Why We Need Static Analyses of Service Compositions

Fault vs. Error Analysis of Soundness

Thomas M. Prinz and Wolfram Amme
Course Evaluation Service and Chair of Software Technology
Friedrich Schiller University Jena
Jena, Germany
e-mail: {Thomas.Prinz, Wolfram.Amme}@uni-jena.de

Abstract—**The programming of classic software systems is well-supported by integrated development environments. They are able to give immediate information about syntax and some logic failures. Although service compositions are widely used within modern systems, such a support for building service compositions is expandable. In this paper, we plead for the creation of an integrated development environment for service compositions, which enables immediate failure feedback during the development. To this end, there is a need for new research activities on occurring failures and how they can be found. Since most current failure finding techniques are based on dynamic approaches, e. g., state space exploration, we show in a case study on soundness that the application of dynamic techniques is not a suitable solution for integrated development environments. In most cases, they are either too time consuming or their output does not lead easily to the root of a failure. As a result, we suggest new advanced (static) analyses of service compositions. To accentuate that pleading, the paper demonstrates a static analysis tool, *Mojo*, which can be used to check soundness and to get detailed fault diagnostics. With the help of this tool, it was possible to compare the behaviour of dynamic and static analysis techniques in a practical context. For this, a benchmark of real world service compositions was checked regarding soundness with a state space-based (dynamic) and a compiler-based (static) tool. Altogether, the case study and the comparison in a practical context show that dynamic analyses are not suitable for development support. Static analyses should be used instead.**

Keywords–Service Composition; Analysis; Case Study; Mojo; Soundness.

I. INTRODUCTION

The development of *service compositions* (aka *work-flows*) is an error-prone task just like the development of software systems. For example, only approx. 46 % of compositions of a real world benchmark have a comprehensible behaviour, as will be shown later in this paper. Whereas integrated development environments (IDEs) exist for the development of software systems, the tool support for the development of service composition is expandable at this time. That is surprising since there is a substantial common ground between both: There is data information passed through variables and there is a flow graph, which represents the structure. However, most tools for service compositions cover only their modelling and execution. They do *not* support the creation of *correct* compositions.

As an example, Figure 1 shows a service composition (taken from the conference version of this paper [1]). The composition handles the logic during the execution of a survey and follows the notation and semantics of the *Business Process Model and Notation* (BPMN) [2]. Typically, the execution begins at the *start node*. Then, the execution reaches a *task/service node*, which loads the survey at first. After the task node, the execution reaches a *fork node*. A fork node produces parallelism so that all outgoing edges are followed by a *control flow*.

One control flow of the fork node follows the lower edge to a further service, which handles the inputs of the survey. Subsequently, it reaches a *join node*, which synchronizes parallel control flows. Since another control flow has not reached the other incoming edge of the join node yet, the current flow has to wait.

The other control flow produced by the fork follows the upper outgoing edge. It loads the current page and reaches a *merge node*. A merge node combines sequential control flows. After the merge is executed, the flow arrives at a *split node*. The split node decides, which outgoing edge the flow follows. Either the flow goes to the upper edge and loads the next page, or it executes the task node at the right hand side and loads the previous page of the survey. If the previous page is loaded, the flow reaches another merge node, which guides the flow to a node visited previously. Therefore, the composition contains a cycle.

If the split node decides to load the next page, then the flow arrives a further split node. It decides whether the survey is finished or not. If the survey is finished, the control flow reaches the join node like the other flow before. Now, the join node can be executed and a conclusion will be shown. Eventually, the survey is finished when it reaches the *end node*.

As mentioned, we expected a wide development support during the creation of the composition of Figure 1 since the research on service-oriented architectures has passed its 20th anniversary. However, it is hard to find tools that give immediate development support. For this missing support, there are two possibilities, which exclude each other: Either there is some research, which seriously supports the development, but it is not used in the tools. Or, there is no such research and, therefore, the tools need qualified research results for that support.

We found some research approaches to *verify* service

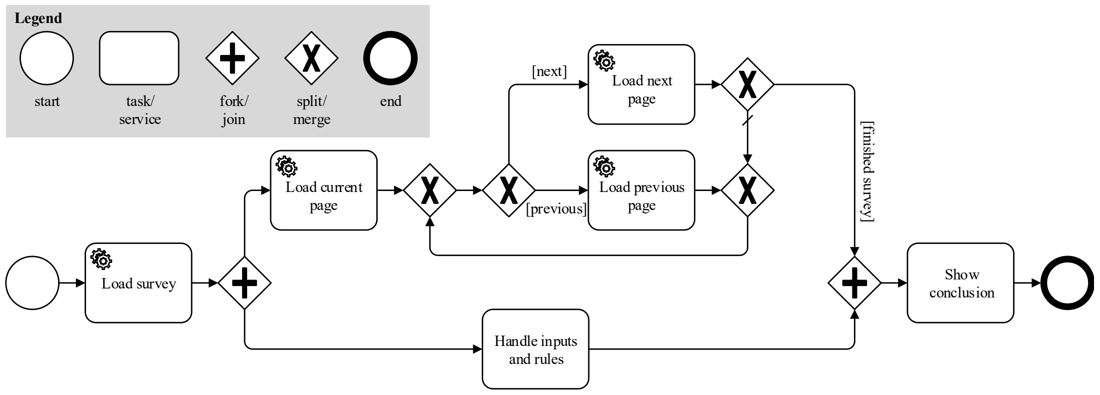


Figure 1. A service composition, which handles the logic of the execution of a survey.

compositions in form of business processes in the literature. A large number of those approaches concentrate on the verification of the *soundness* property [3] requiring that a service composition cannot run into *deadlocks* or in undesired double executions of services (*lacks of synchronization*). Since the soundness property is defined on the runtime behaviour of the composition, most algorithms search for undesired behaviour in a simulation. That means, they regard the *state space* of the composition, whereas the state space defines all possible reachable states.

State space-based algorithms perform *dynamic* analyses of service compositions since each found error can actually appear at runtime. However, as each error is a malformed reachable state, which has a *cause* within the service composition, finding exactly that cause given a specific error is a hard task.

Imagine, a developer extends the composition of Figure 1 to the one of Figure 2, i. e., the developer adds a new service, which delivers additional information (e. g., when the survey was started). The developer does it in a composition tool with a state space-based algorithm. After performing the algorithm, an error message is displayed and the tool visualizes the malformed state — a deadlock — directly into the composition (cf. Figure 2, illustrated by the black dots, *tokens*, which represents the control flows). As the tool does provide the *error* only and *not its cause* (the *fault*), the developer has to search the cause of the deadlock. However, there is not a classic cause of that deadlock since it happens because of a possible previous lack of synchronization, i. e., the double execution of the same nodes (illustrated by grey dots in the figure). If the composition contains additional nodes and becomes complexer, it be harder to detect the cause of that error in the composition. It seems that dynamic approaches are too imprecise and time consuming to support the development of service compositions seriously.

In this paper, we will demonstrate that problem in a case study on soundness with dynamic analyses relating to classical software testing terms. Furthermore, we will accentuate that problem by a direct comparison of dynamic and static analysis techniques in the field of user support. As a case example for a dynamic analysis technique, state space exploration is used throughout the paper. For static analyses, our own proposals for checking soundness are used. They search for faults of deadlocks and lacks of synchronization instead of errors [4][5][6][7]. We refer to our techniques as *fault finding*. The comparison is done in a practical application of both techniques in form of the tools *LoLA* (state space exploration) and our analysis tool *Mojo* (fault finding).

As the result of this paper, it can be shown why it is better to use static analysis techniques for tool support during development instead of dynamic analyses. As a consequence, we plead for more research of static analyses for service composition.

This paper is structured as follows: At first, it introduces the field of verifying service compositions by taking a look on the state of the art (see Section II). Afterwards, a more formal and language independent model for service compositions will be explained — the *workflow graphs* (see Section III). Subsequently, in Section IV, it shows within a case study on soundness that dynamic analysis approaches are not suitable to give profitable tool support. Based on this case study, further practical considerations are done in Section V by a direct comparison of a dynamic and a static approach. Eventually, this paper closes with a summary in Section VI, which will support the notion that it is important to use static analyses instead of dynamic ones during composition development. Furthermore, it shows possible future work.

II. State of the Art

The *soundness* analysis of service compositions, especially in the case of *workflows* and *processes*, has a long tradition. It appears firstly in the work of van der Aalst [3] in the year 1995. To this day, several different notions of soundness were introduced. The interested reader can find them in Puhlmann [8] and van der Aalst et al. [9].

In this paper, we have chosen the classic notion of soundness. There are several approaches, which try to classify whether a workflow is sound or not. The first known algorithm was introduced by van der Aalst [3]. It is based on the rank theorem [10], which can be solved in cubic time complexity regarding the size of the workflow graph

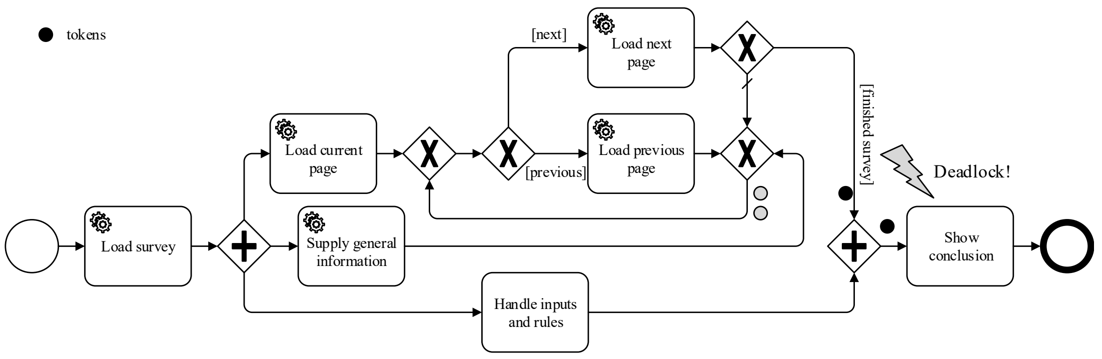


Figure 2. A malformed service composition of the service of Figure 1.

