Rollback Sequences for Dynamic Reconfiguration of IEC 61499

Laurin Prenzel
Technical University of Munich
Munich, Germany
laurin.prenzel@tum.de

Simon Hofmann
Technical University of Munich
Munich, Germany
simon1.hofmann@tum.de

Sebastian Steinhorst
Technical University of Munich
Munich, Germany
sebastian.steinhorst@tum.de

***Abstract*—Dynamic reconfiguration is a core contributor to the flexibility and agility of future industrial control systems. Verification and validation can provide some confidence in the success of a reconfiguration, yet unexpected external events or bugs can always lead to the abortion of the reconfiguration process. This can threaten the real-time behavior and must be anticipated. In this paper, we extend existing real-time models of dynamic reconfiguration to incorporate safe rollback scenarios that allow a disruption-free reversal of the reconfiguration process, thus providing fault-tolerance. We introduce the concept of a point of no return, after which a rollback is no longer feasible. We demonstrate in two example systems how the ordering of operations can affect the length of the rollback sequence and optimize the ordering of operations in two stages to find a sequence that offers a maximal fault-tolerance, while minimizing the real-time disruption. The results indicate that while considering potential failure modes requires additional overhead, it can provide fault-tolerance that promotes the further application of dynamic reconfiguration in practical applications. This may lead to higher agility and resilience in industrial control systems of the future.**

***Index Terms*—Downtimeless System Evolution, Dynamic Software Updating, Dynamic Adaptation, Industrial Automation**

I. Introduction

Dynamic reconfiguration of safety-critical or real-time systems is a difficult process. Although validation and verification can and must be applied, there can never be a guarantee that the reconfiguration will work as expected, especially if there are external factors and the reconfiguration takes place over a long(-er) period. Thus, it is important to consider a failure of the reconfiguration and have a backup plan at hand to recover to a safe state. In the most rudimentary example, this may simply lead to an emergency stop of the system. For systems that can't be stopped, the recovery may require rolling back the changes that were applied and returning the system to its original state. For real-time systems, the rollback must not violate the real-time constraints.

The IEC 61499, as a domain-specific modeling language for distributed industrial control system (ICS), supports dynamic reconfiguration. There have been many works on how to achieve a kind of *downtimeless system evolution* [1, 2], and how to achieve it safely [3, 4]. Yet, it is usually not considered what to do if things inevitably go wrong. Figure 1 demonstrates this behavior. If a reconfiguration sequence can bring the system

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany (BMBF) in the framework of ReMiX (project number 01IS18063B).

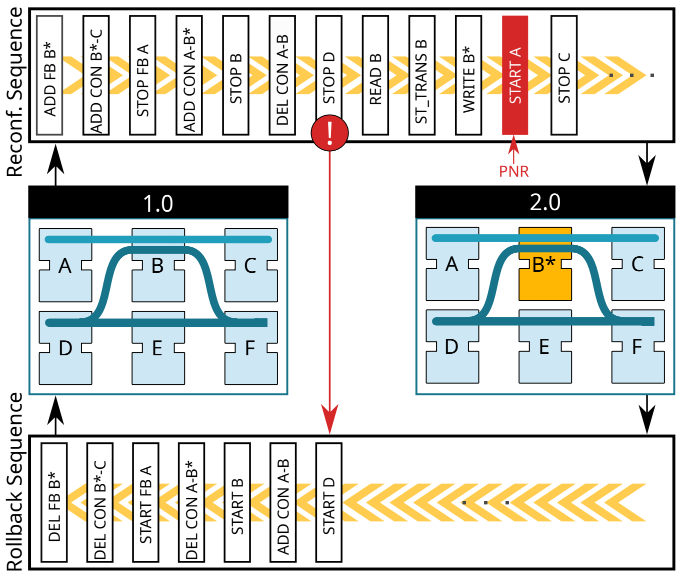


Fig. 1. A rollback sequence represents a sequence of operations that reverses the changes of a reconfiguration sequence. If an error occurs during the reconfiguration, the rollback can be applied to return the system into a safe operational state, as long as it happens before the point of no return (PNR).

from a state 1 to a state 2, then a rollback sequence should bring the system from state 2 back to 1. If somewhere in the middle a failure occurs, it should be possible to jump from the reconfiguration sequence to the rollback sequence. The difficulty appears, when there are real-time constraints that must be satisfied, and when some operations can not be rolled back. In some cases, there is a loss of information during the reconfiguration, e.g. when a component and the contained state are deleted. In other cases, the change of behavior is irreversible.

