A New Paradigm for the Enactment and Dynamic Adaptation of Data-Driven Process Structures*

Dominic Müller[1,2], Manfred Reichert[1,3], and Joachim Herbst[2]

[1] Institute of Databases and Information Systems, Ulm University, Germany
{dominic.mueller,manfred.reichert}@uni-ulm.de
[2] Dept. GR/EPD, Daimler AG Group Research & Advanced Engineering, Germany
joachim.j.herbst@daimler.com
[3] Information Systems Group, University of Twente, The Netherlands

Abstract. Industry is increasingly demanding IT support for large engineering processes, i.e., process structures consisting of hundreds up to thousands of processes. Developing a car, for example, requires the coordination of development processes for hundreds of components. Each of these development processes itself comprises a number of interdependent processes for designing, testing, and releasing the respective component. Typically, the resulting process structure becomes very large and is characterized by a strong relation with the assembly of the product. Such process structures are denoted as *data-driven*. On the one hand, the strong linkage between data and processes can be utilized for automatically creating process structures. On the other hand, it is useful for (dynamically) adapting process structures at a high level of abstraction. This paper presents new techniques for (dynamically) adapting data-driven process structures. We discuss fundamental correctness criteria needed for (automatically) detecting and disallowing dynamic changes which would lead to an inconsistent runtime situation. Altogether, our COREPRO approach provides a new paradigm for changing data-driven process structures at runtime reducing costs of change significantly.

Keywords: Process Coordination, Data-driven Process, Process Adaptation.

1 Introduction

In the engineering domain, the development of complex products (e.g., cars or airplanes) necessitates the coordination of thousands of processes (e.g., to design, test and release each product component). These processes and the many dependencies between them form complex and large *process structures*. While the single processes are usually implemented within different IT systems, the

* This work has been funded by *Daimler AG Group Research* and has been conducted in the *COREPRO* project (http://www.uni-ulm.de/dbis)

Z. Bellahsène and M. Léonard (Eds.): CAiSE 2008, LNCS 5074, pp. 48–63, 2008.

design, coordination, and maintenance of process structures are only rudimentarily supported by current process management technology [1]. In most cases, the different processes have to be manually composed and coordinated.

Another challenge constitutes the dynamic adaptation of process structures during runtime. When adding or removing a car component (e.g., a navigation system), for example, the process structure has to be adapted accordingly; i.e., processes for designing, testing and releasing affected product components have to be manually added or removed. Note that this also might require the insertion or removal of their synchronization dependencies with respect to other processes from the total process structure. Manually modeling and adapting large process structures requires profound process knowledge and is error-prone as well. Incorrectly specified dependencies within a process structure, however, can cause delays or deadlocks blocking the execution of the whole process structure.

Example 1. *To identify practical requirements, we investigated a variety of process structures in the automotive industry. As example consider release management (RLM) processes for electrical systems in a car. The (product) data structure (or configuration structure) for the total electrical system consists of up to 300 interconnected components which are organized by using a product data management system. The goal of RLM is to systematically test and release the different product components at a specific point in time, for example, when a certain milestone is reached. To verify the functionality of the electrical system, several processes (e.g.,* testing *and* release*) have to be executed for each electrical (sub-)component and need to be synchronized with the processes of other components. This is needed, for example, to specify that an electrical system can only be tested if all its components are individually tested before. Altogether, more than 1300 processes need to be coordinated for releasing the total electrical system [2].*

Interestingly, large process structures often show a strong linkage with the assembly of the product; i.e., the processes to be coordinated can be explicitly assigned to the different product components (cf. Example 1). Further, process synchronizations are correlated with the relations existing between the product components. In the following, we denote such process structures as *data-driven*. COREPRO utilizes the information about a product, its components and the component relations. In particular, COREPRO provides techniques for the modeling, enactment, and (dynamic) adaptation of process structures based on given (product) data structures. For example, the assembly of the (product) data structure can be used to automatically create the related process structure [2]. We have shown that COREPRO reduces modeling efforts for RLM process structures (cf. Example 1) by more than 90% [2].

When changing a (product) data structure at buildtime, the related process structure can be automatically re-created. However, long running process structures also need to be adapted during runtime. Then, it does not make sense to re-create the process structure from scratch and to restart its execution from the beginning. Instead, in-progress process structures must be adaptable on-the-fly, but without leading to faulty synchronizations like deadlocks. A particular challenge is to enable users to adapt complex process structures. When the product

structure is changed, we can take benefit from the strong linkage between process structure and (product) data structure again. In COREPRO, data structure changes are automatically translated into adaptations of the corresponding process structure. Thus, changes of data-driven process structures can be introduced by users at a high level of abstraction, which reduces complexity as well as cost of change significantly. Note that this is far from being trivial considering the large number of processes to be coordinated and their complex interdependencies.