[11]. However, this approach does not give any diagnostic information, *where* or *why* the workflow graph is unsound [12]. For this reason, other approaches were developed, which we classify into three main approaches: (1) Model checking, (2) graph decomposition, and (3) pattern and compiler-based approaches.

A. Model Checking

The dominating approach in workflow verification is *state space exploration* [13]. In state space exploration, each possible execution step of a workflow will be considered (the state space). If during the consideration there is an execution step, which contains an error, the analysis will be stopped and the erroneous state inclusive a so-called *trace* will be returned. The analysis will be stopped since some (even small) workflows have a very large (sometimes arbitrary large) state space and a complete examination of the state space would take too much time or will not end. This fact is called the state space explosion problem [14]. To counteract this problem, arbitrary growing states are replaced with a neutral element. The resulting state space is called *Coverability Tree* [15], but its size is still exponential large with regard to the entire workflow.

Therefore, Lohmann and Fahland refined the state space approach by performing some graph reductions [16]. They also tried to explain why errors happen in processes, however, there work never reached a final state and there are still open issues unfortunately.

Finished graph reduction techniques, which consider the state space of processes too, are used by the tools Woflan [17] and LoLA [18]. The *Low Level Petri Net Analyzer* (LoLA) serves as general model checking tool for Petri nets. It performs graph reduction techniques and state space exploration. The verification property to prove (e. g., soundness) must be given as formal equation. Then, LoLA checks whether each state in the state space fits to the propery.

Woflan seems to be the most complete analysis tool for workflows and is based on workflow nets and state space exploration. Besides soundness it checks other quality criteria. This is done by reducing the entire workflow iteratively. If the result of this reduction is trivial, the workflow is sound. Otherwise, Woflan decides with the *S-coverability* [10] whether it can give diagnostic information or not. If the workflow is not S-coverable, it is unsound, however, it is unknown whether there is a deadlock or a lack of synchronization. If the workflow is S-coverable, a state space exploration has to check the soundness property. That results in an exponential runtime of Woflan.

Besides state space exploration there are other model checking approaches to check soundness for workflows. Sadiq and Orlowska have introduced the *instance graphs*, which are subgraphs and represent possible execution traces [19]. Eshuis and Kumar use this approach to find erroneous instance graphs with integer programming [20]. That gives enough diagnostic information to repair a workflow. However, the workflows have to be acyclic and the integer program has an exponential worst-case runtime.

B. Graph Decomposition

Since model checking techniques have their limits in performance and failure diagnostic, other approaches were considered. A prominent approach is the decomposition of the workflow into smaller subgraphs. Chrząstowski-Wachtel et al. use this approach and offer a new concept of representing workflows at the same time: The representation as a tree [21]. They propose to construct a workflow starting by the root and adding child nodes iteratively. Then, the workflow is sound by construction. However, such a structured construction of workflows is not used in practice since most workflows should represent unstructured service compositions or real-world business processes.

The derivation of a tree structure starting by an unstructured workflow was done by Vanhatalo et al. [22][23]. They split the entire workflow into fragments (subgraphs) with one ingoing and one outgoing edge — a *Single-Entry-Single-Exit* (SESE) fragment. Since this splitting into SESE fragments results in a hierarchy of fragments, they can be visualized as a tree: The *Process Structure Tree* (PST). Each fragment can be analysed separately by replacing subfragments with a single edge. Simple fragments are analysed by performing some rules and heuristics. Complex fragments cannot be handled, however, alternative soundness approaches can be applied. That approach reduces the size of the graphs to consider, allows the finding of one error per fragment, and has a linear asymptotic runtime [24]. But it is incomplete regarding soundness.

C. Pattern and Compiler-based Approaches

Actually, SESE decomposition is a compiler-based approach since it considers the workflow without simulation and it was already applied for compilers by Johnson et al. [25][26]. Besides typical compiler approaches, there is a growing number of approaches considering *patterns* in the last years.

Dongens et al. began with pattern approaches by defining two relations called *causal foot prints* [27]. On the base of this foot prints, they find three (anti) patterns for deadlocks and lacks of synchronization. However, it is not sure that an anti pattern means that the workflow is really unsound.

Another pattern-based approach was introduced by Favre and Völzer [12]. They define two relations for deadlocks and lacks of synchronization too. With a kind of dataflow analysis, the information needed by the relations are propagated through the workflow. The approach results in good diagnostic information and has a polynomial asymptotic runtime. However, the pattern approach only works for acyclic workflows.

Based on anti-patterns, Favre et al. proposed another approach [28][29]. The used anti-patterns are similar to those of Dongens et al. and can be applied to workflows with cycles. If a workflow contains such an anti-pattern, then the workflow is unsound. In the case of incorrectness, additional analyses are started to supply diagnostic information. The diagnostic information are very good, but the runtime behaviour is quintic and it is only possible to detect one error at once.

We proposed a compiler-based approach in [4][5][6][7]. Instead of considering the errors of deadlocks and lacks of synchronization, we investigated their faults by starting our analyses from different entry points of the workflow — a partial analysis. This makes it possible to detect *potential* errors behind others. Based on this partial analysis, we introduced two new techniques: One for detecting faults of deadlocks and a second for detecting faults of lacks of synchronization.

For deadlocks, we observed that in sound workflows, a node never jams obviously. It never jams since its execution is guaranteed every moment, the execution reaches it. We figured out that a node never has a deadlock when on each path to this node another node guarantees its execution. Otherwise, there is the potential for a deadlock.

Our second technique considers faults of lacks of synchronization. At first, we observed that parallelism must occur before the manifestation of a lack of synchronization obviously. Each of the parallel control flows can meet each other first, where two paths starting from the start of the parallelism meet at first. We call them *meeting points*. If all meeting points are synchronization nodes (join nodes), we cannot run into a lack of synchronization. Otherwise, there is a potential for a lack of synchronization.

Both techniques have in common that they are complete regarding soundness. Furthermore, they find the causes of deadlocks and lacks of synchronization in a bi-quadratic asymptotic runtime (depending on the number of edges in the workflow). As result, exact diagnostic information are provided and it is possible to detect faults behind faults. At this time, we believe that our approach is the best one helping to build sound service compositions.

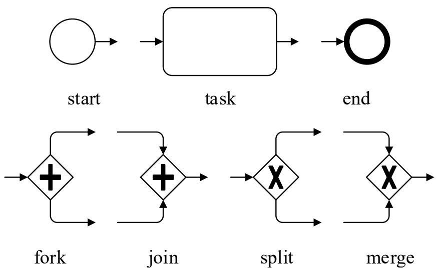


Figure 3. Notations for start, end, task, fork, join, split, and merge nodes.

III. Preliminaries

There are different languages and notations to describe service compositions and business processes. Popular examples of those languages are BPMN [2], the *Business Process Execution Language* (BPEL) [30], and the *Yet Another Workflow Language* (YAWL) [31]. In research, however, the general concepts of those composition languages are simplified to describe service compositions in a language independent way. For this purpose, two notions are used: The *workflow nets* introduced by van der Aalst [3][13] and the *workflow graphs* of Sadiq and Orlowska [19]. Whereas the former uses the notions of Petri nets [32], the latter are similar to control flow graphs of the theory of compiler construction. Since we believe that workflow graphs are easier to understand and to illustrate, we use workflow graphs throughout this paper to represent service compositions.

In general, a workflow graph is a *directed graph* consisting of *nodes* and *edges*. Each of the workflow graph nodes is either a *task*, a *fork*, a *join*, a *split*, a *merge*, the unique *start*, or the unique *end* node; where all nodes of the same type have the same appearance and semantics, i. e., how they are executed. Rules define how the nodes are connected. They depend on the kind of node: The start node has no incoming but exactly one outgoing edge, whereas the end node has exactly one incoming but no outgoing edge. Each task node is reached and leaved by exactly one edge. A split and fork node has exactly one incoming edge and at least two outgoing edges. Merges and joins are reached by at least two incoming edges and can be leaved by one outgoing edge. For the visualization of workflow graphs, we use the same notations as the BPMN standard [2]. For this reason, start and end nodes are visualized as (thick) circles. Tasks are illustrated as simple rounded rectangles. Split and merge nodes are visualized by diamonds with crosses. Eventually, diamonds with pluses are used to illustrate fork and join nodes (cf. Figure 3).