In this paper, we address the problem of finding a suitable rollback sequence for a given reconfiguration sequence. We identify the point of no return (PNR) from two angles: The real-time and the reversibility perspective. Using an optimization algorithm, we find a reconfiguration and rollback sequence with the highest potential for rollback, while preserving the schedulability. In this manner, we can guarantee the timing constraints even during a failure of the reconfiguration. This behavior is demonstrated in two case studies.

The remainder of the paper is structured as follows: Section II introduces the IEC 61499, dynamic reconfiguration, and the existing works on preserving consistency and real-time constraints. Section III represents our extension to the existing theory regarding the invertible operations, the PNR, and how to optimize the rollback sequence. This work is evaluated in Section IV, and we conclude in Section V.

II. Background

A. IEC 61499

The IEC 61499 was introduced as an extension to the models of the IEC 61131 for distributed control systems. The control logic is encapsulated in function blocks (FBs) that can be easily disseminated without side effects, since there is no global state. Recently, the IEC 61499 has seen renewed interest in practice [5] and it has been implemented in various runtime environments with differing execution semantics [6]. The real-time execution of the IEC 61499 was addressed in [7], yet blocking access due to shared resources or reconfiguration were not considered then.

B. Reconfiguration Control Applications

Dynamic Reconfiguration or downtimeless system evolution has been previously proposed for the IEC 61499 [1, 2]. This is facilitated by the event-triggered execution and the fractured system state that can be exchanged efficiently, without halting the entire application. The reconfiguration services can be wrapped inside a reconfiguration control application (RCA) which modifies the actual control application in real-time. This process of reconfiguration can be split into five phases: The upload of the RCA, the initial, uncritical RINIT phase, the critical RECONF phase, the uncritical RDINIT phase, and the final cleanup of the RCA [3].

C. Consistent Dynamic Reconfiguration of IEC 61499

Initially, the RCA had to be manually developed and implemented. This can be rather cumbersome, especially since the size of the RCA can grow very large even for simple changes [7]. Further, this leaves the burden of verification on the developer. There are two concerns that must be met: The reconfiguration must preserve the consistency of the physical process and control logic, and the reconfiguration must be executed in real-time.

When performing a reconfiguration, the continuity of the control logic must be guaranteed. In particular, this means that the behavior of the system before, during, and after the introduction of the reconfiguration must be predictable and as intended. There are different consistency conditions that can be used to achieve this, and [4] uses a version of quiescence [8]. In this approach, FBs are stopped and started in the order of the event flow, i.e. from event source to event sink. Thus, the update is flushed through the system as new events are introduced.

The approach in [4] uses rules to automatically construct a dependency graph of the reconfiguration operations and the relevant precedence constraints that must be kept to preserve the consistency. This dependency graph represents the search space, from which a sequence of operations can be selected and an RCA can be constructed.

D. Timely Dynamic Reconfiguration of IEC 61499

From the search space of consistent reconfiguration sequences, a sequence must be picked that can satisfy the real-time requirements of the system. The real-time behavior and schedulability was investigated in [9]. We shortly summarize the findings as a basis for our extension to this work. Since the previous work focused on rate monotonic (RM) scheduling, we keep this assumption.

1) System Definition: First, we define our system model for the real-time system under reconfiguration. The taskset

$$\tau = (\tau_1, \cdots, \tau_n) \tag{1}$$

consists of n concurrent real-time tasks, ordered by their deadlines / rates $D = T$. These tasks represent the execution traces or event chains ([7]) in the system. The ordered sequence

$$S = \langle o_0, \cdots, o_n \rangle \tag{2}$$

is the reconfiguration sequence and consists of reconfiguration operations $o_i = (a_i, F_i, c_i^o)$, where a_i is an action, F_i is a set of affected FBs, and c_i^o is the worst case execution time (WCET) of the operation. This linear sequence is one possible path through the dependency graph.