In this paper we introduce the COREPRO runtime framework, which addresses the aforementioned requirements. COREPRO enables enactment of data-driven process structures based on a precise and well-defined operational semantics. We show how to translate (dynamic) changes of a currently processed data structure into corresponding adaptations of the related process structure. Finally, we introduce consistency criteria in order to ensure that dynamic adaptations of a process structure lead to a correct process structure again. Altogether, COREPRO addresses the full life cycle of data-driven process structures including modeling, enactment, and (dynamic) adaptation phases.

Sect. 2 describes the COREPRO modeling approach for data-driven process structures and Sect. 3 specifies operational semantics for process structures. Sect. 4 shows how to adapt data-driven process structures and presents methods for detecting invalid changes that would lead to incorrect runtime situations. We discuss related work in Sect. 5 and conclude with a summary in Sect. 6.

2 COREPRO Modeling Framework

The COREPRO modeling framework allows to create data-driven process structures (cf. Fig. 1d and f) [2]. For this, we consider the sequence of states a (data) object goes through during its lifetime. A product component from Example 1 passes states like *designed*, *tested* and *released*. Generally, state transitions are triggered by executing processes modifying the respective object (e.g., *design*, *test* or *release*). An *object life cycle* (OLC) constitutes an integrated and user-friendly view on the states of a particular object and its manipulating processes (cf. Fig. 1d and f). Regarding data-driven process structures, OLC state transitions do not only depend on the processes associated with the respective object, but also on the states and state transitions of other objects. As example consider a total car, which can be only tested, if all sub-systems (e.g., engine, chassis and navigation system) are tested before. By connecting the states of different OLCs, a logical view on a data-driven process structure results (cf. Fig. 1d and f).

Example 2. *The total electrical system of a car (cf. Example 1) differs from car series to car series (e.g., series with and without navigation system). The strong relationship between data structures and process structures implies that different (product) data structures (related to the different series) lead to different RLM process structures (e.g., process structures with and without development processes for the navigation system). Each of these process structures then represents one instance of the total development process for a particular car series.*

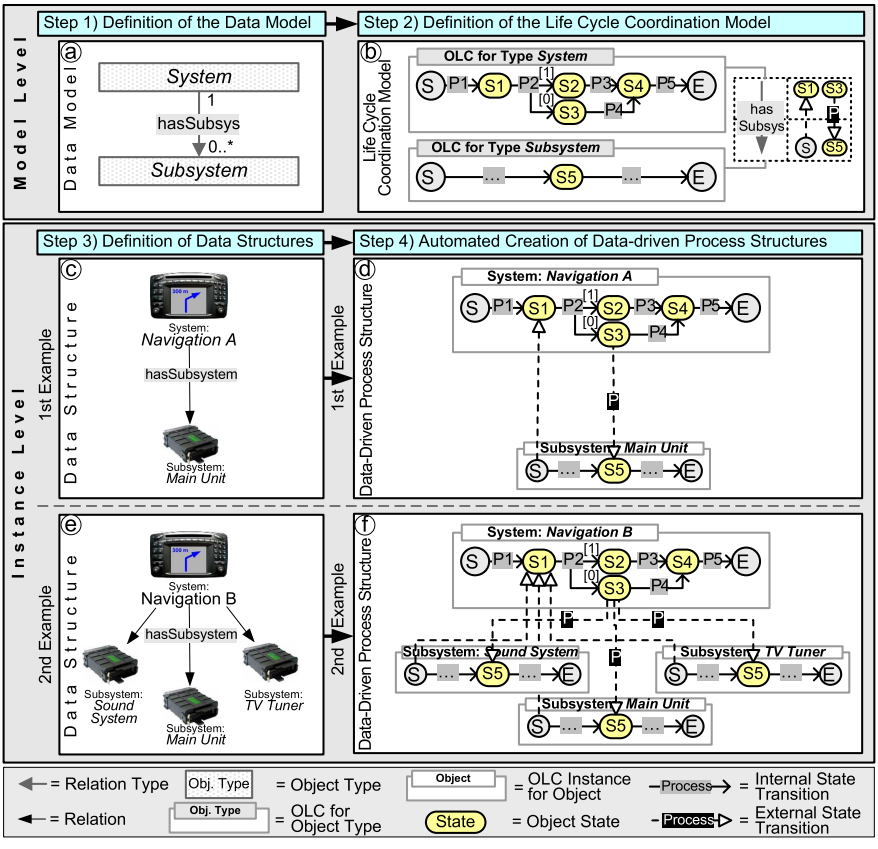

Fig. 1. Creation of Data-Driven Process Structures

Modeling OLCs for hundreds of objects (cf. Example 1) and linking them manually would be too expensive. Therefore, COREPRO follows a model-driven (respectively data-driven) approach by differing between model and instance level. First, for objects with same behavior, we model only one OLC. This OLC can be instantiated multiple times. Second, we utilize the close relationship between data and process structure when connecting different OLCs. Our case studies have shown that semantic relations between data objects can be mapped to dependencies between particular states of OLCs. Based on object (types) and relations between them, process structures can be specified at model level and then be automatically instantiated for every given data structure [2].

