Automated Dependency Resolution for Dynamic Reconfiguration of IEC 61499

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

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

***Abstract*—Dynamic reconfiguration and adaptability are crucial features in the evolution from automation to autonomy of industrial control systems. Component-based systems, such as specified in the IEC 61499, already provide a compelling framework for the distribution and transformation of software components, yet most reconfiguration approaches rely on a manual implementation of the required reconfiguration setup. We propose an automatic mechanism to generate the needed reconfiguration operations and order them into reconfiguration sequences, while preserving the dependencies of each operation, similar to the concepts of quiescence or version consistency. We further identify four scenarios for dynamic reconfiguration with different requirements regarding the treatment of state and showcase the results of our methodology on each scenario.**

***Index Terms*—IEC 61499, Topological Sorting, Dependency Management, Dynamic Update**

I. Introduction

Future industrial control systems must be adaptable and able to change autonomously without user interaction to tolerate failures or changing requirements [1]. This autonomous change includes the identification of disturbances, the decision-making processes to decide on the best course of action, and the ability to implement the needed changes. The IEC 61499 provides the basic infrastructure for distributed online change. This paper extends this infrastructure to automatically generate the reconfiguration sequences needed to execute the change.

The discussion on how to change a distributed system has been started over 30 years ago and is not yet solved [2]. Component-based systems add a layer of abstraction that facilitiates the work on methods of Dynamic Reconfiguration, especially in the domain of real-time distributed systems [3–7].

From the perspective of the control device, most PLCs follow the IEC 61131-3 standard, which is incompatible with this notion of component-based systems. By contrast, the IEC 61499 was modeled to fit into this niche and provides the basic infrastructure of component-based systems through its encapsulation and event-triggered execution.

The ability to change and adapt has been ingrained in the design of the IEC 61499. The underlying infrastructure for this ability has been a topic of discussion since the beginning of the IEC 61499 [8–11]. In combination with the automatic deployment configuration generation, as proposed by [12],

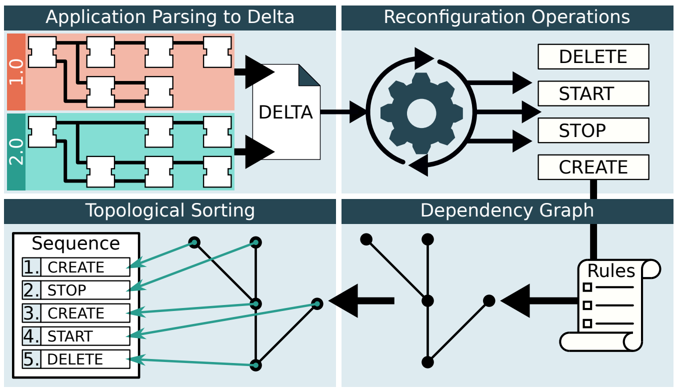

Fig. 1. From the IEC 61499 application, the event traces can be extracted. The difference between two applications yields the required reconfiguration operations for the change, which can be assembled into a reconfiguration sequence given the event traces.

this framework may enable the autonomous adaptation of control systems. Nevertheless, most existing works focus on the development process to create the reconfiguration applications or their verification, instead of the automatic generation of the reconfiguration applications. In fact, the manual implementation of reconfiguration applications is inherently difficult, error-prone, and can have dangerous consequences for safety-critical systems. Existing approaches for automatic dynamic updating procedures rely on discrete models and specifications that are usually not available in practice [13].

We envision that safe and simple dynamic reconfiguration processes are a crucial ingredient for future control systems, especially considering the transition from automatic to autonomous behavior. The existing approaches are insufficient in making automatic dynamic reconfiguration a feasible feature or are yet unable to bridge the gap from theory to practice.

In this paper, we propose a methodology to automatically generate reconfiguration sequences that allow the safe reconfiguration of distributed, component-based systems based on the IEC 61499 (Figure 1). We introduce a mechanism to extract the required reconfiguration operations from the difference between two applications and identify supplemental information. These operations may be organized into a dependency graph using a ruleset, and a linearized reconfiguration sequence can be extracted via topological sorting. The methodology is showcased on four reconfiguration scenarios, and an extension is proposed to identify and handle feedback loops.

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).

II. Background

A. IEC 61499 Execution Model

IEC 61499 applications are event triggered. The standard itself leaves room for interpretation, which resulted in a plethora of different implementations [14, 15]. A favorable real-time execution model was introduced by [16] through the event chain concept. An event chain is a trace of executions caused by an event at an *event source*, which eventually terminates in an *event sink*. This event chain can be implemented in a real-time thread. Alternative execution models implement every FB as their own process, which leads to more asynchronous behavior, but less predictable real-time behavior [17].

Event Loss: Despite the ambiguities of the execution model of the IEC 61499, it appears to be universally agreed upon that event loss should be avoided. The order of execution may depend on whether a cyclic, parallel, or sequential execution model is chosen, but in all semantics, the event should not be dropped [18]. Practically, the consideration of event loss would require the introduction of timeouts or handshakes and quickly blows up the execution control chart of an FB. As a result, losing events must be prevented during a reconfiguration, but the order of occurence may be affected.