2) Blocking Time and Schedulability: We reuse the schedulability condition as defined by [10], and solve it for the laxity

$$L(\tau_i) = \underbrace{T_i i(2^{1/i} - 1)}_{\text{Max. Utilization}} - \underbrace{T_i \sum_{j=0}^{i-1} \frac{C_j}{T_j}}_{\text{Preemption}} - \underbrace{C_i}_{\text{Execution}} - \underbrace{B(\tau_i)}_{\text{Blocking}} \tag{3}$$

as introduced by [9]. The blocking time $B(\tau_i)$ consists of the shared access to FBs, and the blocking introduced by a reconfiguration sequence S:

$$B_i = B(\tau_i) = B^{\text{FB}}(\tau_i) + B^R(\tau_i). \tag{4}$$

The calculation of the blocking time B^R is described in [9]. Intuitively, this blocking time starts when a FB that can block the execution of a task is stopped, and ends when all FBs that can block the task are resumed. To minimize the blocking time, the order of operations within the sequence can be modified to delay some stop operations, or start FBs as early as possible. Ultimately, a reconfiguration sequence must be found with a blocking time that leads to positive laxities for all tasks as defined in Equation 3.

E. Rollback Recovery from failed Reconfiguration

The need for recovery methods for (distributed) reconfigurations is well known in literature and practice [11–13]. The distributed case is particularly error-prone, since distributed nodes may transition independently into unsafe states where a reconfiguration must be aborted. A solution to unanticipated faults and errors in reconfigurations is checkpointing and rollback in case of a detected error [11]. An important problem of checkpointing and rollback is the possibility of message loss, which can not always be tolerated [13].

TABLE I
MOST RECONFIGURATION SERVICES OF THE IEC 61499 HAVE A CORRESPONDING INVERSE OPERATION. FOR SOME SERVICES, AN INVERSION IS EITHER NOT NECESSARY (STATELESS), OR NOT POSSIBLE.

IEC 61499-1	Inverse Operation
`CREATE FB`	`DELETE FB`
`CREATE CON`	`DELETE CON`
`DELETE FB`	Not invertible
`DELETE CON`	`ADD CON`
`START`	Not invertible
`STOP`	`START`
`WRITE`	`WRITE`
`READ`	No inversion necessary
`ST_TRANS`	No inversion necessary

In the general purpose programming language Erlang, a rollback of a dynamic reconfiguration (or hot code loading) is easily implemented and often possible automatically (after the definition of the required state transformations). The work in this paper is partially inspired by this feature, since it provides Erlang with great fault-tolerance. An inspiring formalization of reversibility of Erlang is provided by [14], which promises even greater fault-tolerance if there are mechanisms to automatically control the rollback.

III. METHODS

The generation of a rollback sequence first requires an analysis of the reversible or invertible operations. Once the invertible operations have been identified, the point of no return (PNR) must be found, which represents the point at which a rollback becomes infeasible. This can be either due to consistency or timing requirements. Ultimately, a rollback graph as displayed in Figure 2 is the goal. This graph represents all feasible reconfiguration and rollback sequences that can be performed to achieve the highest fault tolerance, i.e. a maximal number of faults can be tolerated and rolled back.

A. Invertible Operations

The reconfiguration sequence S was defined in Section II-D. We use the notation $S_{p,q}$ to denote the subsequence of sequence S of length n where $0 \leq p \leq q \leq n$. For every operation o_i within this sequence, the inverse operation must be identified. Table I details the inverse operations for some of the reconfiguration services defined in the IEC 61499. For some operations, no inversion is necessary, since there is no active change of state yet. For example, reading the state of a FB does not cause any modification. For other operations, an inversion is usually not feasible, unless a checkpoint or backup was made previously. For example, deleting a FB can lead to an irreversible loss of state. We assume that these operations can not be reversed, since they require additional efforts.

The reconfiguration sequence and the inverse operations can be assembled into a large rollback graph by considering how the sequential changes can be undone. From this graph, we can extract individual reconfiguration and rollback sequences

$$S_{0,f}^{\mathrm{RB}} = \langle o_0, \cdots, o_f, o_f^{-1}, \cdots, o_0^{-1} \rangle, \quad (5)$$

where o_f is the operation which failed. The goal is now to trim the rollback graph, until all reconfiguration and rollback sequences from this graph are both consistent and satisfy the real-time requirements.

