the Factory Shop Floor

Human safety, error recovery, and graceful degradation are critical concerns in Industry 4.0 and 5.0 systems. We study how RA and RC from Sect. 3 can be used to address these challenges via a selection of scenarios describing potential issues that could arise in the factory shop floor model presented in Sect. 2.1.

4.1 Target Architecture

Our shop floor is managed by one MES that orchestrates the movement of MMs and coordinates the high-precision machines and Stocker operations through the FMS. Figure 2 depicts this centralised set-up. The MES receives diagnostic information from the FMS API, *e.g.* $\mathsf{blok}(\mathsf{dev} = \mathrm{MM}_1, \mathsf{at} = \mathrm{Stocker})$, and issues commands in response, *e.g.* $\mathsf{move}(\mathsf{dev} = \mathrm{MM}_1, \mathsf{from} = \mathrm{Stocker}, \mathsf{to} = \mathrm{Bonder}, \mathsf{waypts} = [1,2,3])$, which the FMS relays to machines on the shop floor. Our MES supplements FMS diagnostics by LiDAR, vibration, and particulate sensor data collected through the IIoT network deployed on the shop floor, *e.g.* $\mathsf{vibr}(\mathsf{dev} = \mathrm{MM}_1, \mathsf{amt} = 5)$.

Figure 2 illustrates two alternative methods of implementing the MES. Method Ⓐ implements the MES using *conventional* programs instrumented with

RA monitors, whereas method Ⓑ implements the MES as *reversible* programs. RA induces a *separation-of-concerns*, delineating the core program logic that handles normal-case operation and the RA monitor control flow, which intervenes when unexpected conditions arise [26]. By comparison, RC integrates the core logic and the error-handling mechanisms within a *single* program. Props. P_1 to P_4 in Sect. 4.2 showcase how remedial interventions can be implemented in terms of the RA and RC paradigms.

4.2 RA and RC for Handling Runtime Errors

Blocked Docking Station. Recall that in Fig. 1, each high-precision machine and Stocker own a designated docking station where MMs can deposit or pick up ICs. MMs are equipped with LiDAR modules to perceive and map the environment in real-time. Machine docking stations must be unobstructed for a MM to dock successfully. One property we want to hold is:

'MMs never block when entering a docking station.' (P_1)

Suppose the shop floor executes step ① from Sect. 2.1. The MES starts by issuing a pick command to the FMS, instructing MM_2 to lift the tray of bonded dies from the Bonder, *i.e.,* $\mathsf{pick}(\mathsf{dev} = MM_2, \mathsf{from} = \text{Bonder}, \mathsf{obj} = \text{Bonded\_Dies})$. Subsequently, the FMS relays a second MES command to MM_2 to move the machine from the Bonder to the Stocker via specific waypoints, $\mathsf{move}(\mathsf{dev} = MM_2, \mathsf{from} = \text{Bonder}, \mathsf{to} = \text{Stocker}, \mathsf{waypts} = [1,2,3])$. If the LiDAR on MM_2 detects obstructions at the Stocker docking station, it transmits the data $\mathsf{lidr}(\mathsf{dev} = MM_2, \mathsf{dst} = 0.1)$ to the MES. Since MM_2 is unable to dock, the FMS also sends the $\mathsf{blok}(\mathsf{dev} = MM_2, \mathsf{at} = \text{Stocker})$ signal to the MES. Both lidr and blok enable the MES to detect a violation of prop. P_1.

Runtime Adaptation. To remedy the violation of prop. P_1, the MES RA monitor would instruct MM_2 to return to the Bonder via the command:

$$\mathsf{move}(\mathsf{dev} = MM_2, \mathsf{from} = \text{Stocker}, \mathsf{to} = \text{Bonder}, \mathsf{waypts} = [3,2,1]). \quad (1)$$

To promote *automation flexibility* (see Sect. 1) and minimise blockage on the factory shop floor, the RA monitor could also redirect MM_2 to deposit the bonded ICs at a second Stocker, if available. This manoeuvre is expressed as the command $\mathsf{move}(\mathsf{dev} = MM_2, \mathsf{from} = \text{Stocker}, \mathsf{to} = \text{Stocker}', \mathsf{waypts} = [3,4,5])$.