B. Dynamic Reconfiguration

The topic of Dynamic Software Updating in general [19, 20] and updating Discrete Event Controllers in particular [13] has been addressed in detail. Yet, these solutions often require a formal specification using discrete models, which is rarely available, but allows an automatic, correct-by-design synthesis, if all necessary information is available. By contrast, this paper addresses the update of (potentially distributed) component-based architectures that are not limited to discrete semantics, and discusses the origin of the needed supplemental information with respect to the IEC 61499.

The ability to adapt and change has been ingrained in the IEC 61499 from the very beginning: The standard itself defines a set of reconfiguration services [21] for this purpose. The topic has been addressed in multiple publications, in particular regarding the feasibility and verification of a reconfiguration [8, 22]. In general, the reconfiguration services are implemented inside a reconfiguration application.

[11] introduces the three phases of a reconfiguration: The *startup* sequence, the *reconfiguration* sequence, and the *closing* sequence. In other publications, these phases are referred to as *Initialization (RINIT)*, *Reconfiguration (RECONF)*, and *Deinitialization (RDINIT)*, respectively [22]. The reconfiguration application was further structured into evolution steps that handle a specific *Evolution region of interest* [10]. In this scope, the concept of a dependency graph as an *evolution graph* was introduced, mainly as a means of visualization.

An alternative approach to dynamic reconfiguration was introduced in [23]. In this case, the IEC 61499 application was implemented in Erlang, which allows *Hot Code Loading* by design. Thus, to reconfigure this application, the existing Erlang mechanisms could be used, and the reconfiguration instructions had to be implemented manually. In [17], this was extended to allow the automatic generation of simple reconfigurations.

As of now, there exists no framework for the automatic reconfiguration of IEC 61499 applications. While manual reconfigurations are possible, and they could be verified, this procedure is cumbersome and errorprone. In this paper, we propose the automatic generation of reconfiguration sequences, which represents a stepping stone towards a fully-automatic reconfiguration.

III. Reconfiguration Scenarios

State in Dynamic Reconfiguration: A major hurdle in the dynamic reconfiguration of a component-based system is the handling of state [3]. In [4], the *continuity* property is introduced with respect to the state transformation. To achieve *continuity*, a service must be continued and partially executed services must be completed. In a traditional change scenario, in which the system is shut down and restarted, the state can be discarded and the new system can be restarted from a known, initial state. In the dynamic case, this *continuity* and *integrity* is a critical component. Given two arbitrary systems, the prospect of being able to elegantly transition from one to the other is rather bleak. Transforming a flight controller into an HVAC controller is difficult not because different components are used, but because of the difference in state and the difficulty to achieve *continuity*.

State Transformations: A *state transformation* offers the necessary information to transform the state, either by mapping state X to state Y, or by identifying suitable safe update points, e.g. *wait until state X and switch to state Y*. For a small class of problems, the needed state transformation can be identified automatically. In [4], the system behavior is modeled with interface automata, which are then used to identify a correct state transformation. This requires the availability of corresponding behavior models, and a framework to identify the state transformation. Other automatic frameworks require the availablility of formal specifications and manually created mapping functions [13]. In this paper, we characterize reconfiguration scenarios based on the state transformation *needs* from an architectural perspective, irrespective of their origin.

Reconfiguration Scenarios in IEC 61499: The system view of an IEC 61499 application does not comprise any information how another system may be transformed into this one or how it could be transformed into another one. Current behavior models are not suitable to automatically identify appropriate state transformations, although that may change in the future [24]. We identify four classes of reconfiguration scenarios with different requirements regarding the handling of state.

(I) Stateless In a stateless reconfiguration, the internal state does not matter and can be discarded, or there is no internal state. This is the case for robust processes, which can quickly recover the state, or for specific event FBs or simple FBs that contain little or no state.

(II) State mapping In most mapping reconfigurations, the application behavior is not changed but only the allocation of resources. In this scenario, the state can be mapped from one FB to another without transformation.

(III) State transformation (SISO) In the simple case of state transformation, the state of one FB is transformed into the state of one other FB, for example if the FB version is updated and the implementation changes slightly.

(IV) State transformation (MIMO) For complex changes, the state of multiple FBs can be transformed into the state of multiple others. This can be the case when two FBs are replaced by a single new one, or a subapplication is exchanged by another subapplication.

IEC 61131-3 Online Change: Current PLC software suites based on the IEC 61131-3 may encompass an *online change* feature [10]. Given the IEC 61131-3 execution model, this usually means that the current application program is exchanged for a new version, and the state is mapped, but not changed (Scenario II: State mapping). This allows only for very small changes and requires care of the developer to not cause catastrophic failures. The fragmented state of the IEC 61499 allows for much more fine-grained changes with less impact on the execution and more control over the state transformation.

IV. Reconfiguration Operations

In order to be able to change a component-based system, the necessary operations must be identified. We define a reconfiguration operation o as a tuple (Equation 1), where i is the instruction or service, t is the target, e.g. a component or a connection, and D is a set of operations that this operation depends on.

$$o = (i, t, D) \quad (1)$$

IEC 61499 Reconfiguration Services: The IEC 61499 defines a set of reconfiguration services and a state machine for the operational states of an FB. These services are specific to the runtime environment and may be provided by a set of corresponding management FBs. An exhaustive list of services is given by [16]. These management FBs can be assembled into a *Reconfiguration Application* or *Evolution Control Application* [16, 25].