2.1 Defining Object Types and Object Life Cycles

Fig. 1 shows the steps necessary to model and instantiate both data and process structures. Steps 1 + 2 deal with the creation of the model level, Step 3 + 4 with the definition of the instance level. Step 1 is related to the specification of the

data model, which defines object and relation types, and therefore constitutes the schema for instantiating concrete (product) *data structures*. In our context, an object type represents a class of objects within the data model (cf. Fig. 1a), which then can be instantiated multiples times. Fig. 1e shows a data structure which comprises one instance of type `System` and three instances of type `Subsystem`.

Step 2 is related to the modeling of object life cycles (OLC) and their dependencies. An OLC defines the states through which objects of the respective type go during their life time. To capture the dynamic aspects of subsystems like `TV tuner` and `Sound System`, the process designer models only one OLC for the object type `Subsystem`. COREPRO maps OLCs to *state transition systems* whose states correspond to object states and whose (internal) state transitions are associated with object-related processes. A state transition does not fire immediately when its source state becomes activated, but waits until its associated process has completed. In Fig. 1b, for example, the OLC for object type `System` starts with initial state `S`, then passes state `S1` followed by `S2` or `S3`, and finally completes in state `E`. The *internal state transition* from `S` to `S1` takes place when finishing process `P1`. Note that the procedures for modifying the different objects might slightly differ (e.g., different testing procedures for `Sound System` and `TV Tuner`). If the OLCs are identical for such objects, this behavior can be realized as variants within the processes to be executed. Non-deterministic state transitions are realized by associating different internal state transitions with same source state and process, and by adding a process result as condition (e.g., `P2` in Fig. 1b). This is required, for example, when associating a testing process with a component, which completes either with result *correct* or *faulty*. Depending on the process result, exactly one internal state transition is chosen and its target state becomes activated (cf. `S2` and `S3` in Fig. 1b). The other internal state transitions are disabled (cf. Sect. 3).

2.2 Modeling Object Relations and OLC Dependencies

Defining the dynamic behavior for each object type by modeling its OLC is only half of the story. We also have to deal with the state dependencies existing between the OLCs of different objects types (cf. Example 1). In COREPRO, an OLC dependency is expressed by *external state transitions* between concurrently enacted OLCs. Like an internal state transition within an OLC, an external state transition can be associated with the enactment of a process; i.e., we do not only support event-based synchronizations, but also allow for the enactment of a transition process (e.g., `send email` or `install`). Regarding the two OLCs modeled in Step 2 from Fig. 1b, for example, we associate the external state transition between state `S3` (of the OLC for type `System`) and state `S5` (of the OLC for type `Subsystem`) with process `P`. To benefit from the strong linkage between object relations and OLC dependencies, external state transitions are mapped to object relation types. Regarding our example, the aforementioned external state transition is linked to the `hasSubSys` relation connecting the two object types `System` and `Subsystem`. This information can later be utilized for automatically creating process structures out of given data structures.

The OLCs of all object types and the external state transitions of all relation types form the *Life Cycle Coordination Model* (LCM) (cf. Fig. 1b). The LCM describes the dynamic aspects of the data model and constitutes the scheme for creating data-driven process structures. This is a unique characteristic of COREPRO allowing for the data-driven configuration of process structures. In particular, different process structures (cf. Fig. 1d and Fig. 1f) can be automatically created by instantiating respective data structures (cf. Example 2).

2.3 Generating Data-Driven Process Structures

Picking up the scenario from Example 2, COREPRO allows to instantiate different data structures (cf. Fig. 1c and 1e) and to automatically create related process structures (cf. Fig. 1d and 1f). A data-driven process structure includes an OLC instance for every object from the data structure. This can be seen in Fig. 1e (data structure) and Fig. 1f (related process structure) where the numbers of objects and OLC instances correspond to each other. Likewise, as specified in the LCM, for each relation in the data structure external state transitions are inserted into the process structure; e.g., for every `hasSubsystem` relation in the data structure from Fig. 1e, associated external state transitions (with process `P`) are inserted. As result we obtain an instantiated executable process structure describing the dynamic aspects of the given data structure (cf. Fig. 1d and 1f).

To ensure a correct dynamic behavior, created process structures must be *sound*. A process structure is considered as being sound iff it always terminates properly [3]. Termination of the process structure will be guaranteed if every OLC is sound and there are no cycles caused by external state transitions. An OLC, in turn, is sound (1) if every state can be activated by firing a sequence of state transitions beginning with the start state of the OLC and (2) if the end state of the OLC can be activated by firing a sequence of state transitions beginning from every state within the OLC; further, no state transition must be processing when reaching the OLC end state (i.e., no process is running). It is important to mention that COREPRO allows for checking soundness on model level, i.e., we can ensure that each process structure derived from a sound LCM is sound as well [2]. Thereby, efforts for soundness checks do not rise with the size of the instantiated process structure but only depend on the size of the LCM.