Reversible Computation. Forward computation comprises the operation sequence:

$$\mathsf{pick}(\mathsf{dev} = MM_2, \mathsf{from} = \text{Bonder}, \mathsf{obj} = \text{Bonded\_Dies}); \quad (2)$$
$$\mathsf{move}(\mathsf{dev} = MM_2, \mathsf{from} = \text{Bonder}, \mathsf{to} = \text{Stocker}, \mathsf{waypts} = [1,2,3]) \quad (3)$$

When the MES receives the signal $\mathsf{blok}(\mathsf{dev} = MM_2, \mathsf{at} = \text{Stocker})$ from the FMS, the system transitions to an error state. This triggers the RC program

on the MES to perform the backward computation for op. (3) and reverse the system to a good state. Op. (1) is the *direct inverse* of the forward computation 3, which takes MM_2 back to the Bonder along the reverse waypoints $3, 2, 1$.

Note that RC *reverses* the system to a previous good state, while RA reverts to a good state that need not be the previous one. In this instance, RA offers more flexibility since it permits the MM to deposit ICs at a different Stocker.

Damaged Dies. Recall steps ①, ③ and ⑤ from Sect. 2.1 where MMs are tasked with fetching batches of dies from and depositing them to Stockers. Dies are highly sensitive and slight mishandling, *e.g.*, sudden jolts during transport, can render them defective. Although in such cases testing can be performed to assess whether dies are damaged, it is often cheaper and easier to discard suspect dies in practice. To detect potential damage, vibration sensors are mounted on MM chassis and arm grippers. The following property safeguards the integrity of dies:

'High-precision machines never receive trays with damaged dies.' (P_2)

Suppose a gripper vibration sensor reports values above a predefined threshold (say, 3 units) to the MES:

$$\mathsf{vibr}(\mathsf{dev} = \mathrm{MM}_1, \mathsf{amt} = 5) \tag{4}$$

At this point, we assume that the batch of dies in transit is likely damaged, in which case an error state is reached.

Runtime Adaptation. A RA-driven MES can discard the potentially damaged batch of dies at a designated disposal unit, *e.g.* by issuing the operation sequence

$$\mathsf{move}(\mathsf{dev} = \mathrm{MM}_1, \mathsf{from} = \mathrm{Current}, \mathsf{to} = \mathrm{Disposal\_Unit}, \mathsf{waypts} = [5{,}6]); \tag{5}$$

$$\mathsf{plce}(\mathsf{dev} = \mathrm{MM}_1, \mathsf{to} = \mathrm{Disposal\_Unit}, \mathsf{obj} = \mathrm{Dies}) \tag{6}$$

and instruct the MM to fetch a fresh batch from the Stocker.

Reversible Computation. The sequence of ops. (4) to (6) can be viewed as a RC. The forward computation consists of MM_1 transporting a batch of dies to the high-precision machine. Upon detecting a problem, signalled by op. (4), the MES instructs MM_1 to discard the potentially faulty batch, ops. (5) and (6). Discarding the batch reverses the effect of the error. This strategy is *indirectly reversible*, since a good state (*i.e.,* a high-precision machine receiving functional dies) is reached via a series of operations rather than via one inverse operation. We remark that since the batch of remaining functional dies in the Stocker is necessarily limited, this reverse strategy is also *partially repeatable*.

Damaged Robotic Arm. Robotic arm damage, *e.g.* human or other equipment colliding with the arm, can be indirectly detected through the MM vibration sensors mentioned earlier. The next property captures this requirement:

'Robot arm vibration levels never exceed 50.' (P_3)

Suppose MM_2 in Fig. 1 experiences a collision in step ④ whilst executing the FMS command $\mathsf{move}(\mathsf{dev} = MM_2, \mathsf{from} = \text{Stocker}, \mathsf{to} = \text{Deflasher}, \mathsf{waypts} = [1,4,7])$. The vibration sensor signals the MES, $\mathsf{vibr}(\mathsf{dev} = MM_2, \mathsf{amt} = 100)$, which triggers a violation of the prop. 3.