The different visualizations mark the different semantics of the nodes. Usually (e.g., from Vanhatalo et al. [22]), the semantics of the nodes are described as a *token game* known from Petri net semantics [32]. In token games, the numbers of *tokens* on the edges are used to describe a single execution situation (a *state*). In each state there is a number

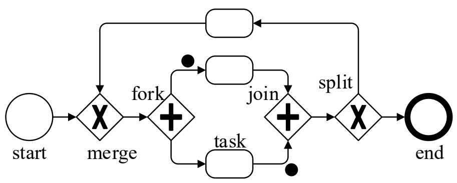


Figure 4. A simple workflow graph within an execution state.

of tokens assigned to each edge. There are two important states of workflow graphs: (1) The initial state and (2) the termination state. Within the initial state, only the single outgoing edge of the start node carries a token, whereas within the termination state only the single incoming edge of the end node carries a token. Throughout this paper, we use black dots on edges to illustrate tokens in a state. Figure 4 shows such a state of a simple workflow graph.

In each state (except the termination state), there should be some nodes, which are *executable*, i. e., their functionality can be performed. After the *execution* of a node, the state changes (a *state transition*). A sequence of state transitions is an execution sequence of the workflow graph and we can say, that the last state of the sequence is *reachable* by each other state of the sequence [22].

As mentioned before, whether a node is executable and what happens after the execution of this node is defined by its type. The start and end node have no special semantics. Therefore, they are only used to mark the start and end of a workflow graph. Each node, except a join node, is executable once there is at least one token on one of its incoming edges. A task node takes a token from its incoming edge and puts it back to its outgoing edge. Split and merge nodes perform non-deterministic choices instead: Split nodes take a token from their incoming edge and put a single token to one of their randomly chosen outgoing edges; whereas merge nodes take one token from one randomly chosen incoming edge (with a token) and put a token to their outgoing edge. Eventually, fork and join nodes handle parallelism. Fork nodes take a token from their incoming edge and put a token on each outgoing edge. However, join nodes are only executable if each of their incoming edges has at least one token. If a join node is executed, a token is removed from each incoming edge and a single new token is placed on its outgoing edge. Figure 5 summarizes the execution semantics.

As explained in the introduction of this paper, we consider the notion *soundness* [3][19] as correctness criterion of service compositions. A workflow graph is called *sound* if neither a *deadlock* nor a *lack of synchronization* is reachable from the initial state. A *deadlock* is a non-termination state, in which no node is executable. A *lack of synchronization* is a state in which at least one edge carries more than one token.

Figure 6 shows a typical and simplified deadlock state. The join node in the figure (the right node with the plus) will never be executed since it needs another token on its lower incoming edge. A typical lack of synchronization is

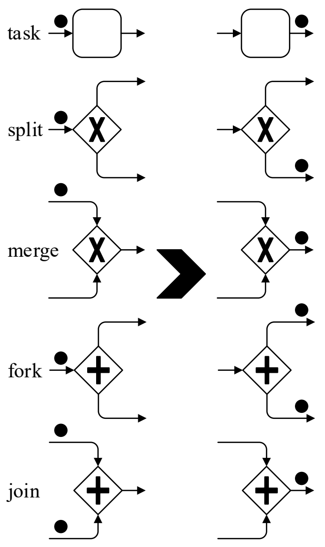


Figure 5. The execution of the different kind of nodes.

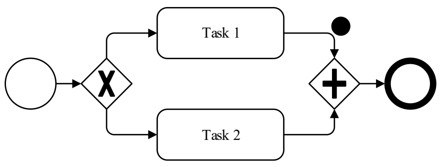


Figure 6. A deadlock.

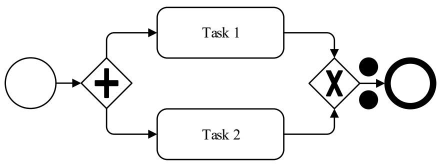


Figure 7. A lack of synchronization.

illustrated in Figure 7. The outgoing edge of the merge node carries two tokens. This is possible since both tokens produced by the left fork node will never by synchronized.

IV. Consideration of Dynamic Analyses for Workflow Development

If an execution of a workflow graph results in a deadlock or lack of synchronization, the graph's behaviour is not well defined and comprehensible. So, it is beneficial to know whether a workflow graph is sound or not. This can be easily answered by the usage of dynamic analysis techniques like state space exploration.

State space exploration dominates the literature in process verification up to the present date. It indicates whether the workflow graph is sound. Furthermore, the developer gets a *failure trace*, or more precisely, a path within the state space from the initial to the erroneous state.

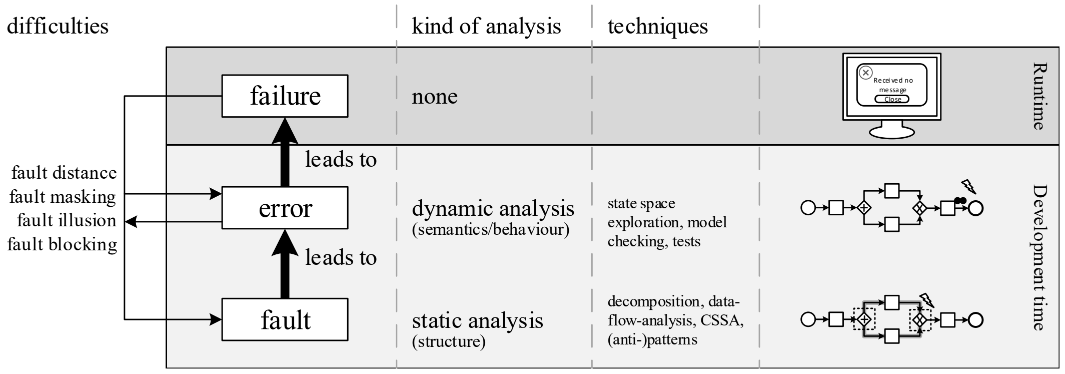


Figure 8. Overview of the connections between failures, errors, and faults.

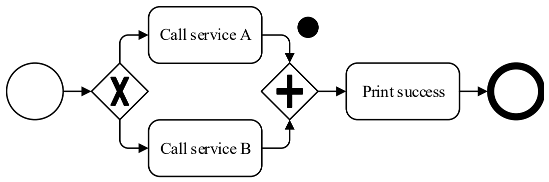


Figure 9. The success message is never printed leading to a failure.

The strength of such dynamic analyses are that new correctness criteria can be defined and checked easily. This makes it possible to use the same dynamic analysis technique for different verification problems and to add new analyses very fast. This is useful to guarantee the delivery of correct software products. However, as mentioned in the introduction, dynamic analysis techniques have their weaknesses during development time since they answering rarely, *why* a workflow graph runs into a deadlock and a lack of synchronization. But a developer needs to know, *why* some wrong behaviours happen to repair and to avoid them.

In the following, we reinforce these weaknesses by showing that dynamic analysis techniques lead to a time expensive and hard troubleshooting when we try to identify the causes of wrong workflow graph behaviours. For this, we consider a case study on dynamic analyses regarding typical *terms of software testing* (taken from [33]). We use this vocabulary since we talk about the development of workflows like software. Furthermore, these terms make it possible to evaluate the located errors and how they can be used for troubleshooting. In addition, the following comparisons motivate the usage of service composition specific static analyses. An overview of the terms and their interdependencies is illustrated in Figure 8.

A. Failures, Errors, and Faults

In software testing, there are different terms with different meanings for wrong execution states of a program. A state is called a *failure* if a user of the program sees an undesired behaviour or result [33]. For example, in the workflow graph in Figure 9, we see that the last task — the printing of the success message — will not be executed since the composition runs into a deadlock in the join node. Therefore, the user is informed by the missing success message that there is a failure.

Such a failure is the manifestation of an incorrect development of the composition. This manifestation is called an *error* [33]. For example, the process developer may know why the user does not see the success message, as the developer may identify that the execution blockades. The reason, *why* the execution blockades, is called a *fault*. A fault is the wrong human action during the development of the service composition [33].

Obviously, to repair an erroneous service composition, a developer has to know the fault instead of errors and failures. If the developer knows only the error or the failure, it has to derive the fault from the diagnostic information.

Considering the previous term definitions, each dynamic analysis technique always results in an error since it searches within the different execution possibilities of a workflow graph instead at the workflow graph itself. As a result, the developer has to derive the real fault after each dynamic analysis, to be able to repair the composition. However, this derivation of the fault is a difficult task since errors can be *masked* or *disguised*. Furthermore, the developer may underestimate the possible *distance* between the error and the fault, thus disregarding an origin early in the composition. All those different difficulties are considered in the following sub sections.

B. Fault Distance

The *distance* between a fault and its error is known as the passed time or passed program instructions until a fault results in an error [34]. The workflow graph in Figure 10 has some bigger subgraphs, which are folded as services D and E for reasons of lack of space. After the subgraph D is executed, the workflow graph will end in a deadlock state as the join on the right-hand side cannot be executed. Naturally, a developer would now search the corresponding fault near the error. Since a lot of time has passed and the workflow graph is complex owing to the subgraphs D and E, it is very difficult to identify the fault. A natural and simple correlation is that the difficulty of finding corresponding faults of errors grows with their distances.

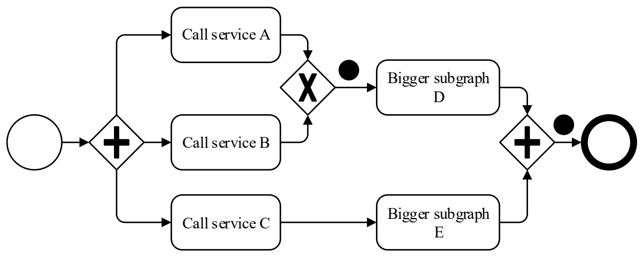


Figure 10. The distance between the fault and its error may be large.