3 Dynamic Behavior of Data-Driven Process Structures

To correctly enact data-driven process structures, a precise and formal operational semantics is needed. This is also fundamental with respect to the analysis of the correctness and consistency of process structures when dynamically changing them (cf. Sect. 4). Therefore, COREPRO uses different *markings* to reflect the runtime status of an enacted process structure. We annotate both, states and (internal as well as external) state transitions with respective markings each of them representing their current runtime status. Fig. 2 shows an example where markings describe a possible runtime status of the process structure depicted

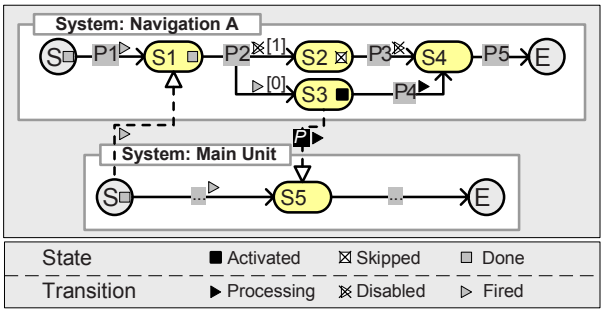

Fig. 2. Process Structure from Fig. 1d with Markings During Runtime

in Fig. 1d. By analyzing the *state markings* from Fig. 2, we can immediately figure out whether a particular state of a process structure has been already passed (e.g., state S1), is currently activated (e.g., state S3), has been skipped (e.g., state S2), or has not been reached yet (e.g., state S4). *Transition markings*, in turn, indicate whether the associated process has been started, skipped or completed. As we will see later, the use of markings eases consistency checks and status adaptations in the context of dynamic changes of process structures significantly. Therefore, markings of passed regions are preserved during runtime.

Altogether, the runtime status of a process structure is determined by the current markings of its constituent OLCs and (external) state transitions. We first introduce the operational semantics of single OLCs and then the one of the external state transitions needed to synchronize the different OLCs.

3.1 Operational Semantics of Single OLCs

Each state of an (instantiated) OLC has one of the markings `NotActivated`, `Activated`, `Skipped`, or `Done`. Fig. 3a shows the conditions for setting these markings. Initial marking of a state is `NotActivated`. An OLC state becomes activated after one incoming internal state transition has fired and all external state transitions are either fired or disabled. To realize this, marking `NotActivated` is subdivided into `IntActivated` and `ExtActivated` (cf. Fig. 3a). A state will become `Activated` if both submarkings, `IntActivated` and `ExtActivated` are reached. At the same time, the preceding state within the OLC is set to `Done`. This excludes concurrently activated states within a single OLC.

The dynamic behavior of OLCs is governed by internal state transitions. An internal state transition enters marking `Processing` and its associated process is started, when the marking of its source state switches to `Activated` (cf. Fig. 3b). When the associated process completes, the marking of the internal state transition either turns to `Fired` or `Disabled` depending on the result of the process (cf. Sect. 2.1). At the same time, the target state of the respective transition enters submarking `IntActivated` (cf. Fig. 3a).

When a transition is disabled, deadpath elimination takes place. Its target state is skipped, and succeeding internal state transitions are disabled as well.

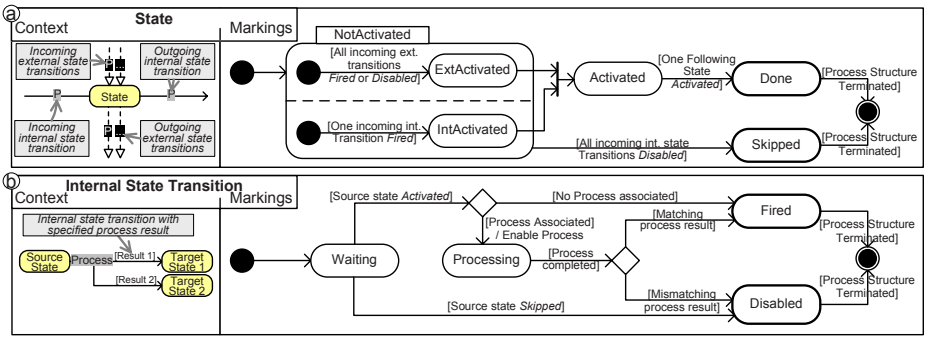

Fig. 3. Behavior of Object States and Internal State Transitions in OLCs

This is continued until the end of the dead path is reached; i.e., until a state is reached which has at least one incoming internal state transition not marked as `Disabled` (cf. Fig. 3b). Deadpath elimination contributes to avoid deadlocks caused by external state transitions which are waiting for activation (cf. Fig. 2, external state transition with process `P`). The described OLC semantics has to be extended in conjunction with loops. Due to lack of space, however, we cannot treat loop backs in this paper.

3.2 Synchronization of OLCs

As mentioned, concurrent processing of objects is a fundamental requirement in engineering processes. At the level of a single OLC, concurrency is encapsulated within the transition processes (e.g., implemented in a workflow management or a product data management system). The proper synchronization of concurrently enacted processes within different OLCs is based on external state transitions. According to internal state transitions, external state transitions are marked as `Processing` (i.e., their processes are started) when their source state becomes `Activated` (cf. Fig. 3 and Fig. 4). When the associated process is completed, in turn, the external state transition is marked as `Fired`. The target state of the fired transition can only become `Activated` if one of its incoming internal transitions has already been marked as `Fired` and all external state transitions are marked either as `Fired` or `Disabled` (cf. Fig. 3a and Fig. 4).

