Why We Need Advanced Analyses of Service Compositions

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 (IDEs). 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 building of an IDE for service compositions, which enables immediate failure feedback during the development. For this, there is the need for new research activities on occurring failures and how they can be found. Since most current failure finding techniques are based on accurate approaches, e.g., state space exploration, we show in a case study that the application of accurate techniques is not a suitable solution for IDEs. In most cases, they are either too time consuming or their accurate output does not lead easily to the root of a failure. As a result, we also plead for new advanced analyses of service compositions.**

Keywords–Service Composition; Analysis; Case Study.

I. INTRODUCTION

The function of the *Course Evaluation Service* at the Friedrich Schiller University Jena is the evaluation of lectures as well as of complete courses. Especially for the evaluation of the latter, the department has to handle complex questionnaires with high adaptivity. For this case, there is no standard software, which is able to define, handle, and evaluate such questionnaires.

As part of the service, we develop a software solution — *coast* [1] — which allows handling of more complex surveys than other survey tools do. This solution is based on a service-oriented architecture and uses service compositions to define processes within the system. Figure 1 shows such an abstract service composition, which handles the logic during the execution of a survey.

As the research on service-oriented architectures has passed its 20th anniversary, we expected a wide tool and development support for service compositions. However, it is hard to nearly impossible to find lightweight tools that allow the modeling and execution of service compositions and give immediate development support. Especially, the development support is improvable regarding the programming support of modern integrated development environments (IDEs).

We can find some approaches to verify service compositions in form of business processes in the literature. A large number of those approaches concentrate on the verification of the *soundness* property [2], whereas a sound service composition cannot run into deadlocks or in undesired double executions of services. Since the soundness property is defined on the runtime behaviour of the composition, most algorithms search for undesired behaviour in a simulation. In other words, they regard the *state space* of the composition, whereas the state space defines all possible reachable states.

State space-based algorithms perform *accurate* analyses of service compositions since each found fault can appear at runtime, actually. However, as each fault is a malformed reachable state, which is caused by an *error* within the service composition, finding exactly that error given a specific fault is a hard task. We will demonstrate this in a case study on soundness and derive why it is better to use inaccurate analysis techniques similar to those used by compilers. For this, we repeat a formal and language independent model for service compositions at first — the *workflow graphs* (see Section II). Subsequently, in Section III, we show within the case study on soundness that accurate analysis approaches are not suitable to give profitable tool support. Based on this case study, we explain why it is so important to have an instantaneous development support during the modelling of service compositions (see Section IV). Eventually, this paper closes with a summary and possible future work in Section V.

II. PRELIMINARIES

In the context of graphs, we use the notions $\triangleright n$ and $n \triangleleft$ to describe the sets of incoming and outgoing edges of a graph node n, respectively.

In the rest of this paper, special directed graphs, the *workflow graphs*, are used to describe service compositions in a language independent way. Workflow graphs have different kinds of nodes for parallelisms, decisions and tasks. They were originally introduced by Sadiq and Orlowska [3]:

Definition 2.1 (Workflow Graph): A workflow graph *is a quadruple* $WFG = (N, E, s, e)$ *where* (N, E) *is a* directed graph *with a set* N *of nodes and a set* E *of edges. The graph has a single* start node s *and a single* end node e*. Furthermore, the set* N *of nodes is splitted into disjoint subsets*

$$N = \{s, e\} \cup N_{Task} \cup N_{Split} \cup N_{Merge} \cup N_{Fork} \cup N_{Join}$$

where all nodes within the same subset have the same semantics and appearance. We call each node within N_{Task} *a* task node*. The nodes of set* N_{Split} *are* split nodes *and nodes of the set* N_{Merge} *are* merge nodes*.* N_{Fork} *contains all* fork nodes *whereas* N_{Join} *contains all* join nodes*.*

Furthermore, the nodes have the following properties:

 ISBN: 978-1-61208-528-9

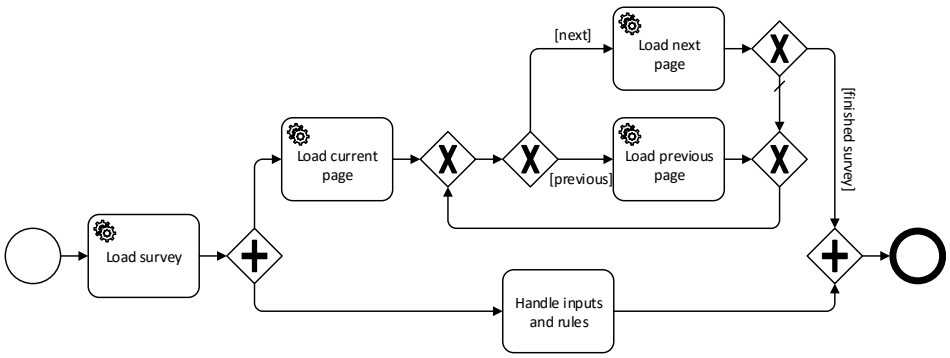

Figure 1. A service composition, which handles the logic of the execution of a survey