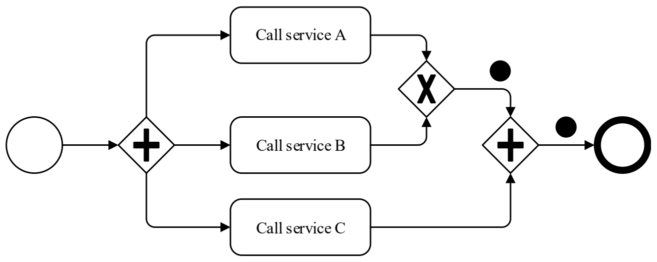


Figure 12. One error produces another error so that there is the illusion of a fault, which does not exists.

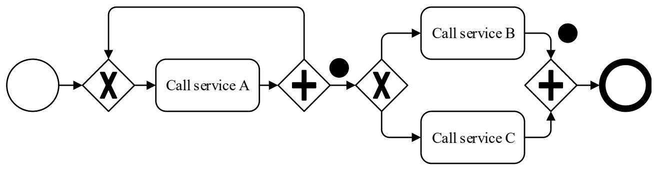


Figure 11. One fault masks another fault so that the failure may disappear.

C. Fault Masking

Fault masking is the situation, in which one fault prevents the detection of another fault [33]. This leads to much difficulty as the faults do not necessarily cause a visible failure. Furthermore, it may happen that one fault is repaired by another one.

An example of fault masking in the context of service compositions is illustrated in Figure 11. The first part of the workflow graph (the loop) results in a lack of synchronization, whereas the second part has an obvious deadlock. However, the first part produces an endless number of tokens so that the previous lack of synchronization always prevents the latter deadlock at runtime. A dynamic approach would now result in a lack of synchronization only — it is not able to detect the deadlock as it does not appear at runtime. To this end, the first fault has to be repaired *before* the deadlock appears within a dynamic approach. This makes the correction of a service composition more time expensive since a necessary analysis has to run for each error at least.

D. Fault Illusion

Fault illusion is not a classic term of software testing. We introduce it at this point, because such a situation is not accurately described by the existing terms. Figure 12 exemplifies this illusion with a workflow graph. Currently, that workflow graph is within a deadlock state since there is no node, which can be executed.

A dynamic analysis technique could provide this deadlock state. However, if the developer of the service composition takes a closer look at the workflow graph, it will not find a good fitting fault of the deadlock. This happens for the reason that the deadlock is caused by a lack of synchronization: The left-hand side fork node has two upper outgoing control flows that are not synchronized by a join node. Only a merge node combines both flows, which possibly results in a lack of synchronization on its outgoing edge. Nevertheless, if, e. g., service A needs much more time than service B, the control flow of service B reaches the join node on the right-hand side before the control flow, which performs service A. Because of this, the join node can be executed before the lack of synchronization appears. Then, however, the workflow graph runs into a deadlock although there is the fault of a wrong control flow synchronization.

So, in short, a fault illusion is the appearance of an error although the faults of other errors cause it. The finding of such a fault illusion is a very hard task in big service compositions. In this context, dynamic analysis techniques are not suitable for fault identification.

E. Fault Blocking

Fault blocking is the condition, in which a fault blocks the further failure detection [35]. Since software testing aims at detecting the presence of errors only, *fault blocking* is not bad. However, when it is the goal to find as many errors as possible, fault blocking makes the fault detection time expensive since a necessary analysis has to run at least for each error (which can be an arbitrary large number, e.g., in the case of lacks of synchronization). It is easy to see that it is not possible to detect errors *after* a deadlock in dynamic approaches since there is no further reachable state. As a result, it is not possible to detect all *errors*, let alone *faults*, within a service composition with dynamic approaches.

Another difficulty of fault blocking is that one error may result in another error. This is linked to fault illusion. In Figure 13, we see a simple workflow graph, in which a split node causes (local) deadlocks in the upper and lower join nodes. However, as we can also see, the deadlock of the lower join node is caused by the deadlock of the upper one, i. e., if the upper join node would be a merge node, the deadlock of the lower join node disappears. Therefore, the deadlock of the lower join node is the result of the blocking of a control flow of the upper join node. Since a dynamic error finding approach like state space exploration may return the deadlock of the lower join node, it is hard to find its fault.

F. Discussion

In summary, dynamic approaches are dependable analyses considering soundness *verification* of workflows. They decide trusty whether a workflow is sound or not. However, the case study has shown their weaknesses as tool support for workflow developers seriously. The typical problems

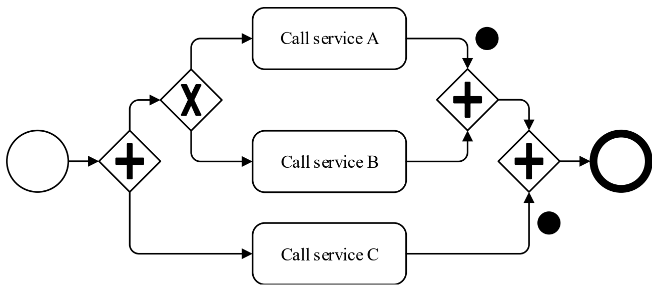


Figure 13. One error blocks another error.

of fault distance, blocking, masking, and illusion complicate the derivation of the fault knowing only the error significantly. They make dynamic approaches inefficient and imprecise. As a result of this case study, to support developers, other tools must be provided.

V. Direct Comparison of Dynamic and Static Analyses

The consideration of dynamic analyses during workflow development showed their weaknesses relating to support the developer seriously. To strengthen this result, we show in this section that problem-specific static analyses have advantages over more general *dynamic* ones. Since the argumentation in the last section is based on a *theoretic* consideration of dynamic approaches verifying soundness, we want to show that these theoretic considerations have relevance in practice. Therefore, we have developed a tool *Mojo*, which uses our static fault finding techniques. Since there are tools like *LoLA* [18] allowing dynamic soundness checks of service compositions, *Mojo* round off the palette of tools by a complete static approach. This makes it possible to compare dynamic and static approaches for soundness checking for the first time.

In this section, we introduce our tool *Mojo* at first. Afterwards, we use *Mojo* to compare static and dynamic analysis techniques.

A. *The Analyser* Mojo

Mojo is a research static analyser, which is freely available and can be downloaded on GitHub [36]. It allows the writing of own analyses, which can be applied as extensions to the system. Conceptionally, *Mojo* is part of our idea of a system for the development and execution of workflows [37], simplified illustrated in Figure 14. In the current state of build, it covers a part of the system's producer side.

On the *producer side* (the static analyser), the *parser* reads the service composition (alias workflow). During the reading, it checks the structure of the input. Afterwards, the *transformer* takes the syntactically correct workflow and transforms it into an intermediate representation (*IR*). *Mojo* uses the notion of workflow graphs as IR. The IR is an abstract format and hides special properties of the entire modelling language, e. g., of BPMN. The *Business Process Executation Language* (BPEL) is difficult to use as IR since it is a structured language whereas most workflows are unstructured.

After the creation of the IR, the composition is checked regarding some semantic properties. An example of such a semantic property is the introduced notion of soundness. If the *semantic analyser* finds faults, the system informs the developer such that the developer can repair the workflow. Otherwise, if the workflow is already correct, it will be encoded and stored within a file or workflow repository.

Besides the producer side, the system even consists of a *consumer side*. The consumer side is a virtual machine. It reads a composition from a file or repository and rebuilds the IR. Furthermore, the *verifier* checks the IR regarding its semantics. This revised check of semantics is necessary to exclude the possibility of manipulations on the IR. With the help of *annotations*, the check can be sped up significantly. In conclusion, the virtual machine executes the workflow and does some runtime analyses in some cases.

Detailed information about our whole system of compiling and executing workflows are available in [37]. An overview about the static analyser and virtual machine can be found in preceding papers [38] and [6].

As mentioned before, the tool *Mojo* can be interpreted as a first version of the producer side of our proposed system. Since *Mojo* is not closed in its functionality and the implementation and testing of new analyses is time-consuming in the context of service compositions, *Mojo* was implemented with the concept of extensions. Extensions (or plugins) allow the easy integration of new analyses and can be used by other researchers without changing the core application. For this, it defines extension points as interfaces, which have to be used to write own plugins. At the moment, extension points for new input languages and new analyses are defined.

In *Mojo*, the order of the performed analyses are defined by *analysis plans*. An analysis plan structures the necessary stages to guarantee correct analyses. In some cases, such a stage can be a complex analysis plan again consisting of different stages. For that reason, *Mojo* follows a classic compiler architecture.

An overview about the current version of *Mojo* is illustrated in Figure 15. The input is possible via files in the languages *Petri Net Markup Language* (PNML) [39] and *BPMN* [2], or directly via programmatic defined workflow graphs. There exist two predefined plugins to enable the input languages PNML and BPMN. Each of these plugins consists of a parser and a transformer.

Afterwards, the resulted workflow graphs can be analysed using the analysis plans. Typical stages of such an analysis plan are a dominance and post dominance analysis [40][41] as well as the determination of the causes of deadlocks and lacks of synchronization. The analysis plan, which should be used, can be defined as a parameter of the tool. Since each analysis plan has a unique number, the precise selection of an analysis plan is easy.