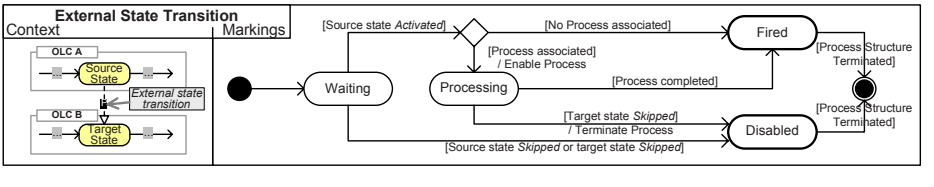

Fig. 4. Dynamic Behavior of External State Transitions Between OLCs

External state transitions whose source state is `Skipped` become `Disabled`. Otherwise the external state transition would remain marked as `Waiting` which could lead to a deadlock within its target OLC. Note that a disabled external state transition does not cause any state change within its source or target state.

An external state transitions whose target state becomes `Skipped`, has to be `Disabled` as well. First, the external state transition does not influence the target state because it has already been `Skipped`. Second, we have to ensure proper termination of the process structure. It terminates after all OLCs have reached their end states (cf. Section 2.3). In particular, soundness necessitates that no constituent process is running (i.e., no transition is processing) then (cf. Sect. 2.3). Therefore, we have to avoid external state transitions whose processes might still be enacted while their target OLC have already reached the end state.

4 Changing Data-Driven Process Structures

So far, we have investigated the modeling, instantiation and enactment of data-driven process structures. To cope with the flexibility requirements of engineering processes, we further have to look at dynamic process structure changes.

Example 3. *The creation, testing and release of the electrical system of a car takes up to several weeks. Meanwhile, changes of the electrical system, such as the addition of new objects to the electrical system, occur often (e.g., adding a new TV Tuner subsystem to the data structure from Fig. 1c). Consequently, the related process structure needs to be adapted (cf. Fig. 1d); i.e., we have to add the processes (e.g., testing) for the new component to the process structure and must synchronize them with the processes of the already existing components.*

Directly changing the process structure would not be a good solution due its size and complexity. Instead, users (e.g., engineers) must be able to perform the changes at a high level of abstraction. In COREPRO this is accomplished by adapting the respective data structure (e.g., describing the electrical system) and by translating these changes into corresponding adaptations of the process structure. Again we utilize the strong relationship between data and process structure to realize this. Furthermore, dynamic changes must not violate soundness of the process structure. To ensure this, we have to constrain changes to certain runtime states (i.e., markings) of the process structure.

First, we provide basic operations for static changes of data as well as process structures (i.e., we do not take runtime information into account). That includes mechanisms for transforming basic change operations applied to a data structure into corresponding adaptations of the process structure. Second, we define marking-based constraints for these change operations in order to ensure soundness of the modified process structure afterwards.

4.1 Static Changes of Data-Driven Process Structures

Basic operations enable the change of existing data and process structures. Concerning data structure changes, COREPRO allows to insert objects and object

relations into a given data structure as well as to remove them (for an overview see the left-hand side of Fig. 5). The offered set of operations is complete; i.e., it allows the engineer to transfer a given data structure into an arbitrary other data structure, which constitutes an instance of the underlying data model (cf. Section 2.1). Regarding adaptations of a process structure, COREPRO provides basic operations for inserting and deleting OLC instances as well as OLC instance dependencies (for an overview see the right-hand side of Fig. 5). In particular, we focus on the coordination and synchronization of different OLC instances and do not consider (dynamic) changes of the internal behavior of a single OLC instance. In this paper, we describe only basic operations for adapting data and process structures and omit a discussion on high-level change operations.

As motivated, data structure changes should be automatically transformed into corresponding process structure changes in order to adapt the overall engineering process to the new (product) data structure. COREPRO accomplishes this based on the information specified at the model level; i.e., we use the life cycle coordination model (LCM) which defines the OLC for each object type and the OLC dependency for each object relation (cf. Section 2.2 and Fig. 1b).

First, we consider the addition of a new object `x` (with type `ot`) to a given data structure. To realize this change, the `addObject` operation (cf. Fig. 5a) can be used. This operation, in turn, is translated into an `addOLC` operation (cf. Fig. 5a) which inserts a corresponding OLC instance to the process structure. The OLC instance to be added is derived from the OLC linked to object type `ot` within the life cycle coordination model (LCM). When adding a new object `x` to a data structure, its relations to other objects can be specified using the `addRelation` operation (cf. Fig. 5b). The newly added relations are then automatically translated into OLC dependencies of the corresponding process structure. This is realized based on the `addOLCDependency` operation, which inserts corresponding external state transitions between the involved OLC instances. Again, information from the life cycle coordination model is used to correctly transform the newly added object relation to a corresponding OLC dependency.