1) *s has no incoming and exactly one outgoing edge, whereas e has one incoming and no outgoing edge.*
2) *N_{Task}: Each node has exactly one incoming and outgoing edge.*
3) *N_{Split}/N_{Fork}: Each node has exactly one incoming and at least two outgoing edges.*
4) *N_{Merge}/N_{Join}: Each node has at least two incoming and exactly one outgoing edge.* □

For the visualization of workflow graphs, we use the same notations as the *Business Process Model and Notation* standard [4]. Therefore, tasks are illustrated as simple 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 1).

These different visualizations mark the different semantics of the nodes. To describe the semantics of a node, we use a *token game* known from Petri net semantics [5]. In token games, *states* are used to describe a single execution situation. Such states can also be defined for workflow graphs [6]:

Definition 2.2 (State): A state *of a workflow graph $WFG = (N, E, s, e)$ is a* multiset *S with the basic set E. The multiset assigns a natural number t of tokens to each edge $edge \in E$ of the workflow graph, $S(edge) = t$. We say, $edge$* has *or* carries *a token in state S, if $S(edge) \geq 1$.* □

There are two important states of a workflow graph: (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. In our graph visualizations, we use black dots on edges to illustrate that those edges carry a token in the current state.

In each state (except the termination state), there should be some nodes, which are *executable*, i.e., their functionality can be performed.

Definition 2.3 (Executability): Let $WFG = (N, E, s, e)$ be a workflow graph in a state S. All nodes $n \in N \setminus \{s, e\}$ are executable *in S if either (1) n is not a join node and at least one of its incoming edges has a token, or (2) each of its incoming edges carries a token. The set $\mathcal{E}xec(S)$ contains all nodes, which are executable in S.* □

If a node is executable, its execution directly follows a state transition from one state to another:

Definition 2.4 (State Transition): Assuming a state S of a workflow graph $WFG = (N, E, s, e)$ who contains an executable node $n \in \mathcal{E}xec(S)$. After the node n is executed, S changes *into the state S', written $S \xrightarrow{n} S'$. S' is defined as follows:*

$n \in (N_{Task} \cup N_{Fork} \cup N_{Join})$:
Each incoming edge in of n loses one token and each outgoing edge out of n gets a token, $S' = (S \setminus {}^{\triangleright}n) \cup n^{\triangleleft}$.

$n \in (N_{Split} \cup N_{Merge})$:
There is exactly one randomly chosen incoming edge in of n which loses a token and exactly one randomly chosen outgoing edge out of n which gets a token, $S' = (S \setminus \{in\}) \cup \{out\}$. □

Together, the executability and the state transitions form the semantics of each kind of workflow graph node. Summarized, the start and end node have no special semantics. Therefore, they are only used to mark the start and end of a workflow graph. Furthermore, 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-deterministical 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.

As a single state can change into different states, we can define states that are *reachable* from a current state [6].

Definition 2.5 (Reachability): A state S_{to} is directly reachable *from a state S_{from} if S_{from} contains an executable node n whose execution in S_{from} leads to S_{to}.*

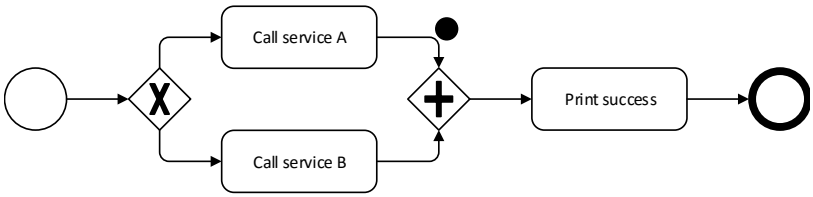

Figure 2. The success message is never printed, so there is a failure

S_{to} is reachable *from state S_{from} (we will write $S_{from} \rightarrow^* S_{to}$) if there is a sequence $S_0, \ldots, S_m$, $m \geq 1$, of states such that $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{m-1} \rightarrow S_m$ and $S_0 = S_{from}$, $S_m = S_{to}$.* □

III. A Case Study on Soundness in the Context of Development Support

Before we argue for advanced analysis techniques for the immediate support during the development of service compositions, we motivate such analyses through a case study. In this case study, we consider the classic notion of *soundness*. A workflow graph is called *sound* if neither a *deadlock* nor a *lack of synchronization* is reachable from the initial state [2] [3]. A *deadlock* is a non-termination state S in which no node is executable, $\mathcal{E}xec(S) = \emptyset$. *Lacks of synchronization* are states S in which at least one $edge$ carries more than one token, $S(edge) \geq 2$.