In this paper, the IEC 61499 services are treated as the instructions to a reconfiguration operation. The operation to create a new FB would thus contain the `CREATE FB` service as the instruction i, the target t is the FB to be created, and D would indicate a set of dependencies that must be performed before the FB can be created.

Selected Reconfiguration Operations: The services as described by the IEC 61499 are intended to be exhaustive, i.e. they should allow any type of reconfiguration. In practice, most reconfigurations are going to be simplistic, such as tweaking a parameter. Even complicated reconfigurations do not typically need the full set of services, at least during normal operation. The `KILL` and `RESET` services, which can be used to interrupt a running FB and may lead to a corrupted state or event loss, are hardly compatible with most execution semantics as long

TABLE I
Description of the used reconfiguration services as defined in [16].

IEC 61499-1	Description
`CREATE FB`	Create a new function block.
`CREATE CON`	Create a new data / event connection.
`DELETE FB`	Delete a function block.
`DELETE CON`	Delete a data / event connection.
`START`	Set the operational state of the FB to `Started`.
`STOP`	Set the operational state of the FB to `Stopped`.
`READ`	Read the data inputs, outputs, and internal variables of a FB.
`WRITE`	Write the data inputs, outputs, and internal variables of a FB.

as *continuity* is desired. Yet, there may be specific edge cases in which their use is required.

In this paper, thus, a selection of these services is used (Table I). Most noteworthy, these are the services to create and delete FBs and connections. Further, the flow of events must be controlled, which can be achieved by setting the operational state of a FB to *stopped* or *started*. Finally, the internal state of FBs must be accessed by *reading* and *writing*. Some services are omitted from the methodology since they do not add any meaningful value. These are the services to create and delete types and resources, and they can always be performed during the *RINIT* or *RDINIT* phases. The methodology can be easily extended to incorporate other or new services.

A. Automatic Generation of Reconfiguration Operations

As identified by [10], most of the needed reconfiguration operations can be extracted from the difference between two applications, i.e. the *delta*. In this paper, we generate the operations automatically from the delta: After parsing the applications, their contents can be compared, and the operations needed to patch this difference are generated. Yet, the delta fails to capture the intention of the developer, and thus complicates the generation of the state transformation. In [4], the behavior models are used to automatically generate feasible state transformations. Given the lack of appropriate behavior models in the IEC 61499, we are assuming that any state transformation is supplied by the system developer until suitable models are available.

In Section III we have identified four scenarios with different needs for the state handling. In this section, we discuss how the operations for these scenarios can be generated.

1) Stateless Reconfiguration: For Scenario I, the stateless reconfiguration, the difference contains all needed information: The added and removed function blocks and connections. As a result, the needed `CREATE`, `START`, `STOP`, and `DEL` operations can be added. Every FB that is created must be started, and every FB that is deleted must be stopped.

2) State Mapping Reconfiguration: For Scenario II, not all information is given in the difference. `CREATE`, `DEL`, `START`, and `STOP` operations can be generated the same as in Scenario I. The mapping of one FB to another must be performed manually and supplied to the generation process. The necessary state mapping m_{map} is defined in Equation 2, where s is the source FB and t is the target FB. As a result, the necessary `READ` and `WRITE` operations can be added, where the state of s is read, and the state of t is written.

$$m_{\text{map}} = (s, t) \tag{2}$$

3) SISO State Transformation Reconfiguration: Scenario III can be treated similarly to Scenario II, with the exception of an additional state transformation operation added between the `READ` and `WRITE` operations. This state transformation operation must be provided by the application developer, and may be implemented as an FB. A definition of a state transformation m_{SISO} is given in Equation 3, where s is the source FB, t is the target FB, and f is the transformation function.

$$m_{\text{SISO}} = \big(s, t, f(\text{state}_s) \rightarrow \text{state}_t\big) \tag{3}$$

4) MIMO State Transformation Reconfiguration: The operations for Scenario IV are identical to Scenario III. In addition to the state transformation operation between two FBs, this operation must synchronize more than one input and/or output FB. A definition for the MIMO state transformation m_{MIMO} is given in Equation 4, where S is a set of source FBs, T is a set of target FBs, and f is the transformation function.

$$m_{\text{MIMO}} = \big(S, T, f(\text{state}_S) \rightarrow \text{state}_T\big) \tag{4}$$

V. Reconfiguration Sequences

After generating the reconfiguration operations from the *delta*, the operations must be assembled into the right order to preserve the *continuity* of the application. This preservation can be achieved by updating the system from *event source* to *event sink*. Traditionally, the IEC 61499 reconfiguration services were implemented in management FBs, which are assembled into reconfiguration applications. In this paper, the operations are mapped into a dependency graph, and a linearized sequence is extracted. In particular, the dependencies are added according to a set of rules, and the linearized sequence is found through topological sorting. The reconfiguration sequence could be implemented in a *reconfiguration application* or simply sent to a runtime service that executes the operations. We deliberately chose a linearized sequence to prevent concurrency issues and increase determinism, yet the dependency graph can be easily used to create a concurrent reconfiguration plan or configuration manager as described by [5].