To remove an object relation `r` from a data structure, we provide operation `removeRelation`. It is mapped to operation `removeOLCDependency` for the process structure, which removes all external state transitions associated with the relation `r` (cf. Fig. 5c). Removal of an isolated object (i.e., relations to other objects have been already removed) from a data structure is mapped to the removal of the associated OLC instance from the process structure (cf. Fig. 5d).

4.2 Dynamic Changes of Data-Driven Process Structures

As motivated, data-driven process structures have to be dynamically adapted when the related (product) data structure is changed. This section considers dynamic process structure changes and deals with relevant issues.

A key challenge is to preserve soundness of the overall process structure when dynamically adapting it. As long as we only consider static changes (cf. Section 4.1) soundness of an automatically changed process structure can be ensured if the underlying LCM (e.g., Fig. 1b), from which the process structure is derived,

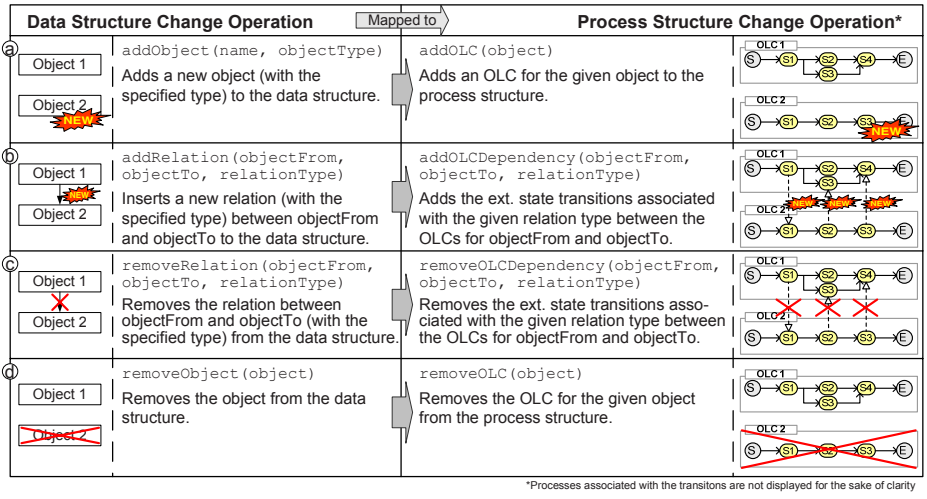

Fig. 5. Data Structure Changes and Related Process Structure Adaptations

is sound. The COREPRO modeling tool always checks this before an LCM can be released. When adapting data and process structures during runtime, we have to define additional constraints with respect to the status of the process structure in order to guarantee soundness afterwards. In particular, the addition or removal of external state transitions must not result in deadlocks or livelocks. Note that this problem is related to correctness issues discussed in the context of dynamic and adaptive workflows [4,5]. Here, one has to decide whether a structural change can be applied to a running workflow instance or not. One correctness notion used in this context is *compliance* [5]. Simply speaking, a workflow instance is compliant with a structurally modified workflow schema if its current execution history is producible on the new workflow schema as well.

In principle, the compliance criterion could be applied to process structures as well. COREPRO logs the events related to the start and completion of the processes coordinated by the process structure in an execution history. However, the compliance criterion is too restrictive for data-driven process structures. Consider, for example, the dynamic removal of an object from a data structure; e.g., an already designed component might have to be removed from the electrical system when a problem is encountered during testing. Respective runtime changes often become necessary in practice and must be supported, even if the OLC instance related to the data object has been already started or completed (i.e., corresponding entries were created in the execution log of the process structure). For this case, compliance would be violated.

COREPRO uses specific correctness constraints to adequately deal with dynamic changes. We utilize the trace-oriented markings of OLC states and the semantics of the described change operations when defining these constraints.

First, COREPRO allows to delete an already started or finished OLC instance `olc`, together with its external state transitions, as long as the execution of other OLC instances has not yet been affected by `olc`. This constraint will be satisfied if the target states of all outgoing external state transitions of `olc` have not yet been activated. In particular, this ensures that the removal of `olc` will not have any effect on other OLC instances, and therefore does also not influence soundness of the overall process structure. Note that this exactly reflects reality; i.e., as long as the removal of a data object has no effects on the status of other data objects (i.e., on the states of their related OLC instance) the object and its corresponding OLC (with its external transitions) can be removed.

Second, we constrain the addition of new OLC instances and their dependencies to other OLC instances. In particular, it must not be allowed to add an external state transition with the new OLC as source if the target state (of another OLC instance) is already marked as `ACTIVATED` or `DONE`. Otherwise soundness of the overall process structure would be violated. Apart from this, the constraint makes sense from a practical perspective; e.g., it must not be possible to insert untested components into a (product) data structure and relate them to electrical systems which have already been released.