The analysis plan with number 0 performs a soundness inspection with our fault finding techniques explained in previous work [4][5][7] and in Section II. Therefore, (1) Mojo uses a complete static analysis based on well-known compiler theory, (2) it is the first implementation of our soundness checking algorithms, and (3) it finds development

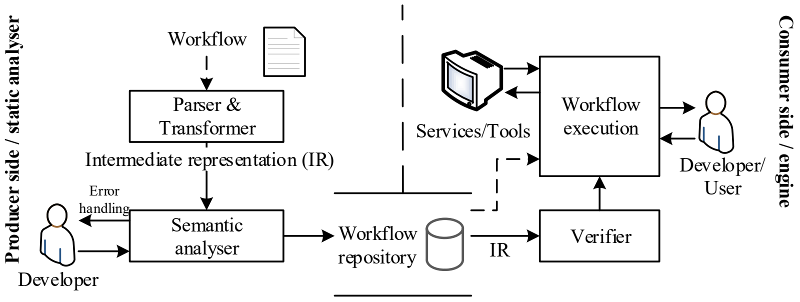


Figure 14. Overview of a system for the development and execution of workflows (taken and simplified from [37]).

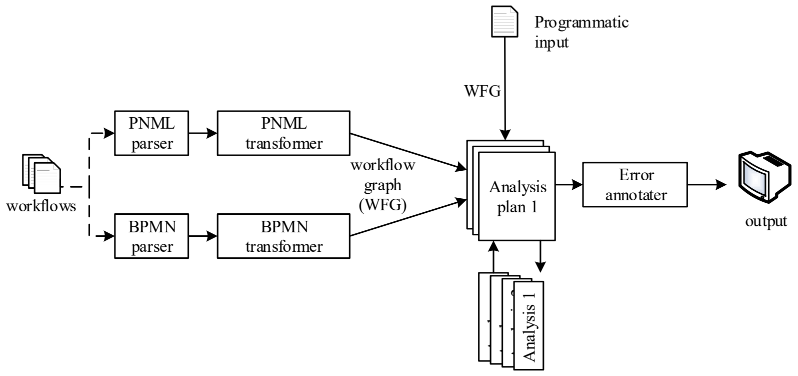


Figure 15. Structure of the static analyser *Mojo*.

faults instead of runtime errors. The found faults will be registered and annotated to the workflow graph. It is possible to get a textual fault output or to use a graphical visualization when *Mojo* is integrated in a workflow modelling tool. Such a tool is the *Activiti BPMN 2.0 Designer*[42], which we have modified to allow analyses with *Mojo*. Figure 16 shows the application of *Mojo* in *Activiti*.

In this application, *Mojo* performs analyses without a visible delay in each modification step of the workflow. Furthermore, the faults can be visited in two modes: (1) The overview mode and (2) the detailed mode. In the overview mode (1), all faults are visualized within the workflow with reduced information. Thus, it is possible to get an overview of the faults within the composition. In the detailed mode (2), the user selects a fault and gets all detailed diagnostic information. That visualizes the fault more precisely to the developer and it should be easier to repair the workflow.

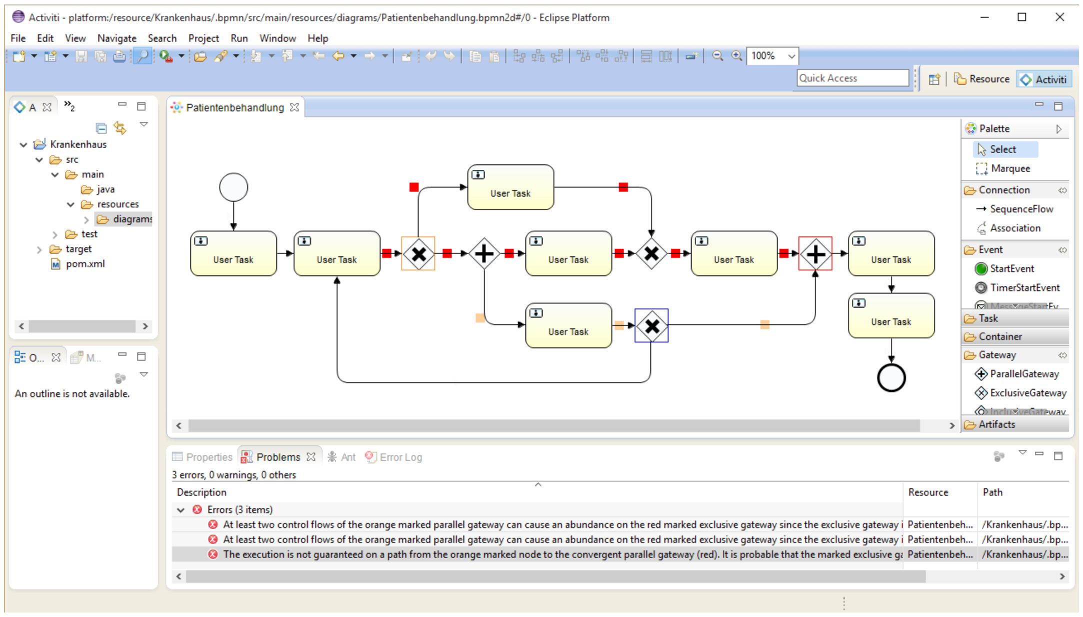
B. Comparison

After the introduction of the tool *Mojo*, it is possible to practically compare the application of static and dynamic analysis approaches for soundness in service compositions. As a typical example of dynamic approaches, we use the state space exploration tool *LoLA*. For static analyses, our compiler-based tool *Mojo* is considered. Both tools were used to study how they perform for real-world service compositions. These service compositions were taken from a business process library [43].

Before we can consider the results of the study, the evaluation settings have to be explained at first. Afterwards, three examples of compositions from the library are considered in detail. For these three compositions, the dynamic approach leads to different results than a static approach in practice. Subsequently to this detailed consideration, we give an overview of the results and differences of static and dynamic analyses of all service compositions of the considered library. At the end, the evaluation shows that a detailed fault analysis has not to be expensive regarding the invested time.

1) Settings: For quantitative statements of evaluation, a large number of test cases is necessary to minimize the effect of irrelevant influencing factors. In the context of soundness checking of workflows, a library of real world business processes of the *IBM WebSphere Business Modeler*[44] is used. That library contains $1,368$ processes (i. e., workflows) and is separated into five benchmarks: A (282 workflows), $B1$ (288 workflows), $B2$ (363 workflows), $B3$ (421 workflows), and C (32 workflows). Thereby, the benchmarks $B1$ to $B3$ describe ongoing improved and developed workflows.

Originally, the library was provided by IBM Zurich. However, the official support was stopped. A more simple parsing and usage is possible in the standardized PNML format. PNML describes Petri nets in a simple syntax with transitions, places and arcs. The workflows are available in the context of the work of Fahland et al. [45][46], who compared different soundness checkers in year 2011. For our evaluation, we have used these PNML files.



```
3 errors, 0 warnings, 0 others
Description                                                                                                                         Resource          Path
Errors (3 items)
  At least two control flows of the orange marked parallel gateway can cause an abundance on the red marked exclusive gateway since the exclusive gateway i  Patientenbeh...  /Krankenhaus/.bp...
  At least two control flows of the orange marked parallel gateway can cause an abundance on the red marked exclusive gateway since the exclusive gateway i  Patientenbeh...  /Krankenhaus/.bp...
  The execution is not guaranteed on a path from the orange marked node to the convergent parallel gateway (red). It is probable that the marked exclusive ga  Patientenbeh...  /Krankenhaus/.bp...
```

Figure 16. Integration of *Mojo* in the Activiti BPMN 2.0 Designer.

We have considered the PNML workflow library with two tools: (1) The previous introduced tool *Mojo* [5] and (2) the state space-based tool *LoLA* [18]. *LoLA* requires the Petri nets to be in a special, proprietary file format. The needed files can be downloaded inclusive an installation guide on [45].

The runtime system for the evaluation was a typical computer with a 64 bit *Debian GNU/Linux 9.0 (stretch)* operating system. The Linux kernel was *4.8.0-2-amd64 x86_64*. The computer used a 4-core *Intel©Core*^{TM}*i5-4570* CPU with 3.2 GHZ frequency and 8 GB main memory. Since *Mojo* is implemented in Java, we run an OpenJDK runtime environment in version *1.8.0_111* with 2 GB heap space.

2) Examples: In our first evaluation setting, we consider some workflows of the library that show remarkable differences between *LoLA* and *Mojo*. We have reduced the complexity of these compositions so that we can illustrate them in the context of this paper.

The first workflow, we consider in more detail, has the name *a.s00000031__s00001361* and is part of benchmark A. The reduced version of this workflow is illustrated in Figure 17 *a)*.

LoLA finds exactly one deadlock (error) for this workflow. It detects it in one of the upper both join nodes depending on the strategy of state space exploration. Furthermore, *LoLA* is able to give a failure trace to that error. In the figure, we have illustrated this trace with the help of tokens, which contain numbers where, e. g., the number 2 describes the position of all tokens in the second state.

In contrary, *Mojo* finds the causes of two potential deadlocks and one potential lack of synchronization (cf. Figure 18 *a)*). That means, *Mojo* finds a structural fault that may lead to a lack of synchronization which *LoLA* cannot detect. In this case, it is impossible to find the potential lack of synchronization with the help of a state space-based approch since there is no state in the whole state space, in which an edge contains more than two tokens. However, if we assume that both upper join nodes should actually be executed correctly, then the lack of synchronization is possible — the static approach of *Mojo* discovers faults behind other faults and ignores the problem of fault blocking.