Reconfiguration Sequence Definition: A reconfiguration sequence s is an ordered sequence of reconfiguration operations o_i that must be performed to reconfigure an application. It must preserve the order of the operations to prevent undesirable side effects. For example, a new function block must be started before it can be connected, and it should be fully connected before the old function block is deleted. Thus, the ordering of the sequence s must ensure that the dependencies of an operation o_i are satisfied by the preceding operations o_0 to o_{i-1}.

$$s = \Big[o_0, \cdots, o_n | o_i.D \subseteq \big\{o_0, \cdots o_{i-1}\big\}\Big] \tag{5}$$

A. Sequence Dependencies

Dependencies between *evolution regions of interest (EROI)* were introduced by [10]. In this paper, we propose the introduction of dependencies between the reconfiguration operations themselves. These dependencies lead to the identification of the EROI, since by definition, an EROI is a region that can be reconfigured independently. If a set of operations can be executed independently, this set can be treated as an EROI.

Dependency Rules: The dependencies can be injected through a set of rules. The rules are applied to the set of reconfiguration operations and yield the dependencies of each particular operation. Once all dependencies have been added, a dependency graph can be used for visualization, and a suitable reconfiguration sequence can be extracted via topological sorting.

A list of dependency rules is given in Table II. These rules are applied to any operation in the reconfiguration sequence, i.e. if an operation with a `CREATE FB` instruction is detected for a target FB, it must occur before a corresponding `START` operation for this particular FB. In this method, any operation may only appear once in a reconfiguration sequence, i.e. an FB can not be started and stopped two consecutive times since the rules can not differentiate between the operations.

The `START` and `STOP` ordering rules ensure that all FBs are stopped and started from event *source* to *sink*. In this manner, the change can propagate through the application while old events are processed in the old version, and new events are processed in the new version. These rules add dependencies for the event and data connections in the application. In the case of feedback loops, a circular dependency is created that cannot be resolved. A solution to this problem is discussed in Section VII.

An example of a resulting dependency tree is given in Figure 2. As can be seen, the dependencies leave several implementation choices for the order of execution. The dependencies also allow a transparent and explainable reasoning, why a specific operation is or is not currently allowed. This facilitates a conversation between the automatic tooling and the system developer, who can reproduce why a reconfiguration is or is not possible.

B. Sequence Ordering

Once the reconfiguration operations are created and the dependencies are injected, the operations can be sorted by a simple topological sorting algorithm based on Kahn's algorithm [26], in which the operations are sorted by the dependency first, and a static priority second. The priorities are defined for every instruction i, e.g. `START` has a higher priority than `STOP`.

The sequence ordering according to the dependencies guarantees that any dependency is fulled, i.e. that any execution will

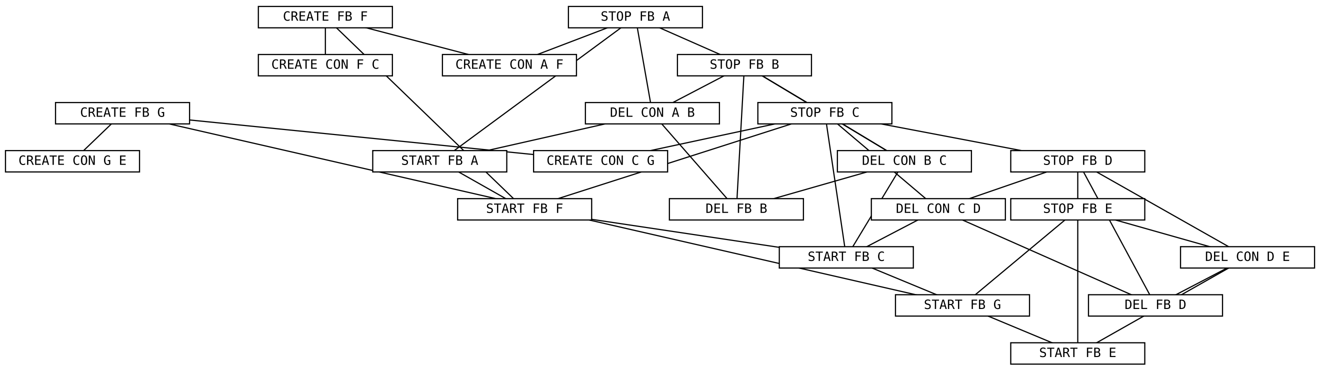

Fig. 2. A dependency tree for the reconfiguration operations of Scenario 1 (Figure 3). A node represents an operation, an edge indicates that the lower operation depends on the higher one.

TABLE II
DEPENDENCY RULES FOR THE INJECTION OF DEPENDENCIES INTO THE OPERATIONS. THE RESULTING DEPENDENCIES ALLOW THE SORTING OF THE OPERATIONS.