Process structures can become very large. Therefore one goal is to efficiently check whether a dynamic change can be applied to a given process structure. It is important to mention that the operations presented in Sect. 4.1 only modify the process structure itself, but do not affect individual OLC instances (i.e., OLC states and internal state transitions are not changed by these operations). Furthermore, when analyzing the change scenarios sketched above, we can see that the applicability of a basic runtime change mainly depends on the markings of the target states of the external state transitions to be added or deleted (either explicitly or implicitly due to the deletion or addition of an OLC instance). Thus, we can reduce efforts for constraint checking to the manipulated external state transitions. More precisely, we consider the *change regions* defined by each of these transitions. Such a region comprises the transition itself as well as its source and target state. In principle, two fundamental issues emerge when dynamically adding or deleting an external state transition:

1. May an external state transition be added or deleted when considering the current marking of the states of its change region?
2. Which marking adaptations become necessary with respect to a change region when adding or removing the respective external state transition?

As example consider Fig. 6a which shows two concurrently executed OLC instances `OLC A` and `OLC B`. Assume that an external state transition (with associated process `P`) shall be dynamically added with `S1` as source and `S2` as target state. First, COREPRO checks whether this change is possible considering the current markings of the change region. Since the target state `S2` of the respective external state transition has not yet been activated, the change is allowed (cf. Fig. 6b). When applying it, in addition, the markings of the change region have to be adapted to correctly proceed with the flow of control. In the given example the newly added external state transition is automatically evaluated,

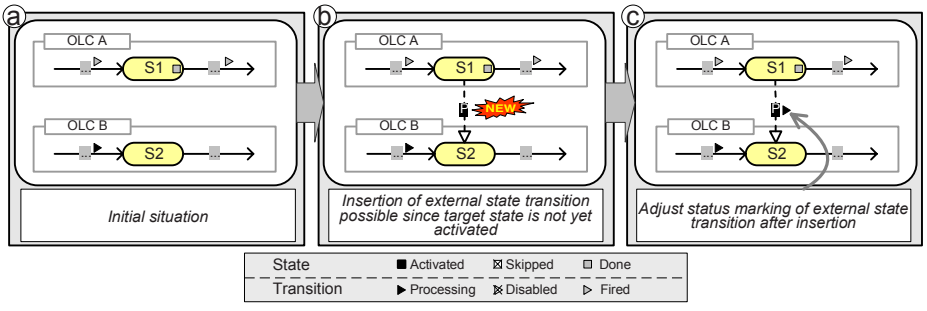

Fig. 6. Dynamic Addition of an External State Transition

which changes its marking from `WAITING` to `PROCESSING`; i.e., the process related with the transition is instantiated and started (cf. Fig. 6c). Note that a deadlock would occur in the given scenario if the newly added transition had not been marked as `PROCESSING`. As another example consider again Fig. 6a. If we tried to add an external state transition with source state `S2` and target state `S1` the change would be not allowed. Otherwise both `OLC A` and `OLC B` might be completed before the process associated with the new external state transition completes; i.e., soundness of the resulting process structure would be violated.

Generally, a structure change comprises multiple operations of which either all or none of them have to be applied to the data and process structure. To enable change atomicity and isolation, COREPRO allows to group change operations within a change transaction. Removing an object, for example, might require the removal of several relations associated with this object and finally the removal of the object itself. Temporary inconsistencies might occur, but will not become visible to other change transactions and users respectively.

4.3 Practical Impact

In the automotive domain, electrical systems comprise up to 300 components. Related process structures have more than 1300 processes and 1500 process dependencies [2]. We have already shown the need for dynamically adapting process structures in this scenario. In practice, often more than 50 dynamic changes of a process structure become necessary during its enactment (cf. Example 3). A software error within a component (e.g., the `Main Unit` of the navigation system), for example, requires the exchange of this component; i.e., a new version of the component has to be constructed (cf. Fig. 7). When exchanging the component, the related OLC has to be exchanged as well to ensure that the associated processes can be correctly re-enacted. Whether the adaptation is possible or not depends on the status of the process structure; e.g., if the `Main Unit` subsystem has been already installed and the `Testdrive` process has already started, the `Main Unit` subsystem must not be exchanged (cf. Fig. 7). COREPRO enables engineers to adapt respective process structures by changing the (product) data structure and by transforming these changes to the process structure.

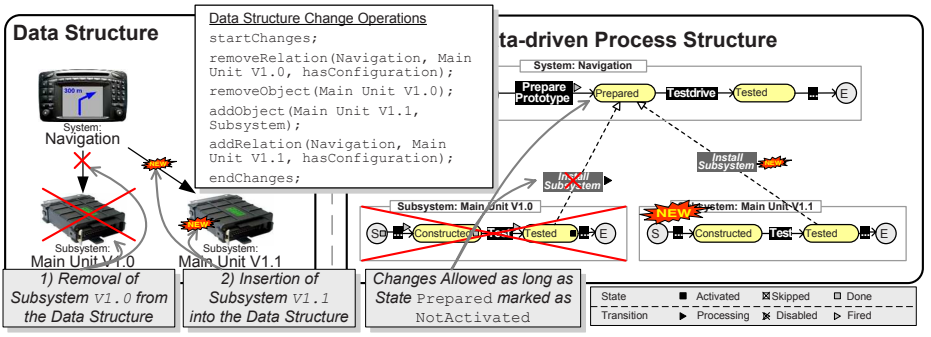

Fig. 7. Release Management Process: Creation of a System Release

5 Related Work

At first glance, conventional approaches support modeling and coordination of process structures. Choreography definition languages, for example, allow for the activity-centered specification of the dependencies between (distributed) processes [6]. Changes of process choreographies are discussed in [7,8]. The data-driven derivation and change of process structures, however, is not considered.