B. Point of no return

At some point in the reconfiguration, a non-reversible modification has to be performed. Usually, this is the time when either the system is resumed and we let go of the control over the state of the system, or we irreversibly delete something. The reconfiguration and rollback sequence $S_{0,f}^{\mathrm{RB}}$ can only be assembled if the failure happened before the PNR. Once the PNR has been passed, the reconfiguration can not be reversed and the only options are to either continue the reconfiguration, or perform an emergency shutdown. There are two reasons why the PNR must be considered.

1) Non-Invertible Operations: Some operations can not be reversed. This is relevant for the deletion of FBs (unless there is a backup), but more importantly due to the consistency requirements posed in [4]. FBs are stopped and started from event source to event sink, and the reconfiguration is pushed through the system as new events arrive. This keeps the FBs in a consistent state during the reconfiguration. Starting a FB means that the state is released, and can be modified by external sources. As a result, once a FB is started, its state can change in an irreversible manner over which we have no control. Rolling back the state is no longer possible (unless explicitly considered). [1]

2) Real-time Constraints: Just because a reconfiguration can be rolled back consistently does not mean that it can be rolled back in real-time. As seen in [9], the laxity in practical applications of reconfiguration can be very small. Considering the worst-case scenario, in which a full sequence of invertible operations must be reversed, the length of the reconfiguration and rollback sequence $S_{0,f}^{\mathrm{RB}}$ can be twice as long as the initial reconfiguration. In practice, unless the laxity allows it, there must be a PNR after which a rollback can not be performed without violating the real-time constraints.

An example of a rollback graph that implements all feasible rollback sequences before the PNR is shown in Figure 2. Any failure before the PNR will result in a unique reconfiguration and rollback sequence, and the system can be returned to the previous state without noticeable interruptions.

C. Optimization

The rollback graph of feasible reconfiguration and rollback sequences in Figure 2 indicates all possible sequences before the PNR. In this case, the PNR is caused by the first *start* operation. However, there are other operations after the PNR that could be reversed. We can change the reconfiguration sequence S in a way that allows for more invertible operations before the occurrence of the PNR. In this section, we show a modified optimization problem that maximizes the fault-tolerance of the reconfiguration.

[1]Erlang solves this problem elegantly by explicitly providing rollback state transformations from *new* to *old*, which is a manual overhead.

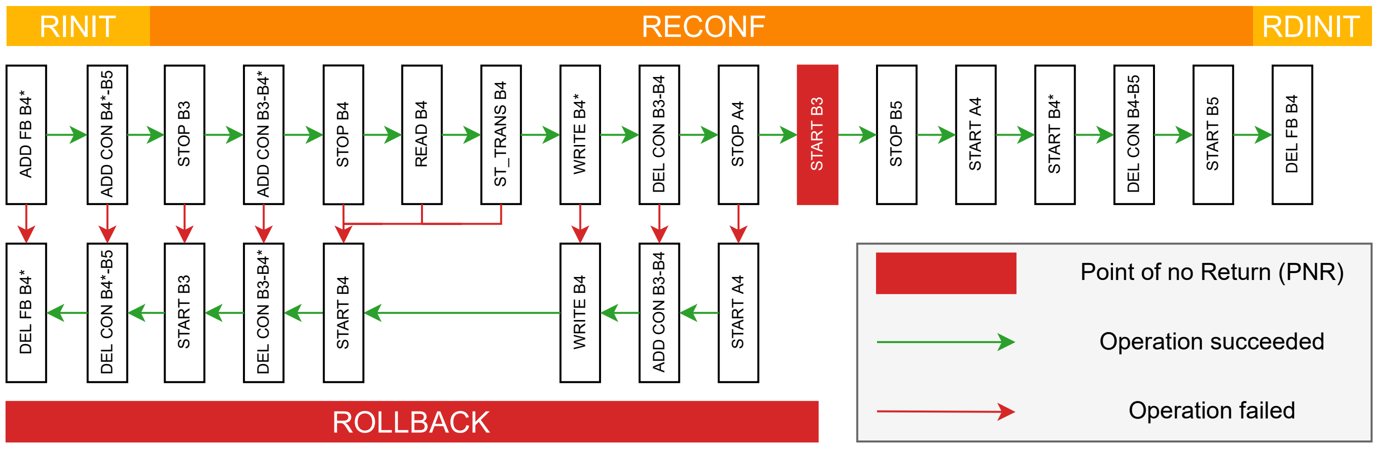


Fig. 2. The Point of no Return is determined by the first non-invertible operation if high deadlines guarantee timing constraints for each possible rollback sequence.