Rule	**Description**
`STOP` before `START`	Existing FBs must be stopped before they can be started.
`CREATE FB` before `START`	New FBs must be created before they can be started.
`STOP` before `DEL`	Any FB must be stopped before it can be deleted.
`START` ordering	All FBs must be started in the order of the FB connections.
`STOP` ordering	All FBs must be stopped in the order of the FB connections.
`STOP` before `CREATE CON`	Any FB must be stopped before its connections can be created.
`STOP` before `DEL CON`	Any FB must be stopped before its connections can be deleted.
`CREATE FB` before `CREATE CON`	Any FB must be created before its connections can be created.
`DEL CON` before `DEL FB`	Any connection must be deleted before the FB can be deleted.
`DEL CON` before `START`	Any connection must be deleted before the FB can be started.
`STOP` before `QUERY`	Any FB must be stopped before its state can be queried.
`STOP` before `WRITE`	Any FB must be stopped before its state can be written.
`WRITE` before `START`	Any state must be written before the FB can be started.

be performed either by the old version, or the new version. No events are lost or have to be discarded. The priority ensures that given two choices, the more urgent operation is performed. In general, it is preferable to `START` as soon as possible and `STOP` as late as possible to minimize the disturbance. This algorithm works well for many scenarios, but can suffer from priority inversion. In particular, a higher priority operation (`START`) could be blocked by a lower priority operation (`DEL CON`). The general problem could be extended to an optimization problem if an appropriate cost function was introduced, e.g. minimizing the disturbance from `STOP` to `START`. This would require the quantified disturbance per operation and could also incorporate communication overhead. This optimization problem is not considered in this paper.

VI. Evaluation of Scenarios

The feasibility of using the dependency graph to generate a topological sorting is exhibited in four scenarios as identified in Section III. For each scenario, the necessary operations and the potential impact on the execution are presented. For the sake of brevity, only the relevant FBs of an application are displayed. All other function blocks in the application are able to continue uninterruptedly unless they have connections through the affected FBs.

Reconfiguration Phases: In each scenario, the different phases according to [11] are indicated. The startup sequence contains the creation of new FBs and connections. During the *reconfiguration* sequence, the operation is disrupted. This is marked by the first `STOP` operation, and ends with the reversal of this operation. In the *closing* sequence, remaining connections and FBs are deleted and further FBs are started. Some services, such as `CREATE TYPE` or `CREATE RES` are omitted in this paper. These operations would always occur during the *RINIT* or *RDINIT* phases.

A. Scenario I: Stateless Reconfiguration

In Figure 3, a reconfiguration is expressed in which two FBs are replaced by two other FBs. In this particular scenario, the exchange is considered to be stateless, i.e. the states of FBs `B` and `D` can be discarded, and FBs `F` and `G` can be started from their initial states.

Operations: The reconfiguration requires operations to create, start, stop, and delete the affected FBs in the correct order. According to the flow of events and data, the reconfiguration must occur from FB `A` to FB `E`, i.e. the FBs must be stopped from left to right and started from left to right. In this manner, the integrity and continuity of the event flow can be guaranteed. The dependency tree for this scenario is given in Figure 2.

Impact: After the new FBs are added, the two FBs can be exchanged one after another. This allows the reconfiguration of FB `B` to start, while FB `D` can still process old events. Similarly, FB `F` can already continue, while FB `G` is still under reconfiguration.

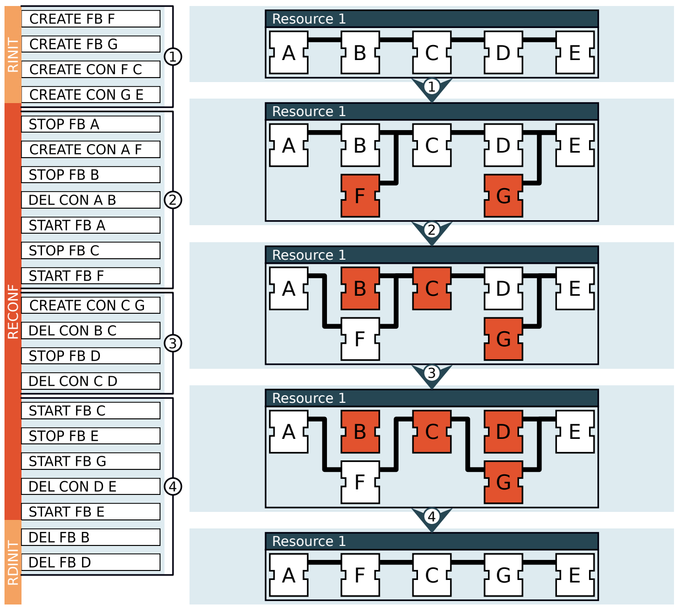

Fig. 3. Scenario I: Function blocks `B` and `D` are replaced by function blocks `F` and `G`, respectively. A resulting reconfiguration sequence with the *RINIT*, *RECONF*, and *RDINIT*-phases is displayed on the left. The evolution of the application is displayed on the right. In this scenario, no state is carried over.