Runtime Adaptation. A RA monitor that flags this violation can decommission the affected MM by instructing it to return to the repair bay, *e.g.*, $\mathsf{move}(\mathsf{dev} = MM_2, \mathsf{from} = \text{Current}, \mathsf{to} = \text{Repair\_Bay}, \mathsf{waypts} = [2,3])$. The MES can then reassign the pending tasks of MM_2 to other MMs in the fleet, and optionally, notify a technician on the shop floor to assess the damage. The following operation sequence is a hypothetical adaptation the MES initiates via the FMS to reassign the affected task to MM_1.

$$\mathsf{pick}(\mathsf{dev} = MM_1, \mathsf{from} = \text{Stocker}, \mathsf{obj} = \text{Encased\_Dies});$$
$$\mathsf{move}(\mathsf{dev} = MM_1, \mathsf{from} = \text{Stocker}, \mathsf{to} = \text{Deflasher}, \mathsf{waypts} = [6,3,1])$$

Reversible Computation. Damage to the robotic arm is irreversible since the MES cannot restore MM_2 to an operational state using a backward computation.

Degraded Air Quality. Industrial environments can generate particulate matter from physical processes, *e.g.*, wafer dicing and IC deflashing (see Sect. 2.1), foot traffic, *etc.* This can damage the assembly of ICs, and in higher amounts, degrades sensor accuracy and endangers human health [44]. For instance, an ISO Class 5 cleanroom according to ISO 14644-1 allows a maximum of 3,520 particles/m^3 that are 0.5 µm in size or larger [1]. Our shop floor of Fig. 1 is equipped with particulate sensors that monitor air quality and report readings to the MES via the IIoT network. The shop floor is also outfitted with high efficiency particulate air (HEPA) filters to purify the air when required. A property that ensures cleanroom air quality levels is:

'Particulate matter levels never exceed 3,520.' (P_4)

Particulate sensors sample the air quality at the end of every workflow round, *i.e.,* after all MMs have completed their tasks and return to their respective docking stations. Suppose that at the termination of one such round, *e.g.* step ⑤ of ex. 1, the particulate sensor detects degraded air quality and transmits the reading $\mathsf{part}(\mathsf{amt} = 5001)$ to the MES, which violates prop. P_4.

Runtime Adaptation. The RA monitor for prop. P_4 can trigger a recovery procedure that activates the HEPA filtration system to purify the air

$$\mathsf{hepa}(\mathsf{spdlvl} = 5, \mathsf{duration} = 10); \tag{7}$$

and instructs MMs to execute wheel-cleaning routines at designated adhesive cleaning zones

$$\mathsf{move}(\mathsf{dev} = \mathrm{MM}_1, \mathsf{from} = \mathrm{Stocker}, \mathsf{to} = \mathrm{Adhesive\_Zone}, \mathsf{waypts} = [2,4,6]); \quad (8)$$

$$\mathsf{move}(\mathsf{dev} = \mathrm{MM}_2, \mathsf{from} = \mathrm{Deflasher}, \mathsf{to} = \mathrm{Adhesive\_Zone}, \mathsf{waypts} = [1,6]) \quad (9)$$

Reversible Computation. The handling of a violation of prop. P_4 is not a directly reversible RC operation. In fact, reversing the MMs to their original position on the shop floor (*e.g.*, reroute MM_1 back to the Stocker in step ①) does not restore the air quality, since this is a by-product of the physical production process. Activating the HEPA filtration system, however, *indirectly reverses* the system to a state of optimal air quality. More generally, restoring the air quality may require repeated activations of the filtration system, *e.g.* performing op. (7) twice, until a safe air quality level is reached. Lastly, if our notion of reversing the system to a previous good state includes the MM wheel-cleaning procedure, the indirect reversal strategy can be extended to include the ops. (8) and (9). Notably, the reversal ops. (8) and (9), together with op. (7) can be executed concurrently. This appears to contrast with previous assumptions about reversibility strategies necessarily being structured as sequences of operations, and suggests the application of more elaborate reversibility theories [32,59,65] to IIoT, *e.g.* approaches such as [22] that allow for concurrent compensation actions.

4.3 RA and RC Complementarity

RA and RC can be viewed as *complementary* software design principles. It is possible to implement RA properties using *only* the reversible operations in a reversible programming language: props. P_1 and P_2 in Sect. 4.2 are instances of this approach. Conversely, RC can be engineered in terms of RA, where adaptation actions encapsulate the *ad hoc* implementation logic that corresponds to high-level reversible operations, *e.g.* see discussion on prop. P_4. Other examples of the latter view include recent work [26,38,39], which proposes reversibility for reliable and fault-tolerant message-passing concurrency via choreographed RA monitors. We discuss other aspects of RA and RC next.