1) Rollback Length: The first metric to optimize is the length of the reconfiguration and rollback sequence. While previous works ([9]) minimized the blocking time, we maximize the number of operations within the sequence $S_{0,\text{PNR}}$:

$$\max_{S} |S_{0,\text{PNR}}|. \tag{6}$$

Since we have real-time constraints, we must additionally guarantee the satisfaction of these constraints. Thus, the laxity of all tasks τ_i must be positive (see Equation 3):

$$L(\tau_i) \overset{!}{\geq} 0. \tag{7}$$

The objective function in Equation 6 maximizes the fault tolerance, while the constraint in Equation 7 ensures the schedulability. As a result, we effectively increase the utilization to account for potential failures during the reconfiguration.

2) Rollback Length and Blocking Time: Focusing on the rollback length instead of the blocking time means that we no longer minimize the disruption of the real-time behaviors. This can lead to a higher utilization than necessary, since we may choose an ordering that blocks the execution longer than necessary. Thus, in practice, it can be desirable to minimize the blocking time as well.

We can incorporate this by first identifying the maximum number of operations that can be rolled back in real-time, and then minimizing the blocking time over all reconfiguration sequences with this number of operations:

$$\begin{aligned} \min_{S} \quad & \sum_{i=1}^{n} \frac{B^{R}(\tau_i)}{B^{R,\max}(\tau_i)} \\ \text{s.t.} \quad & L(\tau_i) \geq 0 \\ & |S_{0,\text{PNR}}| = \max_{S} |S_{0,\text{PNR}}| \end{aligned} \tag{8}$$

This two-staged optimization promises to find the reconfiguration sequence with the highest number of recoverable operations, while minimizing the disruption of the real-time behavior. This works well for small systems, yet fails to find a solution for more complex systems. In general, the problem could be solved by multi-objective metaheuristic optimization as well. A metaheuristic algorithm may not find the global optimum, yet this is not necessary, as long as the real-time constraints are satisfied and the level of fault-tolerance is acceptable.

IV. Evaluation

A. Example Scenarios

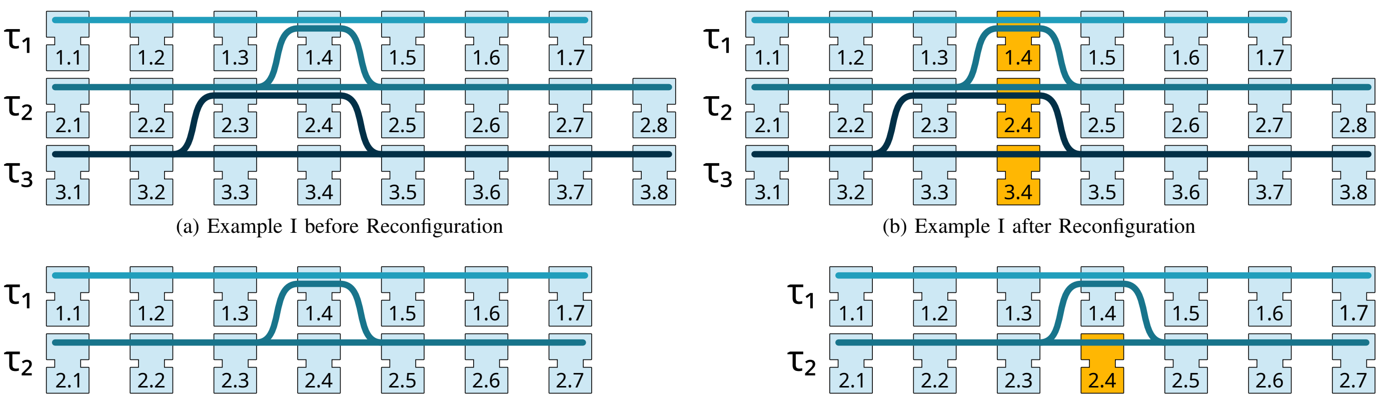
We evaluate our methodology on two example scenarios that demonstrate the benefits and limitations. The examples, their reconfigurations, and the concurrent tasks / execution traces / event chains are displayed in Figure 3. In Example I, there are three real-time tasks, and we exchange three FBs. In Example II, there are only two real-time tasks, and we exchange only a single FB. There are shared FBs in both examples, and the consistency requirements will demand a suspension of all real-time tasks in order to keep the state steady during the reconfigurations.