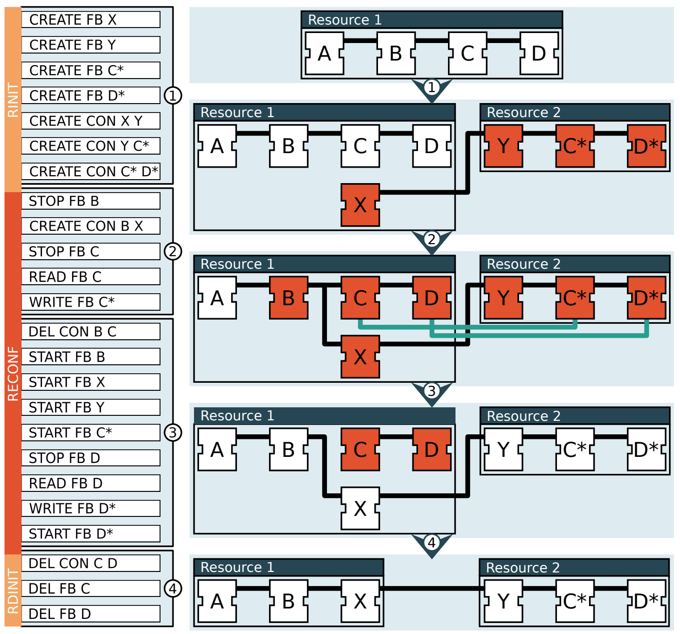

Fig. 4. Scenario II: The mapping of FBs to resources is reconfigured. Throughout the reconfiguration, the *continuity* must be preserved. Thus, the internal state of FB `C` must be read and written before `C*` can be started.

B. Scenario II: FB Mapping Reconfiguration

The second scenario investigates the reconfiguration sequences for the redistribution of FBs from one resource to another (Figure 4). Two function blocks from Resource 1 (`C`, `D`) are shifted to Resource 2, which requires the addition of two communication FBs (`X` and `Y`).

Operations: Similar to Scenario I, the operations to create, start, stop, and delete are added as needed. In addition, two operations to read and write are added as well, to map the state between the resources.

Impact: Initially, the new function blocks and connections are added. During the critical phase, FB `B` is stopped to prevent the further execution. FB `C` is stopped first and the state is read and written to FB `C*`. This allows FBs `X`, `Y`, and `C*` to be started while the state from `D` is read and written to `D*`. If concurrent or parallel execution were possible, some of these operations could be performed in parallel, thus further reducing the overhead. It is noteworthy that the distinction between the *RECONF* and *RDINIT* phase starts to blur, as the execution can be reinstated continuously.

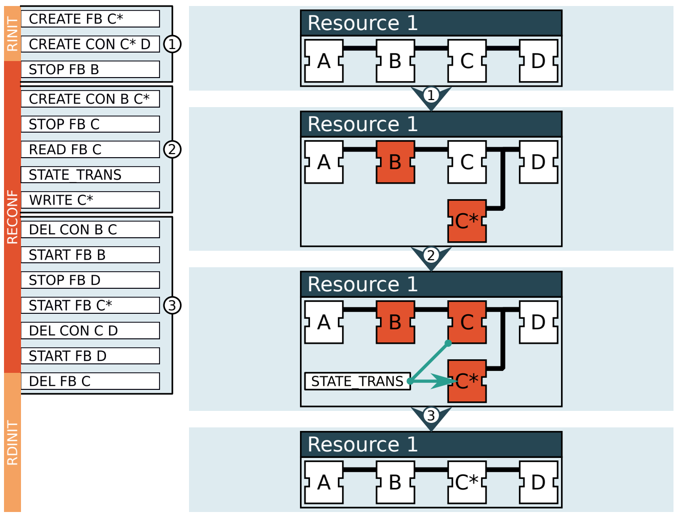

Fig. 5. Scenario III: The version of a function block is changed, thus requiring an explicit state transformation between the stopping of FB `C` and the start of FB `C*`.

C. Scenario III: SISO State Transformation

In Scenario III (Figure 5), an FB is replaced by another FB with an explicit state transformation. This new FB may be a new version of the same type, or another FB altogether. The key difference to Scenario II is the explicit transformation of the state between the reading and the writing of the state.

Operations: A state transformation operation is added to explicitly transform the state between the `READ` and `WRITE` operations. Other operations remain similar to scenarios I and II.

Impact: This change has a particularly small impact since only the FB before the one to be exchanged must be suspended momentarily. The biggest difficulty is the identification of the state transformation, which can be performed offline. Given the fragmented state of the IEC 61499 application, the state of an FB is small and consists of the ECC state and the set of variables (input, output, internal).

D. Scenario IV: MIMO State Transformation

In the MIMO state transformation case, the state of multiple FBs is needed to reconfigure one or multiple FBs. In this particular scenario, the state of three FBs must be read and transformed to reinitialize a new FB. FB `B*` replaces FBs `B` and `E`, while also requiring the state of `C`.