Reversible Programs via RA. Categorising operations and strategies as (i) *directly reversible*, *i.e.,* invertible and can be reversed automatically, (ii) *indirectly reversible*, *i.e.,* requires a manually-defined reverse possibly consisting of multiple operations, or (iii) *irreversible*, enables compilers to provide *static* guarantees about reversible programs [62]. Writing reversible programs requires explicit reasoning about *forward* and *backward* computations layered over implicit notions of good and error states. Reasoning about such programs may be challenging, as the logic for correct behaviour and error mitigation is embedded directly in the code. RA can alleviate this burden by *decoupling* the monolithic program logic that intertwines forward and backward system computation. Much like aspect-oriented programming in mainstream languages [50], this approach

treats reversibility as a *cross-cutting* concern. It structures reversible programs into two parts: forward-computation (executable) code and declarative high-level RA properties whose adaptation actions capture the backward-computation logic of the program *exclusively* through reversible operations. Once RA synthesises properties into executable monitor code, it *weaves* it with the forward-computation program code to generate the complete reversible program. This two-stage approach to generating reversible programs through RA has three benefits. First, it simplifies program reasoning by separating the forward- and backward-computation logic, and automating the generation of the completed reversible program. Second, backward-computation code can be further decomposed into fine-grained RA actions that are layered around the core forward-computation logic, improving modularity and enabling incremental development. The last crucial benefit is that by limiting RA actions to reversible operations inherits the static guarantees enjoyed by the RC programming language.

Combining RA with Reversibility. Irreversible RC strategies cannot revert a system to a previous good state. RA is not bound by these all-or-nothing reversibility constraints. This makes RA applicable to cases where *partial recovery* is sufficient to achieve graceful degradation and uphold automation flexibility outlined, *e.g.* prop. 3 in Sect. 4.2. RA can still benefit from reversible operations and the static guarantees they bring about. This can be obtained at two different levels. The first method organises RA properties where *each* property expresses its remedial procedure using either *ad hoc* logic *or* reversible operations. The second method mixes both *ad hoc* logic and reversible operations within the same RA property. For instance, the remedial RA actions that uphold prop. 3 in Sect. 4.2 instruct MM_2 to return to its base. The 'returning to base' operation may need to be reversed if the base of MM_2 is blocked, and this part of the property can easily be expressed using reversible operations (see prop. P_1). However, reassigning pending tasks via the fleet manager in prop. 3 is not a backward computation, and this latter segment of the remedial action can be easily expressed using *ad hoc* logic.

5 Conclusion

This paper compares the strengths and limitations of runtime adaptation (RA) and reversible computation (RC) as software paradigms for detecting and recovering from errors in industrial IoT (IIoT) environments. We use a representative IC manufacturing shop floor use case to showcase how both approaches can be used to address common issues, *e.g.* equipment damage and environmental hazards, emphasising different design trade-offs.

We observe that RA can support *ad hoc* recovery and graceful degradation in the presence of irreversible operations. It delineates the reasoning about forward computations, analysed as runtime events, and remedial actions, executed as adaptation actions. RA is ideal for integrating cross-cutting behaviour as monitors, modularising and supporting incremental IIoT systems development.

By contrast, RC guarantees fine-grained reversibility and supports repetition strategies for trial-and-error recovery. The static guarantees given by RC are counterbalanced by the (i) upfront effort required to encode high-level correctness criteria directly into program logic, (ii) cognitive overhead of reasoning about interleaved forward and backward computations, and (iii) limited applicability of the approach in scenarios involving irreversible actions. We also explore the complementary nature of RA and RC. RA can benefit from incorporating reversible operations to gain predictability and verification guarantees, while RC development processes may be enriched by declarative RA specifications to facilitate development and enhance modularity. Future work will explore automated synthesis of hybrid RA-RC monitors and their use in real-world smart manufacturing settings.