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. More precisely, a developer of a service composition wants to know, *why* the workflow graph runs into a deadlock or a lack of synchronization.

There are many approaches, which are able to classify whether a workflow graph is sound. The first known algorithm was introduced by van der Aalst [2]. It is based on the rank theorem [7], which can be solved in cubic time complexity regarding the size of the workflow graph [8]. However, this approach does not give any diagnostic information *where* or *why* the workflow graph is unsound [9]. For this reason, other approaches were developed, which we classify into three main approaches: (1) Model checking, (2) graph decomposition, and, finally, (3) pattern and compiler-based approaches. Examples for model checking approaches are the performed *state space explorations* by *LoLA* [10] and Woflan [11]. The *Single-Entry-Single-Exit* (SESE) approach by Vanhatalo et al. [6] is an example for a graph decomposition, whereas the anti-pattern approach of Favre et al. [12] and our compiler-based approach [13] [14] are instances for the latter class of approaches. There are many other significant approaches, however, they resemble one another in their classification.

Most techniques, like the graph decomposition and compiler-based approaches, are profitable in the context of developing service compositions although their output is inaccurate, i.e., they cannot detect faults appearing at runtime. Instead, they find incorrect structures in the compositions *may* leading to a wrong behaviour. To accentuate that inaccurate analysis techniques are suitable for IDEs, we argue in the following case study that accurate analysis techniques lead to a time expensive and hard troubleshooting.

For this case study, we consider as an example the approach of state space exploration, which dominates the literature in process verification for a long time period. Within state space exploration, the *state space* starting at the initial state is examined. Thereby, the state space is a directed graph in which each node is a state and each edge between two states S_1, S_2 means that S_2 is directly reachable from S_1. During the building of this state space, each state is checked whether it is a deadlock or a lack of synchronization. As the state space can have an exponential size depending on the size of the workflow graph, the building of the state space will be broken after the first wrong state is found. As a result, the approach indicates whether the worklow graph is sound. Furthermore, the developer gets a *failure trace*, more precisely, a path within the state space from the initial to the erroneous state.

Now, we compare the verification results of state space exploration with well known software testing terms [15]. This vocabulary makes it possible to evaluate the located faults and how they can be used for troubleshooting. Furthermore, the following comparisons motivate the usage of service composition specific advanced analyses.

A. Failures, Faults, and Errors

In software testing, there are different terms with different meanings for wrong execution states of a program. A wrong state is called a *failure* if an user of the program sees an undesired behaviour or result [15]. For example, in the workflow graph in Figure 2 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. Another example is a composition, which results in duplicated results as some nodes were executed twice in series caused by a lack of synchronization.

Such a lack of synchronization is the manifestation of an incorrect development of the composition. This manifestation is called a *fault* [15]. For example, the process developer may know why the user sees some duplicated results, as the developer may identify that some service calls were performed twice unnecessarily. The reason *why* the service is called twice is called an *error*. An error is the wrong human action during the development of the service

 ISBN: 978-1-61208-528-9

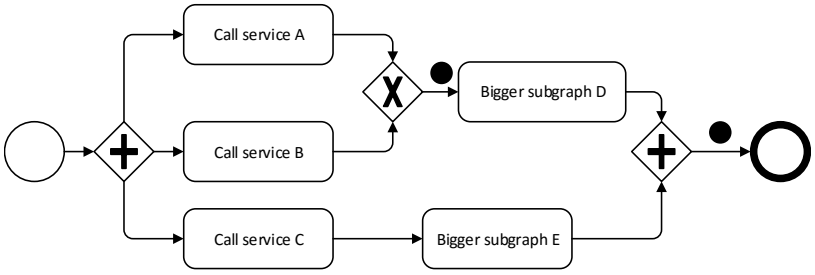

Figure 3. The distance between the fault and its error may be large

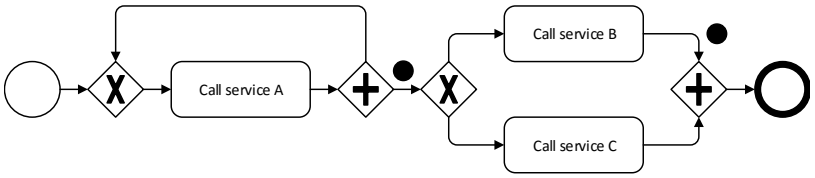

Figure 4. One fault masks another fault so that the failure may disappear

composition [15].