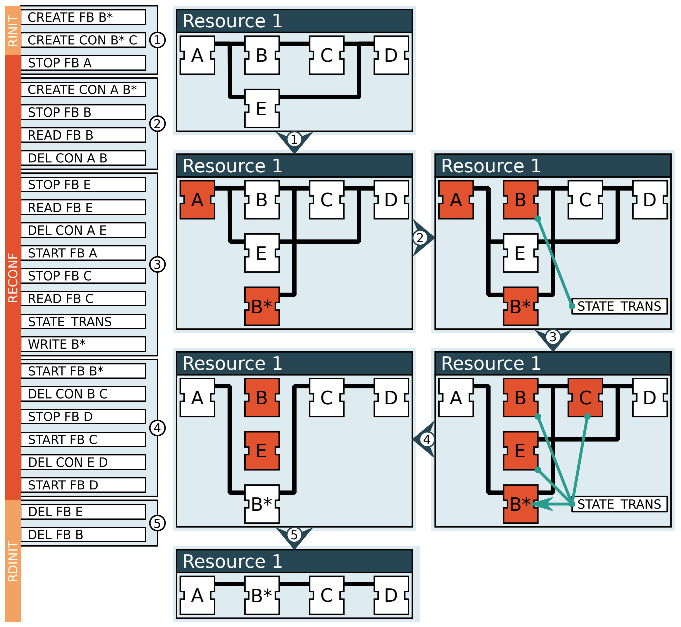

Fig. 6. Scenario IV: The version of a function block is changed, thus requiring a state transformation.

Operations: In addition to the operations discussed before, the state transformation in this scenario requires the synchronization with three `READ` operations. Thus, the state transformation can only start once the states of FBs `B`, `C`, and `E` are read.

Impact: Stopping FB `A` suspends the arrival of new events, while previous events can still be processed. Once the connections are rewired to FB `B⋆`, FB `A` can be restarted, and FB `B⋆` is started as soon as the state is transformed. The required inputs of the state transformation lead to a further synchronization of the sequence, thus also increasing the overall impact.

E. Discussion

The reconfiguration scenarios show how the methodology enables the automatic generation of reconfiguration sequences that preserve the *continuity* of the application throughout the reconfiguration. The changes in the applications are swept through the FB networks from source to sink. They also indicate the elements that currently cannot be automated: Namely, the state mapping and state transformation. This information is not presently included in the IEC 61499 application, but is represented by the intent of the system developer. Nevertheless, this added layer of abstraction simplifies and facilitiates the utilization of dynamic reconfiguration for a range of reconfiguration scenarios.

VII. Outlook: Transient Reconfiguration Sequences

The usage of the dependency tree based on the FB connections works well for systems without cycles. In control applications, feedback loops are a common occurrence. Thus, to be able to achieve a fully automatic generation of reconfiguration sequences, these feedback loops must be handled as well.

Feedback Loop Identification: The first problem of handling feedback loops is their detection. A feedback loop is defined by its trace (in the form of a directed, acyclic graph) through the application, and the system state in which it occurs. In an IEC 61499 application, these loops are defined implicitly through the connection network and the behavior of the FBs. By definition, every execution trace must terminate eventually, thus every feedback loop should also terminate eventually. This leaves several options to identify the relevent feedback loops:

1) Manual identification / specification
2) Exploration of the IEC 61499 application
3) Exploration of an IEC 61499 behavior model
4) Measurements on a running system

Manual identification and measurements on a real system lack the necessary exhaustiveness, while the exploration of the actual application will quickly run into a state space explosion given the lack of boundaries typically set by the physical process under control. The most promising approach would utilize a behavior model of the IEC 61499 application, which must include a model of the physical process as well. Recently, there has been an increased interest in behavior modeling, which could offer solutions to his problem [24, 27].

Breaking the loop: Once the trace of the loop and the state in which it occurs is known, this information can be used to devise whether it warrants an action, and what action can be used to split this cycle. The goal of this procedure is to identify a way to split the loop into *old* and *new* behavior.

To break the loop, a guard could be added to reach a *quiescent* [2]. A guard would block the further execution of a reconfiguration sequence until a condition is fulfilled, e.g. a specific state of an FB is reached. The methodology of this paper could easily be extended to incorporate new rules and dependencies, yet the implementation of these guards in an IEC 61499 runtime environment would require more effort. Alternatively, the *passive* state could be adapted to allow the resolution of cycles, while preventing new executions [2].

Limitations: There are two limitations to this methodology. The first is the issue of long blocking guards that can prolong the reconfiguration sequence, especially during the critical *RECONF* phase. Offline verification and validation methods should be used to ensure that any guard can be satisfied within a suffiently short time, or that the overall disruption is compatible with the real-time properties of the application. Secondly, even if a method to break feedback loops is used, it is possible to develop applications that are extraordinarily difficult to reconfigure, e.g. deeply interconnected systems with strict hard real-time requirements and minimal margins.

Obviously, this synchronization of feedback loops represents an additional impact on the real-time execution of the application. Nevertheless, if events should not be lost and the integrity and consistency of the application should be preserved, they must be handled appropriately.

VIII. Conclusion and Future Works

The transition from traditional manufacturing systems to more adaptable or even (partly) autonomous systems is inevitable. A major step in this direction is taking the human out of the loop and automating the reconfiguration procedure as much as possible. Dynamic reconfiguration as a topic has been addressed in the scope of component-based systems, the IEC 61499, and even in industrial applications with the IEC 61131-3. It is, thus, no longer a question of *if* it will be used more frequently, but *when*.