References

1. ISO/TC 209. ISO 14644-1: Cleanrooms and Associated Controlled Environments. Technical report, Iinternational Organization for Standardization (2025)
2. Aceto, L., Achilleos, A., Attard, D.P., Exibard, L., Francalanza, A., Ingólfsdóttir, A.: A monitoring tool for linear-time μhml. SCP, **232**, 103031 (2024)
3. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in monitorability: from branching to linear time and back again. PACMPL, **3**, 52:1–52:29 (2019)
4. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: An operational guide to monitorability with applications to regular properties. SSM **20**, 335–361 (2021)
5. Aceto, L., Attard, D.P., Francalanza, A., Ingólfsdóttir, A.: Runtime instrumentation for reactive components. In: ECOOP, LIPIcs, vol. 313, pp. 2:1–2:33. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024)
6. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: On runtime enforcement via suppressions. In: CONCUR. LIPIcs, vol. 118, pp. 34:1–34:17 (2018)
7. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: On first-order runtime enforcement of branching-time properties. Acta Informatica **60**(4), 385–451 (2023)
8. Adlin, N., Nylund, H., Lanz, M., Lehtonen, T., Juuti, T.: Lean Indicators for small batch size manufacturers in high cost countries. Procedia Manuf. **51**, 1371–1378 (2020). 30th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM2021)
9. Ahmed, S.F., et al.: Industrial internet of things enabled technologies, challenges, and future directions. CEE **110**, 108847 (2023)
10. Aman, B., et al.: Foundations of reversible computation. In: Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405, pp. 1–40 (2020)
11. Anandan, R., Gopalakrishnan, S., Pal, S., Zaman, N.: Intelligent Analytics for Predictive Maintenance. Wiley, Industrial Internet of Things (IIoT) (2022)
12. Attard, D.P., Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Better late than never or: verifying asynchronous components at runtime. In: Peters, K., Willemse, T.A.C. (eds.) FORTE 2021. LNCS, vol. 12719, pp. 207–225. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78089-0_14

13. Attard, D.P., Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Introduction to runtime verification. In: Behavioural Types: from Theory to Tools, Automation, Control and Robotics, pp. 49–76. River (2017)
14. Attard, D.P., Francalanza, A.: A monitoring tool for a branching-time logic. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 473–481. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_31
15. Attard, D.P., Francalanza, A.: Trace partitioning and local monitoring for asynchronous components. In: Cimatti, A., Sirjani, M. (eds.) SEFM 2017. LNCS, vol. 10469, pp. 219–235. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_14
16. Balakrishnan, A., et al.: Safety assurance for autonomous systems with multiple sensor modalities. In: MEMOCODE, pp. 106–111 (2024)
17. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
18. Baydar, C.M., Saitou, K.: Off-line error prediction, diagnosis and recovery using virtual assembly systems. JIM **15**(5), 679–692 (2004)
19. Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev. **17**(6), 525–532 (1973)
20. Berndt, M., Krummacker, D., Fischer, C., Schotten, H.D.: Centralized robotic fleet coordination and control. In: Mobile Communication - Technologies and Applications; 25th ITG-Symposium, pp. 1–8 (2021)
21. Bocchi, L., Lanese, I., Mezzina, C.A., Yuen, S.: revTPL: the reversible temporal process language. LMCS, **20**(1) (2024)
22. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: POPL, pp. 209–220. ACM (2005)
23. Buch, J.P., et al.: Applying simulation and a domain-specific language for an adaptive action library. In: Brugali, D., Broenink, J.F., Kroeger, T., MacDonald, B.A. (eds.) SIMPAR 2014. LNCS (LNAI), vol. 8810, pp. 86–97. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11900-7_8
24. Cassar, I., Francalanza, A.: Runtime adaptation for actor systems. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 38–54. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_3
25. Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A.: A survey of runtime monitoring instrumentation techniques. In: PrePostiFM, EPTCS, vol. 254, pp. 15–28 (2017)
26. Cassar, I., Francalanza, A., Mezzina, C.A., Tuosto, E.: Reliability and fault-tolerance by choreographic design. In: PrePostiFM, EPTCS, vol. 254, pp. 69–80 (2017)
27. Chen, H., Zhang, G., Zhang, H., Fuhlbrigge, T.A.: Integrated robotic system for high precision assembly in a semi-structured environment. Assembly Autom. **27**(3) (2007)
28. Christensen, H., et al.: A Roadmap for US Robotics - From Internet to Robotics, vol. 2020. Now Publishers Inc, Edition (2021)
29. European Commission, Directorate-General for Research, Innovation, M. Breque, L. De Nul, and A. Petridis. Industry 5.0 – Towards a sustainable, human-centric and resilient European industry. Publications Office of the European Union (2021)
30. Conesa-Muñoz, J., Gonzalez-de-Soto, M., González de Santos, P., Ribeiro, R.: Distributed multi-level supervision to effectively monitor the operations of a fleet of autonomous vehicles in agricultural tasks. Sensors, **15**(3), 5402–5428 (2015)