The reconfiguration sequences for Examples I and II consist of 53 and 17 operations, respectively. In the next section, we investigate how many operations can be rolled back before we encounter a PNR or real-time constraints are violated.

B. Results

The results of the reconfiguration- and rollback sequence generation are summarized in Table II. In both examples, even a simple heuristic algorithm that chooses the next operation in the reconfiguration sequence based on priorities ([4]) can satisfy the schedulability condition, and even allow a reasonable number of rollback steps in time, before the PNR happens.

Minimizing the blocking time, on the other hand, as suggested in [9], usually leads to a smaller number of feasible rollback operations, i.e. 24 to 11 operations before the PNR in Example I, and 9 to 8 in Example II. This can be explained by the idea that to minimize the blocking time, non-invertible operations are executed earlier. In particular, the blocking time is defined by the length between the first *stop* and the last *start* operation. Since a *start* operation is non-invertible (it



(a) Example I before Reconfiguration

(b) Example I after Reconfiguration

(c) Example II before Reconfiguration

(d) Example II after Reconfiguration

Fig. 3. Two example systems evaluate our methodology. In Example I, there are three concurrent real-time tasks, in which three FB are replaced. In Example II, one FB is replaced in two concurrent tasks.

TABLE II

We compare the time when a PNR occurs for both examples using four different algorithms that select a reconfiguration sequence from a dependency graph. Optimizing the blocking time alone leads to a less fault-tolerant reconfiguration.

Tasks				Heuristic [4]		Optimized for Blocking Time (BT) [9]		Optimized for Rollback Length (RBL)		Optimized for BT & RBL	
τ_i	$D(\tau_i) = T_i$	$C^\tau(\tau_i)$	$B^{\mathsf{FB}}(\tau_i)$	$L(\tau_i)$	$\lvert S_{0,\mathsf{PNR}}\rvert$	$L(\tau_i)$	$\lvert S_{0,\mathsf{PNR}}\rvert$	$L(\tau_i)$	$\lvert S_{0,\mathsf{PNR}}\rvert$	$L(\tau_i)$	$\lvert S_{0,\mathsf{PNR}}\rvert$
Example I											
τ_1	3000	350	50	1550.00	25	2470.00	12	1210.00	40		
τ_2	5000	400	100	1948.80	25	2698.80	12	1648.80	40	Infeasible	
τ_3	6000	400	0	1988.58	25	2508.58	12	1668.58	40		
Example II											
τ_1	800	350	50	150.00	9	360	8	90	11	360	11
τ_2	2200	400	0	80.04	9	80.04	8	80.04	11	80.04	11

breaks the consistency requirements), this will lead to less opportunities to rollback.

Optimizing the rollback length will lead to a maximum number of operations before the PNR. This yields 39 operations before the PNR for Example I, and 10 for Example II. On the other hand, the laxity of the tasks is smaller than both previous solutions. In this case, the PNR is postponed as far as possible.

Finally, it is possible to optimize the blocking time for all sequences with a maximum number of operations that can be rolled back. Due to the state-space explosion, we could only solve this for Example II, yet we can see that this combination can reach the same laxity as the pure blocking-time optimization, while achieving 10 operations that can be rolled back. This result is displayed in Figure 4.

C. Discussion

Selecting a suitable reconfiguration- and rollback sequence is a compromise. Generally, the search space is defined by the dependency graph and the real-time constraints. The heuristic algorithm proposed in [4] is very efficient in exploring the search space, yet the solution may violate the hard real-time constraints. Minimizing the blocking time will lead to the greatest laxity, however this also reduces the opportunities to rollback, since we execute irreversible operations (start) as soon as possible. Maximizing the rollback length minimizes the laxity as a side effect by trying to find a sequence in which we can fit the largest number of invertible operations before the PNR. Finally, combining the optimization of the blocking time and rollback length can be infeasible for complex systems. This limitation can be resolved by implementing a more efficient or metaheuristic algorithm that can efficiently explore the large search space.