An approach for enacting and adapting data-driven process structures is presented in [9]. Focus is on simple data structures (e.g., lists, sets). COREPRO, by contrast, enables the definition and change of arbitrary complex data structures as well as automated creation of related process structures. Approaches for deriving process structures from bills of material are described in [10,11]. While [10] focuses on product-driven (re-)design of process structures based on design criteria like time or cost, [11] discusses how to coordinate activities based on object relations. The latter approach constitutes the basis of the *Case Handling* paradigm [12]. The idea is to model a process (structure) by relating activities to the data flow. The concrete activity execution order at runtime then depends on the availability of data. Further approaches providing support for modeling process structures consisting of OLCs are presented in [13,14]. For example, they enable the generation of activity diagrams from OLCs and vice versa. The generic definition of data-driven process structures as well as their adaptation, however, is not covered by the aforementioned approaches.

There exist approaches from the database area for describing object-oriented data structures and their behavior [15,16]. They provide a rich set of elements for modeling data structures and for mapping them to OLCs, but neglect enactment and dynamic changes. COREPRO does not focus on the data modeling part for the following reason. In the engineering domain, data modeling (incl. the expression of constraints and the avoidance of data inconsistencies) is usually done within a product data management system, which further provides techniques for versioning and variant handling. Our data model constitutes a simplified view on the PDM data model capturing the needs for coordinating and dynamically adapting process structures.

6 Summary and Outlook

IT support for enactment and consistent change of data-driven process structures is a major step towards the use of process management technology in engineering domains. COREPRO provides a new approach with respect to the automated creation and data-driven adaptation of process structures during runtime. In particular, our approach reduces modeling efforts for large process structures and ensures correct coordination of processes. Further, COREPRO enables adaptations of process structures at both, build- and runtime at a high level of abstraction while it disallows dynamic changes of process structures which would lead to an inconsistent runtime situation. Our case studies have shown that in real world scenarios it might be necessary to apply changes even if they lead to inconsistencies. Dealing with such situations requires extensive exception handling techniques (e.g., backward jumps within OLCs) which are addressed by COREPRO and will be presented in future publications.

References

1. Müller, D., Herbst, J., Hammori, M., Reichert, M.: IT Support for Release Management Processes in the Automotive Industry. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 368–377. Springer, Heidelberg (2006)
2. Müller, D., Reichert, M., Herbst, J.: Data-driven modeling and coordination of large process structures. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 131–147. Springer, Heidelberg (2007)
3. Aalst, W.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
4. Rinderle, S., Reichert, M., Dadam, P.: Flexible support of team processes by adaptive workflow systems. Distributed & Parallel Databases 16(1), 91–116 (2004)
5. Rinderle, S., Reichert, M., Dadam, P.: Correctness Criteria For Dynamic Changes in Workflow Systems: A Survey. DKE 50(1), 9–34 (2004)
6. W3C: WS-CDL 1.0 (2005)
7. Rinderle, S., Wombacher, A., Reichert, M.: Evolution of process choreographies in DYCHOR. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 273–290. Springer, Heidelberg (2006)
8. Aalst, W., Basten, T.: Inheritance of workflows: an approach to tackling problems related to change. Theoretical Computer Science 270(1-2), 125–203 (2002)
9. Rinderle, S., Reichert, M.: Data–Driven Process Control and Exception Handling in Process Management Systems. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 273–287. Springer, Heidelberg (2006)
10. Reijers, H., Limam, S., Aalst, W.: Product-based workflow design. MIS 20(1), 229–262 (2003)
11. Aalst, W.: On the automatic generation of workflow processes based on product structures. Comput. Ind. 39(2), 97–111 (1999)
12. Aalst, W., Berens, P.J.S.: Beyond workflow management: Product-driven case handling. In: GROUP, pp. 42–51 (2001)

13. Liu, R., Bhattacharya, K., Wu, F.Y.: Modeling Business Contexture and Behavior Using Business Artifacts. In: Krogstie, J., Opdahl, A., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 324–339. Springer, Heidelberg (2007)
14. Küster, J.M., Ryndina, K., Gall, H.: Generation of Business Process Models for Object Life Cycle Compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 165–181. Springer, Heidelberg (2007)
15. Kappel, G., Schrefl, M.: Object/behavior diagrams. In: ICDE, pp. 530–539 (1991)
16. Dori, D.: Object-process methodology as a business-process modelling tool. In: ECIS (2000)