31. Danos, V., Krivine, J.: Formal molecular biology done in CCS-R. In: BioConcur@CONCUR, Electronic Notes in Theoretical Computer Science, vol. 180, pp 31–49. Elsevier (2003)
32. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19
33. Donald, B.R.: Error Detection and Recovery in Robotics. LNCS, vol. 336. Springer, New York (1989). https://doi.org/10.1007/BFb0039640
34. Fabbretti, G., Lanese, I., Stefani, J.-B.: Causal-consistent debugging of distributed erlang programs. In: Yamashita, S., Yokoyama, T. (eds.) RC 2021. LNCS, vol. 12805, pp. 79–95. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79837-6_5
35. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. STTT **23**, 255–284 (2021)
36. Forte, P., Mannucci, A., Andreasson, H., Pecora, F.: Online task assignment and coordination in multi-robot fleets. IEEE Robot. Autom. Lett. **6**(3), 4584–4591 (2021)
37. Francalanza, A., et al.: A foundation for runtime monitoring. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 8–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_2
38. Francalanza, A., Mezzina, C.A., Tuosto, E.: Reversible choreographies via monitoring in erlang. In: Bonomi, S., Rivière, E. (eds.) DAIS 2018. LNCS, vol. 10853, pp. 75–92. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93767-0_6
39. Francalanza, A., Mezzina, C.A., Tuosto, E.: Towards choreographic-based monitoring. In: Ulidowski, I., Lanese, I., Schultz, U.P., Ferreira, C. (eds.) RC 2020. LNCS, vol. 12070, pp. 128–150. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-47361-7_6
40. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 176–210. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
41. Garrett, C.R., Lozano-Pérez, T., Kaelbling, L.P.; Backward-forward search for manipulation planning. In: IROS, pp. 6366–6373. IEEE (2015)
42. Graham, S.L., Kessler, P.B., McKusick, M.K.: Gprof: a call graph execution profiler. In: SIGPLAN Symposium on Compiler Construction, pp. 120–126. ACM (1982)
43. Hažík, J., Dekan, M., Beňo, P., Duchoň, F.: Fleet management system for an industry environment. JRC **3**(6), 779–789 (2022)
44. Hayes, D.: Making chips with dust-free poison. Sci. Cult. **1**(1), 89–104 (1987)
45. Hoey, J., Lanese, I., Nishida, N., Ulidowski, I., Vidal, G.: A case study for reversible computing: reversible debugging of concurrent programs. In: Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405, pp. 108–127 (2020)
46. Hoey, J., Ulidowski, I.: Reversing an imperative concurrent programming language. SCP **223**, 102873 (2022)
47. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
48. Kagermann, H., Wolfgang, W., Helbig, J.: Recommendations for implementing the strategic initiative INDUSTRIE 4.0. Technical report, Work. Group. Acatech, Frankfurt am Main, Ger., (2013)