Furthermore, the static approach of *Mojo* provides all faults with a lot of detailed diagnostic information. Figure 19 shows a possible description of one of the deadlocks of the example, which helps a developer to repair the composition.

In conclusion to the first remarkable composition, the static approach gives more help to the developer than the state space-based approach of *LoLA*.

The second workflow to consider with name *b2.s00000793__s00006437* of benchmark $B2$ is shown in Figure 17 *b)*. In this example, the state space-based approach finds a lack of synchronization and a deadlock. The lack of synchronization is possible after the merge node since two parallel flows can execute that node at the same time. However, if they are run asynchronously, the failure trace of Fig. 17 *b)* is possible containing a deadlock in the join node.

If we consider the same workflow with the static approach of *Mojo*, we find only the cause of one lack of synchronization because the merge node cannot synchronize two parallel control flows (cf. Figure 18 *b)*). Since the deadlock found by *LoLA* results from a lack of synchronization, the static approach does not find it. That behaviour helps the developer since the deadlock is a fault illusion. In conclusion, for this second remarkable case, the static approach shows its benefits since fault illusions are ignored.

The last example shows similarities to the previous one.

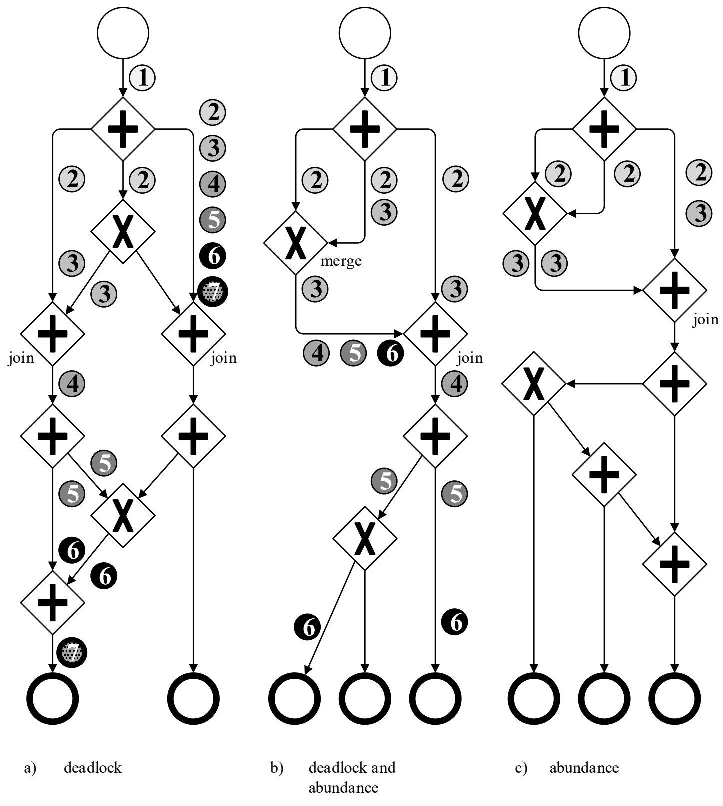


Figure 17. Workflows with noteworthy differences between *LoLA* and *Mojo*: The *LoLA* output.

It has the name *b1.s00000115__s00003189* and is part of benchmark $B1$ (Figure 17 *c)*). Although the example has the same basic structure as the previous workflow, *LoLA* results only in one lack of synchronization, i. e., a deadlock in the upper join node is not reached in the state space. Maybe, the differences in the structures lead to those different results. This could be supported by the fact that *LoLA* stops its error detection after a first error is found — a case of fault blocking. As a consequence, the result of state space-based approaches depends on the strategy and, sometimes, on pure coincidence.

Instead, *Mojo* shows the same behaviour and, furthermore, finds the fault of a potential deadlock in the lower join node (cf. Figure 18 *c)*). That means, fault blocking and fault illusion does not have a chance to occur in static approaches.

As summary of the consideration of the three different workflows of the benchmark, state space-based and static compiler-based approaches show different results. For the three considered examples, a static approach shows more benefits since fault illusions and fault blocking are ignored, the results are transparent, and it provides detailed diagnostic information.

3) Benchmark: It is of interest whether the observations of the last sub section hold in general. For this, we have compared the results of *Mojo* and *LoLA* for the whole workflow library.

Table I shows the number of faults (*not* errors!) found by *Mojo* for the different benchmarks. In total, *Mojo* has found $4,007$ faults. That means, each workflow contains approx. 2 to 3 faults on average. Furthermore, on average a workflow contains more causes for potential lacks of synchronization than deadlocks.

Table I. Number of *faults* found by *Mojo*.

	Deadlocks	Lacks of synchronization
A	140	170
B1	273	720
B2	326	948
B3	289	1,056
C	24	61
Sum	1,052	2,955
Total		4,007

Our expectation was that *LoLA* finds more errors than *Mojo* faults, because an arbitrary number of errors could be derived from one single fault. However, Table II shows a different picture: Only, $1,137$ errors (*not* faults!) were found by *LoLA*. That means, *LoLA* does only find approx. a quarter of the number of errors than *Mojo* faults. One reason for this behaviour is that *LoLA* can find only up to one error per analysis since it stops its state exploration after a first error is found. As *LoLA* performs two separated analyses for the deadlock and lack of synchronization detection, two errors are the maximum.

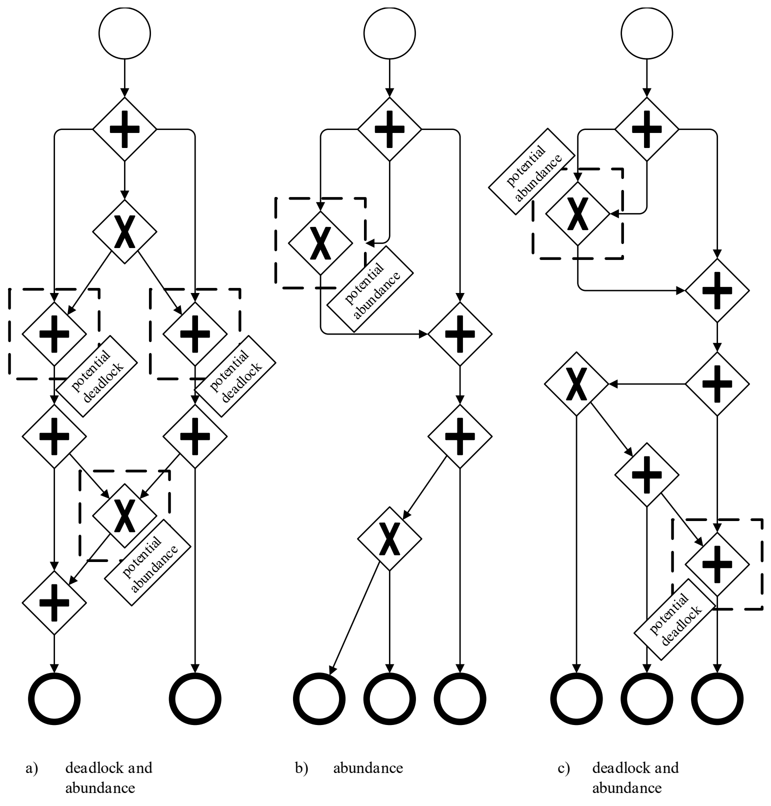


Figure 18. Workflows with noteworthy differences between *LoLA* and *Mojo*: The *Mojo* output.

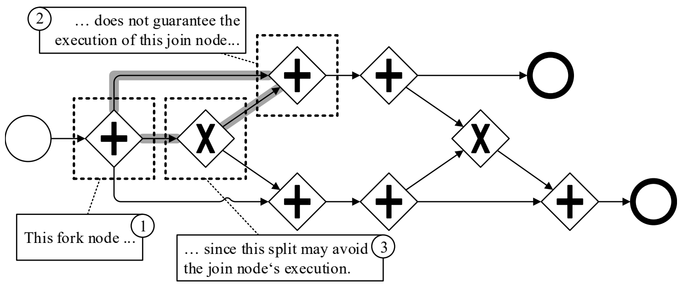

2 … does not guarantee the execution of this join node...
1 This fork node ...
… since this split may avoid 3 the join node's execution.

Figure 19. Detailed fault diagnostic with *Mojo*.

Table II. Number of *errors* found by *LoLA*.

	Deadlocks	Lacks of synchronization
A	97	68
B1	81	200
B2	84	238
B3	83	262
C	10	14
Sum	355	782
Total		1,137

On closer inspection of the differences between *LoLA* and *Mojo*, for 3 workflows *LoLA* finds one lack of synchronization as well as one deadlock. *Mojo*, however, finds only causes of lacks of synchronization for them (cf. example *b)* of Figure 17). For these workflows, the deadlocks are the result of lacks of synchronization as explained before. Such a behaviour could be the case for many workflows, however, since *LoLA* stops its state space exploration after the first found error, they are hard to identify.

For 171 workflows, *Mojo* finds causes of lacks of synchronization and deadlocks; in comparison, *LoLA* reaches only lacks of synchronization (cf. example *c)* of Figure 17). In 205 cases, *LoLA* finds no lack of synchronization since a deadlock blocks the occurrence of a lack of synchronization at runtime (cf. example *a)* of Figure 17). However, *Mojo* finds both: Lacks of synchronization as well as deadlocks.