In this paper, we proposed a methodology to automatically generate reconfiguration sequences for component-based systems, such as the IEC 61499. These sequences may be implemented in various forms, such as reconfiguration applications. By automatically generating the required operations and ordering them according to their dependencies, we are able to guarantee the *continuity* of the provided services.

We also provided an outlook on future extensions of this methodology. The reconfiguration sequences are a single linearization of the dependency tree. Optimization algorithms could be used to minimize the disruption of the application during the reconfiguration if the disruption can be quantified appropriately. The handling of feedback loops can be solved by integrating guards into the sequences.

References

[1] Laurin Prenzel and Sebastian Steinhorst. “Decentralized Autonomous Architecture for Resilient Cyber-Physical Production Systems”. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2021)*. IEEE. Grenoble, France, 2021, pp. 1300–1303.

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

[3] Andreas Rasche and Andreas Polze. “Dynamic Reconfiguration of Component-based Real-time Software”. In: *IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE, 2005, pp. 347–354.

[4] Valerio Panzica La Manna. “Dynamic Software Update for Component-based Distributed Systems”. In: *Proceedings of the 16th International Workshop on Component-oriented Programming*. ACM, 2011, pp. 1–8.

[5] Michel Wermelinger. “A hierarchic architecture model for dynamic reconfiguration”. In: *International Workshop on Software Engineering for Parallel and Distributed Systems*. IEEE, May 1997, pp. 243–254.

[6] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D’Hondt. “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates”. In: *IEEE Transactions on Software Engineering* 33.12 (2007), pp. 856–868.

[7] Luciano Baresi, Carlo Ghezzi, Xiaoxing Ma, and Valerio Panzica La Manna. “Efficient Dynamic Updates of Distributed Components Through Version Consistency”. In: *IEEE Transactions on Software Engineering* 43.4 (2017), pp. 340–358.

[8] Christoph Sünder. “Evaluation of downtimeless system evolution in automation and control systems”. PhD thesis. Technische Universität Wien, 2003.

[9] Thomas Strasser, Alois Zoitl, Franz Auinger, and Christoph Sünder. “Towards Engineering Methods for Reconfiguration of Distributed Real-Time Control Systems Based on the Reference Model of IEC 61499”. en. In: *Holonic and Multi-Agent Systems for Manufacturing*. Lecture Notes in Computer Science. Springer, 2005, pp. 165–175.

[10] 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.

[11] Christoph Sünder et al. “Towards Reconfiguration Applications as basis for Control System Evolution in Zero-downtime Automation Systems”. In: *Intelligent Production Machines and Systems*. Oxford: Elsevier Science Ltd, 2006, pp. 523–528.

[12] Tarik Terzimehić. “Optimization and Reconfiguration of IEC 61499-based Software Architectures”. In: *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS ’18. Copenhagen, Denmark: ACM, 2018, pp. 180–185.

[13] Leandro Nahabedian et al. “Dynamic Update of Discrete Event Controllers”. In: *IEEE Transactions on Software Engineering* (2018), pp. 1220–1240.

[14] Thomas Strasser, Alois Zoitl, James H Christensen, and Christoph Sünder. “Design and Execution Issues in IEC 61499 Distributed Automation and Control Systems”. In: *IEEE Transactions on Systems, Man and Cybernetics* 41.1 (2011), pp. 41–51.

[15] 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, pp. 453–460.

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

[17] Laurin Prenzel and Julien Provost. “FBBeam: An Erlang-based IEC 61499 Implementation”. In: *International Conference on Industrial Informatics*. IEEE, 2019.

[18] Valeriy Vyatkin. “The IEC 61499 standard and its semantics”. In: *IEEE Industrial Electronics Magazine* 3.4 (2009), pp. 40–48.

[19] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. “A survey of dynamic software updating”. In: *Journal of Software: Evolution and Process* 25.5 (May 2013), pp. 535–568.

[20] Razika Lounas, Mohamed Mezghiche, and Jean-Louis Lanet. “Formal methods in dynamic software updating: A survey”. In: *International Journal of Critical Computer-Based Systems* 9.1-2 (2019), pp. 76–114.

[21] International Electrotechnical Commission. *IEC 61499*. Tech. rep. International Electrotechnical Commission, Nov. 2012.

[22] 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.

[23] Laurin Prenzel and Julien Provost. “Dynamic Software Updating of IEC 61499 Implementation Using Erlang Runtime System”. In: *World Congress of the International Federation of Automatic Control*. Vol. 50. Elsevier, 2017.

[24] Bianca Wiesmayr and Alois Zoitl. “Requirements for a dynamic interface model of IEC 61499 Function Blocks”. In: *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE, 2020, pp. 1069–1072.

[25] Oliver Hummer et al. “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.

[26] Arthur B Kahn. “Topological sorting of large networks”. In: *Communications of the ACM* (1962).

[27] Duc Do Tran, Jörg Walter, Kim Grüttner, and Frank Oppenheimer. “Towards Time-Sensitive Behavioral Contract Monitors for IEC 61499 Function Blocks”. In: *IEEE Conference on Industrial Cyberphysical Systems (ICPS)*. Vol. 1. IEEE, 2020, pp. 27–34.