49. Kalareh, M.A.; Evolving Software Systems for Self-Adaptation. PhD thesis, University of Waterloo, Ontario, Canada (2012)
50. Kiczales, G., et al.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
51. Kneževic, N., Savić, A., Gordić, Z., Ajoudani, A., Jovanović, K.: Toward Industry 5.0: a neuroergonomic workstation for a human-centered, collaborative robot-supported manual assembly process. IEEE Robot. Autom. Mag. 2–13 (2024)
52. Koval, M.C., King, J.E., Pollard, N.S., Srinivasa, S.S.: Robust trajectory selection for rearrangement planning as a multi-armed bandit problem. In: IROS, pp. 2678–2685. IEEE (2015)
53. Kuhn, R., Hanafee, B., Allen, J.: Reactive Design Patterns. Manning (2016)
54. Kutrib, M.: Reversible and irreversible computations of deterministic finite-state devices. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9234, pp. 38–52. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48057-1_3
55. Lami, P., Lanese, I., Stefani, J.-B., Coen, C.S., Fabbretti, G.: Reversible debugging of concurrent erlang programs: supporting imperative primitives. JLAMP **138**, 100944 (2024)
56. Landauer, R.: Irreversibility and heat generated in the computing process. IBM J. Res. Dev. **5**, 183–191 (1961)
57. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Controlling reversibility in higher-order Pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_20
58. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Controlled reversibility and compensations. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 233–240. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_19
59. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Reversibility in the higher-order π-calculus. TCS **625**, 25–84 (2016)
60. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for erlang. JLAMP **100**, 71–97 (2018)
61. Lanese, I., Schultz, U.P., Ulidowski, I.: Reversible execution for robustness in embodied AI and industrial robots. IT Prof. **23**(3), 12–17 (2021)
62. Laursen, J.S., Ellekilde, L.-P., Schultz, U.P.: Modelling reversible execution of robotic assembly. Robotica **36**(5), 625–654 (2018)
63. Liao, Y., Freitas, E., Loures, R., Deschamps, F.: Industrial internet of things: a systematic literature review and insights. IEEE Internet Things J. **5**(6), 4515–4525 (2018)
64. Loborg, P.: Error Recovery in Manufacturing Control Systems. PhD thesis, School of Engineering, Linköping University, Sweden (1994)
65. Melgratti, H., Mezzina, C.A., Michele Pinna, G.: A truly concurrent semantics for reversible CCS. LMCS, **20**(4) (2024)
66. Melgratti, H., Mezzina, C.A., Ulidowski, I.: Reversing Place transition nets. LMCS, **16** (2020)
67. Mezzina, C.A., Pérez, J.A.: Causally consistent reversible choreographies: a monitors-as-memories approach. In: PPDP, pp. 127–138. ACM (2017)
68. Mezzina, C.A., Tiezzi, F., Yoshida, N.: Checkpoint-based rollback recovery in session programming. LMCS, **21**(1), 2 (2025)
69. ST Microcontroller Division Applications. Application Note: Introduction to Semiconductor Technology (2000)

70. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: IPDPS, pp. 494–503 (2015)
71. Neto, P., Mendes, N., Araujo, R., Norberto Pires, J., Paulo Moreira, A.: High-level robot programming based on CAD: dealing with unpredictable environments. Ind. Robot, **39**(3) (2012)
72. NXP Semiconductors. Official Website (2024). https://www.nxp.com
73. Okumura, S., Take, N., Okino, N.: Error prevention in robotic assembly tasks by a machine vision and statistical pattern recognition method. Int. J. Prod. Res. **43**(7), 1397–1410 (2005)
74. Pasha, A.: Lights-Out Manufacturing: Revolutionizing the Factory Floor with Automation. Technical report, Bosch (2025)
75. Kalyan, S.: Perumalla. Introduction to Reversible Computing, CRC Press (2014)
76. Schultz, U.P., Bordignon, M., Støy, K.: Robust and reversible execution of self-reconfiguration sequences. Robotica, **29**(1), 35–57 (2011)
77. Sehrawat, D., Gill, N.S.: Smart sensors: analysis of different types of IoT sensors. In: 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), pp. 523–528 (2019)
78. Sparc: The partnership for Robotics in Europe. Robotics 2020:Multi-Annual Roadmap for Robotics in Europe. Technical report, EU Robotics AISBL, Brussels (2017)
79. STMicroelectronics. Official Website (2024). https://www.st.com
80. Tan, M., Wang, P., Luo, W.: Optimized real-time monitoring and fault diagnosis system for industrial robots with integrated sensor data. In: ICIPCN, pp. 764–768 (2024)
81. Tarkoma, S.: Overlay Networks: Toward Information Networking. Auerbach (2010)
82. Texas Instruments. Official Website (2024). https://www.ti.com
83. Vassor, M., Stefani, J.-B.: Checkpoint/rollback vs causally-consistent reversibility. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 286–303. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_20
84. Villani, V., Gabbi, M., Sabattini, L.: Promoting operator's wellbeing in Industry 5.0: detecting mental and physical fatigue. In: 2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 2030–2036 (2022)
85. Wang, Y., et al.: Probabilistic graph based spatial assembly relation inference for programming of assembly task by demonstration. In: IROS, pp. 4402–4407. IEEE (2015)
86. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Conference Computing Frontiers, pp. 43–54. ACM (2008)
87. Zhang, B., Wang, J., Rossano, G., Martinez, C.: Vision-guided robotic assembly using uncalibrated vision. In: 2011 IEEE International Conference on Mechatronics and Automation, pp. 1384–1389 (2011)