These results ignore the problem that not all operations are equally rollback-worthy. To maximize the fault-tolerance, it is not only necessary to maximize the number of faults that can be tolerated, but it's important to consider the likelihood and criticality of each fault. We minimize the blocking time in Figure 4 by rearranging the operations. This reordering does not necessarily make the system more or less fault-tolerant, unless we are able to pin a failure risk to each operation.

In Figure 5, we demonstrate how the laxity of the reconfiguration depends on when we trigger a rollback. In this case, we have chosen shorter deadlines to reduce the laxity. In this case, the laxity of task τ_1 reaches zero if the rollback is triggered after 30 operations. A later rollback would only be possible if the timing constraints are relaxed. For example, it may be possible to temporarily slow down the operation of the system in a way that allows the relaxation of the timing constraints. This would provide additional laxity to maximize

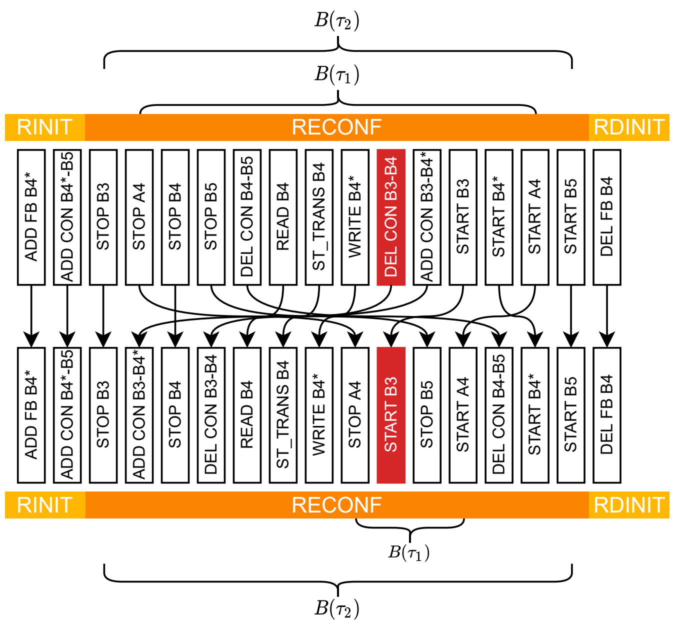


Fig. 4. Example II: Rearranging the operations inside the reconfiguration sequence S can increase the laxity of τ_1 by decreasing its blocking time while keeping the maximum rollback length.

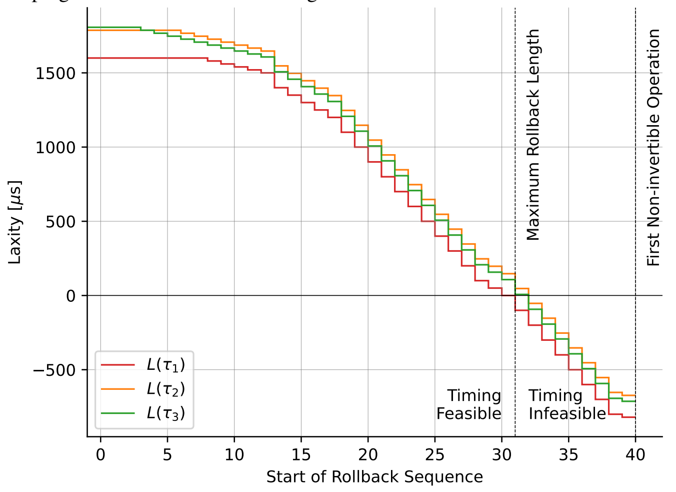


Fig. 5. For shorter deadlines in Example I ($D(\tau_1) = 2000$, $D(\tau_2) = 3500$, $D(\tau_3) = 4500$), the maximum rollback length is determined by timing constraints of the rollback sequence instead of the first non-invertible operation. The Point of no Return is the first operation with a negative laxity for one of its tasks if a rollback would be executed.

the fault-tolerance of the reconfiguration.