All those cases strengthen our hypothesis that the static

Table III. Number of sound and unsound workflows with *Mojo* and *LoLA*.

		Mojo		*LoLA*	
	Total	sound	unsound	sound	unsound
A	282	152	130	152	130
B1	288	107	181	107	181
B2	363	161	202	161	202
B3	421	207	214	207	214
C	32	17	15	15	17
Summe	1,386	644	742	642	744

soundness approach used by *Mojo* has many advantages over the state space-based approach of *LoLA*.

We also have checked whether a static approach can be used for pure soundness checking. Therefore, we have checked whether *Mojo* and *LoLA* detects the same sound and unsound workflows in the library. Furthermore, we have counted the total number of sound and unsound workflows. Table III shows the results of this aggregation for *Mojo* (left) and *LoLA* (right).

As it turned out, *Mojo* and *LoLA* mark the same workflows as sound except for two of them. *Mojo* marks both as sound, *LoLA* as unsound. They have the names *c.s00000042__s00001033* and *c.s00000042__s00001050* and are part of benchmark C. This can also be observed in Table III on the differences between the total number of sound and unsound workflows. We considered both compositions in detail and worked out that both workflows are unconnected. Since *Mojo* was implemented as a tool, which supports the *development* of workflows, it must be able to handle unconnected workflows. For this, it builds a connected workflow using the semantics of BPMN. After a long inspection of both workflows, we could not find any fault. Since we do not know how *LoLA* handles unconnected workflows, we assume that this fact is the origin of the divergence of both tools. As a consequence, the results of Fahland et al. [46] should be handled with care since they consider the same process library and the tool LoLA too.

In summary, we see that the usage of static analysis approaches has benefits in the context of soundness verification. It results in a more detailed fault overview ignoring difficulties like fault illusion and fault blocking.

4) Time Behaviour: In the last step of our evaluation, we want to take a look on the time behaviour of both tools since it is possible that static approaches are more time expensive than dynamic techniques. To be used in an IDE, analyses should be fast such that they can be performed during each modification step of a program or, in our case, of a service composition. Figure 20 shows the distribution of the analysis times of *Mojo* and *LoLA* in two histograms. As we can see in the figure, *Mojo* spends for approx. 95 % of the workflows less than two milliseconds to find the faults of deadlocks and lacks of synchronization. It needs $0.03\,[ms]$ in the best, $0.25\,[ms]$ in the median, and $24.5\,[ms]$ in the worst case. In summary, *Mojo* works very fast with the workflows of the library and can be used without a noticeable latency.

LoLA spends a bit more time for its analysis of the workflows (bottom histogram). Nearly 67% of the workflows need approx. $5\,[ms]$ to be analysed with *LoLA*. Almost all workflows ($99\,\%$) of the library can be analysed within 5 to 10 milliseconds. In the minimum, *LoLA* needs $5.26\,[ms]$, in the median $5.78\,[ms]$, and in the maximum $17.97\,[ms]$ for the analysis of a single workflow.

In total, *Mojo* spends 818 milliseconds to check all $1,386$ workflows. *LoLA* uses $8,238$ milliseconds instead and is, therefore, approx. 10 times slower than *Mojo*. Although it is slower than *Mojo*, it has a respectable time and can also be used without any visible latency. However, as shown in this evaluation, the usage of a static analysis as used by *Mojo* has more advantages than a state space-based approach as used by *LoLA*.

VI. Conclusion and Future Work

The major advantages of well-known analyses used in modern IDEs for software development are the extensive diagnostic information and the possibility to find potential failures along the whole program. We have shown in a case study that dynamic analysis techniques can result in an imprecise and time consuming error detection. Though, most analyses for service compositions do use dynamic error finding techniques as motivated in the state of the art. But dynamic techniques can only find first appearing errors since afterwards the program is within a dirty state. This makes it difficult and inefficient to repair a defect composition. Furthermore, the case study showed that *dynamic analysis techniques are not suitable as immediate tool support during the development of service compositions*. This fact was strengthen by a direct comparison of dynamic and static analyses. The comparison was done with our static fault finding technique and the dynamic state space exploration. The static analysis techniques give detailed and precise fault information throughout a whole process, whereas the dynamic analysis techniques are only suitable for which they were made for: Verification, i. e., checking whether a verification criterion holds or not. So, dynamic error finding techniques like state space exploration have some serious disadvantages during the reparation of malformed service compositions.

The introduced tool *Mojo* shows the strengthen of applying *static* analyses during the creation of workflows. The analyses cannot only be performed in each modification step of the service composition, they also give detailed diagnostic information about the faults. For this reason, *Mojo* is the first tool that can be used profitable as an extension to a service composition modeller making the modeller to a first IDE. We believe that there are many other composition-specific problems, which can be avoided by static analyses.

Although there is a substantial common ground between the creation of service compositions and a software product, there are some serious differences making the adaptation of static analyses from classical software development to service compositions difficult: (1) In most cases, service compositions are developed by the use of visual modelling languages, e. g., *BPMN* and Event-driven process chains [47]. Visually modelled compositions often result in unstructured workflow graphs, e. g., approx. $60\,\%$ of all real world processes taken from IBM Zurich [43] are unstructured. Un-

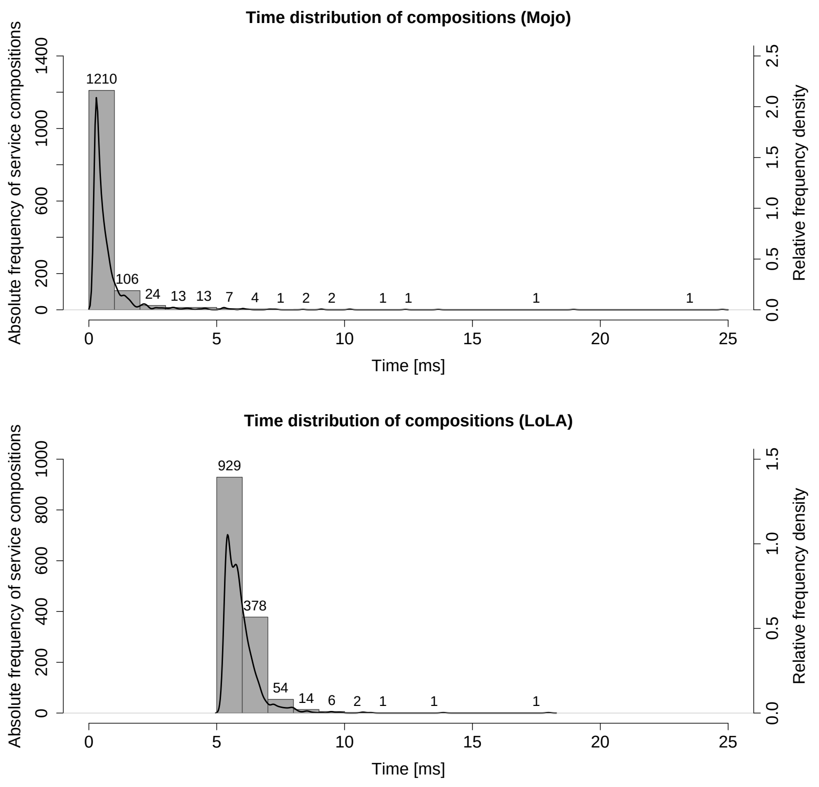


Figure 20. Distribution of analysis times for *Mojo* (top) and *LoLA* (bottom).

fortunately, most known fast analysis algorithms of compiler theory work only for structured graphs.

(2) A second major difference between the development of software systems and service compositions is the ability to model explicit parallelism within service compositions. Since most algorithms for program analysis cannot be applied to parallel programs, they must be adapted [48]. Lee et al. [49] introduced the *Concurrent Static Single Assignment* (CSSA) form, making it possible to use algorithms of sequential programs for parallel software. Unfortunately, the building of the CSSA form requires knowledge about possible race conditions to ensure high quality analysis results. The derivation of race conditions, however, is inefficient for unstructured workflow graphs so far [49].

In summary, we plead for an adaptation of fast and well-known analysis techniques of modern IDEs to the development of service compositions. Furthermore, we argue for the development of new static analysis techniques especially for service compositions to solve composition-specific problems. In this context, we also plead for a first real compiler for service compositions, which enables those analyses as well as the transformation of service compositions into runnable applications [37]. The practical benefits of such an approach were demonstrated by the introduction of the analysis tool *Mojo* and its usage in an evaluation of the soundness checking of real world service compositions.

References

[1] T. M. Prinz and W. Amme, "Why We Need Advanced Analyses of Service Compositions," in SERVICE COMPUTATION 2017: The Ninth International Conferences on Advanced Service Computing, Athens, Greece, February 19–23, 2017. Proceedings, pp. 48–54.

[2] Object Management Group (OMG), "Business Process Model and Notation (BPMN) Version 2.0," OMG, Jan. 2011, standard. [Online]. Available: http://www.omg.org/spec/BPMN/2.0

[3] W. M. P. van der Aalst, "A class of Petri nets for modeling and analyzing business processes," Eindhoven University of Technology, Eindhoven, Netherlands, Computing Science Reports 95/26, 1995, Technical Report.

[4] T. M. Prinz and W. Amme, "Practical Compiler-Based User Support during the Development of Business Processes," in Service-Oriented Computing - ICSOC 2013 Workshops - CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium, Berlin, Germany, December 2-5, 2013. Revised Selected Papers, pp. 40–53.

[5] T. M. Prinz, N. Spieß, and W. Amme, "A First Step towards a Compiler for Business Processes," in Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, pp. 238–243.

[6] T. M. Prinz, R. Charrondière, and W. Amme, "Geschäftsprozesse kompiliert - Wichtige Unterstützung für die Modellierung," in Proceedings 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach am Wörthersee, Austria, pp. 476–491.

[7] T. M. Prinz, "Entwicklung von kontrollflussbasierten Methoden und Techniken für einen benutzerfreundlichen Entwurf von sicheren Geschäftsprozessen," Ph.D. dissertation, Friedrich-Schiller-

Universität, Jena, Germany, Oct 2017.

[8] F. Puhlmann, "Soundness Verification of Business Processes Specified in the Pi-Calculus," in On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I, pp. 6–23.

[9] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn, "Soundness of workflow nets: classification, decidability, and analysis," Formal Aspects of Computing, vol. 23, no. 3, May 2011, pp. 333–363.

[10] J. Desel and J. Esparza, Free Choice Petri Nets, ser. Cambridge Tracts in Theoretical Computer Science, C. van Rijsbergen, S. Abramsky, P. H. Aczel, J. W. de Bakker, J. A. Goguen, Y. Gurevich, and J. V. Tucker, Eds. Cambridge, Great Britain: Cambridge University Press, 1995, no. 40.

[11] P. Kemper and F. Bause, "An Efficient Polynomial-Time Algorithm to Decide Liveness and Boundedness of Free-Choice Nets," in Application and Theory of Petri Nets 1992, 13th International Conference, Sheffield, UK, June 22-26, 1992, Proceedings, pp. 263–278.

[12] C. Favre and H. Völzer, "Symbolic Execution of Acyclic Workflow Graphs," in Business Process Management - 8th International Conference, BPM 2010, Hoboken, NJ, USA, September 13-16, 2010. Proceedings, pp. 260–275.

[13] W. M. P. van der Aalst, "Verification of Workflow Nets," in Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings, pp. 407–426.

[14] A. Valmari, "The State Explosion Problem," in Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996, pp. 429–528.

[15] T. Murata, "Petri Nets: Properties, Analysis and Applications," Proceedings of the IEEE, vol. 77, no. 4, Apr. 1989, pp. 541–580.

[16] N. Lohmann and D. Fahland, "Where Did I Go Wrong? - Explaining Errors in Business Process Models," in Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings, pp. 283–300.

[17] H. M. W. E. Verbeek, T. Basten, and W. M. P. van der Aalst, "Diagnosing Workflow Processes using Woflan," The Computer Journal, vol. 44, no. 4, Sep. 2001, pp. 246–279.

[18] K. Schmidt, Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000 Aarhus, Denmark, June 26–30, 2000 Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, ch. LoLA A Low Level Analyser, pp. 465–474.

[19] W. Sadiq and M. E. Orlowska, "Analyzing Process Models Using Graph Reduction Techniques," Information Systems, vol. 25, no. 2, Apr. 2000, pp. 117–134.

[20] R. Eshuis and A. Kumar, "An integer programming based approach for verification and diagnosis of workflows," Data & Knowledge Engineering, vol. 69, no. 8, Mar. 2010, pp. 816–835.

[21] P. Chrząstowski-Wachtel, B. Benatallah, R. Hamadi, M. O'Dell, and A. Susanto, "A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling," in Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings, pp. 336–353.

[22] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition," in Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings, pp. 43–55.

[23] J. Vanhatalo, H. Völzer, and J. Koehler, "The Refined Process Structure Tree," Data & Knowledge Engineering, vol. 68, no. 9, Sep. 2009, pp. 793–818.

[24] J. E. Hopcroft and R. E. Tarjan, "Dividing a Graph into Triconnected Components," SIAM Journal on Computing, vol. 2, no. 3, Sep. 1973, pp. 135–158.

[25] R. Johnson, D. Pearson, and K. Pingali, "Finding regions fast: Single entry single exit and control regions in linear time." Cornell University, Ithaca, NY, Ithaca, NY, USA, Tech. Rep. TR 93-1365, Jul. 1993.

[26] R. Johnson, D. Pearson, and K. Pingali, "The Program Structure Tree: Computing Control Regions in Linear Time," in Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994, pp. 171–185.

[27] B. F. van Dongen, J. Mendling, and W. M. P. van der Aalst, "Structural Patterns for Soundness of Business Process Models," in Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16-20 October 2006, Hong Kong, China, pp. 116–128.

[28] C. Favre, "Detecting, Understanding, and Fixing Control-Flow Errors in Business Process Models," Ph.D. dissertation, ETH Zürich, Zürich, Switzerland, 2014, DISS. ETH NO 22266.

[29] C. Favre, H. Völzer, and P. Müller, "Diagnostic Information for Control-Flow Analysis of Workflow Graphs (a.k.a. Free-Choice Workflow Nets)," in Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, pp. 463–479.

[30] M. B. Juric, B. K. Mathew, and P. Sarang, Business Process Execution Language for Web Services: An Architects and Developers Guide to BPEL and BPEL4WS, second edition ed., ser. From Technologies to Solutions, M. Little and D. Shaffer, Eds. Birmingham, UK: Packt Publishing Ltd., Jan. 2006.

[31] W. M. P. van der Aalst and A. H. M. ter Hofstede, "YAWL: Yet Another Workflow Language," Information Systems, vol. 30, no. 4, Jun. 2005, pp. 245–275.

[32] C. A. Petri, "Communication with Machines (Kommunikation mit Maschinen)," Ph.D. dissertation, Faculty for Mathematics and Physics, Technische Hochschule Darmstadt, Bonn, Jul. 1962.

[33] A. Avizienis, J. Laprie, B. Randell, and C. E. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, 2004, pp. 11–33.

[34] I. Sommerville, Software Engineering, 8th ed. Munich, Germany: Pearson Studium, Apr. 2007.

[35] M. A. Friedman and J. M. Voas, Software Assessment: Reliability, Safety, Testability, 1st ed., ser. New Dimensions In Engineering Series. New York, USA: John Wiley & Sons, Inc., Aug. 1995, vol. Book 16.

[36] T. M. Prinz, "GitHub – mojo.core," website, visited on July 18th, 2017. [Online]. Available: https://github.com/guybrushPrince/mojo.core

[37] T. M. Prinz, T. S. Heinze, W. Amme, J. Kretzschmar, and C. Beckstein, "Towards a Compiler for Business Processes - A Research Agenda," in SERVICE COMPUTATION 2015: The Seventh International Conferences on Advanced Service Computing, pp. 49–54.

[38] T. M. Prinz, "Proposals for a Virtual Machine for Business Processes," in Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015, Jena, Germany, February 19-20, 2015., pp. 10–17.

[39] M. Weber and E. Kindler, "The Petri Net Markup Language," in Petri Net Technology for Communication-Based Systems - Advances in Petri Nets, pp. 124–144.

[40] R. T. Prosser, "Applications of Boolean Matrices to the Analysis of Flow Diagrams," in Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference, pp. 133–138.

[41] E. S. Lowry and C. W. Medlock, "Object Code Optimization," Communications of the ACM, vol. 12, no. 1, Jan. 1969, pp. 13–22.

[42] Alfresco Software, Inc, “Activiti bpm software home,” website, visited on July 18th, 2017. [Online]. Available: https://www.activiti.org/

[43] IBM, “IBM Research - Zurich: Computer Science,” website, visited on January 30th, 2017. [Online]. Available: http://www.zurich.ibm.com/csc/bit/downloads.html

[44] IBM Corporation, “IBM - Software - IBM Business Process Manager,” website, visited on July 18th, 2017. [Online]. Available: http://www-03.ibm.com/software/products/de/modeler-basic

[45] service-technology.org, “service-technology.org/publications,” website, visited on July 18th, 2017. [Online]. Available: http://www.service-technology.org/publications/fahlandfjklvw_2009_bpm/

[46] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf, “Analysis on Demand: Instantaneous Soundness Checking of Industrial Business Process Models,” Data & Knowledge Engineering, vol. 70, no. 5, May 2011, pp. 448–466.

[47] G. Keller, M. Nüttgens, and A.-W. Scheer, “Semantic Process Modelling Based on ”Event-driven Process Chains (EPC)” (Semantische Prozessmodellierung auf der Grundlage ”Ereignisgesteuerter Prozessketten (EPK)”),” Institut für Wirtschaftsinformatik, Saarbrücken, Germany, Veröffentlichungen des Instituts für Wirtschaftsinformatik 89, 1992, Technical Report. [Online]. Available: http://www.iwi.uni-sb.de/iwi-hefte/heft089.zip

[48] S. P. Midkiff and D. A. Padua, “Issues in the Optimization of Parallel Programs,” in Proceedings of the 1990 International Conference on Parallel Processing, Urbana-Champaign, IL, USA, August 1990. Volume 2: Software., pp. 105–113.

[49] J. Lee, S. P. Midkiff, and D. A. Padua, “Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs,” in Languages and Compilers for Parallel Computing, 10th International Workshop, LCPC’97, Minneapolis, Minnesota, USA, August 7-9, 1997, Proceedings, pp. 114–130.