V. Conclusion

Dynamic Reconfiguration can improve the flexibility, agility, and resilience of ICS. Yet, it also represents a potential source of errors and bugs, and it may leave the system vulnerable to unexpected errors. In particular, external events during a reconfiguration can lead to issues. As a result, checkpointing or rollback mechanisms have been proposed in research and can be found in industry.

In this paper, we have analyzed the feasibility of rollback mechanisms for dynamic reconfiguration with the IEC 61499 standard. We have investigated disturbance of real-time behaviors, and proposed an optimization algorithm that can maximize the number of recoverable operations while preserving real-time performance.

The results indicate that the consideration of rollback sequences will require additional laxity in the system to prevent the violation of real-time constraints. Finding a feasible rollback sequence can be rather straightforward, the main difficulty is identifying the PNR, at which a rollback is no longer possible. This PNR can be either caused by irreversible changes to the system (such as a loss of state), or due to constraining real-time behaviors.

In the future, the further implementation of reconfiguration sequences and the corresponding rollback sequences in RCAs is an important part to bring this methodology into practice. This requires work on the implementation of the IEC 61499 reconfiguration services. Improving the performance of these services, e.g. by merging multiple operations into one, can reduce the required overhead of dynamic reconfiguration, and necessary rollback sequences.

References

[1] Oliver Hummer, Christoph Sünder, Alois Zoitl, Thomas Strasser, Martijn N Rooker, and Gerhard Ebenhofer. "Towards Zero-downtime Evolution of Distributed Control Applications via Evolution Control based on IEC 61499". In: *IEEE Conference on Emerging Technologies and Factory Automation*. IEEE, 2006, pp. 1285–1292.

[2] Oliver Hummer, Christoph Sünder, Thomas Strasser, Martijn N Rooker, and Gerold Kerbleder. "Downtimeless System Evolution: Current State and Future Trends". In: *IEEE International Conference on Industrial Informatics*. Vol. 2. IEEE, 2007, pp. 1077–1082.

[3] Christoph Sünder, Valeriy Vyatkin, and Alois Zoitl. "Formal Verification of Downtimeless System Evolution in Embedded Automation Controllers". In: *ACM Transactions on Embedded Computing Systems* 12.1 (Jan. 2013), pp. 1–17.

[4] Laurin Prenzel and Sebastian Steinhorst. "Automated Dependency Resolution in Dynamic Reconfiguration for IEC 61499". In: *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA)*. Västerås, Sweden: IEEE, 2021.

[5] Schneider Electric. *Universal Automation - A call for change*. Tech. rep. 998-21066447. Sept. 2020.

[6] Laurin Prenzel, Alois Zoitl, and Julien Provost. "IEC 61499 Runtime Environments: A State of the Art Comparison". In: *International Conference on Computer Aided Systems Theory*. Springer, 2019.

[7] Alois Zoitl. *Real-time Execution for IEC 61499*. en. Instrumentation, Systems, and Automation Society, 2009.

[8] Jeff Kramer and Jeff Magee. "The evolving philosophers problem: dynamic change management". In: *IEEE Transactions on Software Engineering* 16.11 (Nov. 1990), pp. 1293–1306.

[9] Laurin Prenzel, Simon Hofmann, and Sebastian Steinhorst. "Real-time Dynamic Reconfiguration for IEC 61499". In: *International Conference on Industrial Cyber-Physical Systems*. IEEE, 2022.

[10] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. "Priority inheritance protocols: an approach to real-time synchronization". In: *IEEE Transactions on Computers* 39.9 (Sept. 1990), pp. 1175–1185.

[11] Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. "A Rollback Mechanism to Recover from Software Failures in Role-based Adaptive Software Systems". In: *Companion to the first International Conference on the Art, Science and Engineering of Programming*. ACM, 2017.

[12] Ngoc-Tho Huynh, An Phung-Khac, and Maria-Teresa Segarra. "Towards reliable distributed reconfiguration". In: *Adaptive and Reflective Middleware on Proceedings of the International Workshop*. ARM '11. Lisbon, Portugal: Association for Computing Machinery, 2011.

[13] Richard Koo and Sam Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems". In: *IEEE Transactions on Software Engineering* SE-13.1 (Jan. 1987), pp. 23–31.

[14] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. "A theory of reversibility for Erlang". In: *Journal of Logical and Algebraic Methods in Programming* 100 (Nov. 2018), pp. 71–97.