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

Considering the previous term definitions, each accurate analysis technique always results in a fault since it searches within the different execution possibilities of a workflow graph instead at the workflow graph itself. For this, the developer has to derive the real error, to be able to repair the composition. However, this derivation of the error is a hard task since faults can be *masked* or *disguised*. Furthermore, the *distance* between the fault and the error may be very great so that it seems impossible that a fault has its origin so 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 an error results in a fault [16]. The workflow graph in Figure 3 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. Such a detected deadlock state is the result of an accurate soundness approach. Naturally, a developer would now search the corresponding error near the fault. Since a lot of time has passed and the workflow graph is complex caused by the subgraphs D and E, it is very hard to identify the error. A natural and simple correlation is that the difficulty of finding corresponding errors of faults grows with their distances.

C. Fault Masking

Fault masking is the situation in which one fault prevents the detection of another fault [15]. 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.

For example, in a program, one function should process payment information in Euro, however, it processes the data in Dollar instead. Now, another function takes the value and should translate the currency from Euro to Dollar. Coincidentally, within this function, the programmer has forgotten to implement this translation and the value is passed as-is. The result: The program has a correct behaviour as no failure happens although it does not have the desired functionality.

An example of fault masking in the context of service compositions is illustrated in Figure 4. 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. An accurate 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 an accurate approach. This makes the correction of a service composition more time expensive since a necessary analysis has to run for each fault 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 5 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.

 ISBN: 978-1-61208-528-9

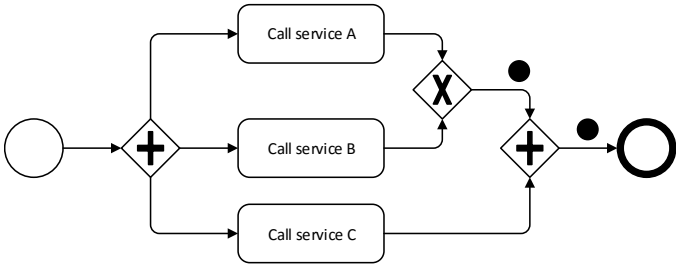

Figure 5. One fault produces another fault so that there is the illusion of an error, which does not exists

An accurate 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 error 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 as 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 error of a wrong control flow synchronization.

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

E. Fault Blocking

In software testing, *fault blocking* is the condition in which a fault blocks the further failure detection [17]. In accurate approaches, it is easy to see that it is not possible to detect faults *after* a deadlock since there is no further reachable state. As a result, it is not possible to detect all *errors* within a service composition. Hence, fault blocking makes the error detection time expensive since a necessary analysis has to run at least for each fault (which can be an arbitrary large number in the case of lacks of synchronization).

Another difficulty of fault blocking is that one fault may result in another fault. This is linked to fault illusion. In Figure 6, 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 an accurate fault finding approach like state space exploration may return the deadlock of the lower join node, it is hard to find its error.

IV. Development Support During the Creation of Service Compositions

In the last section, we have demonstrated in a case study that accurate analysis techniques for tool support during the development of service compositions is not suitable. Now, we argue on the base of this case study, why it is important to support the modelling of compositions.

The development of service compositions is an error-prone task just like the development of software systems. As for the latter exist IDEs, the tool support for the development of service composition is expandable. That is surprising since there is a substantial common ground between both: (1) There is data information passed through variables and (2) there is a flow graph, which represents the structure.

Besides those similarities there are some serious differences making the adaption of analyses from classical software development to service compositions difficult: (1) In most cases, service compositions are developed by the use of visual modeling languages, e.g., *BPMN* and Event-driven process chains [18]. Visually modeled compositions often result in unstructured workflow graphs, e.g., approx. 60% of all real world processes taken from IBM Zürich [19] are unstructured. Unfortunately, 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 [20]. Lee et al. [21] 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 difficult for unstructured workflow graphs [21].

The major advantages of well-known analyses used in modern IDEs for software development are the extensive diagnostic information and the possibility to find possible failures along the whole program instead of only finding first reachable failures from the start. For example, imagine we use a variable a at line 10 and a variable b at a subsequent line 20 in a program, however, we have forgotten to define both variables before. Now, most modern IDEs will give us a feedback for *both* undefined variables although the variable

 ISBN: 978-1-61208-528-9

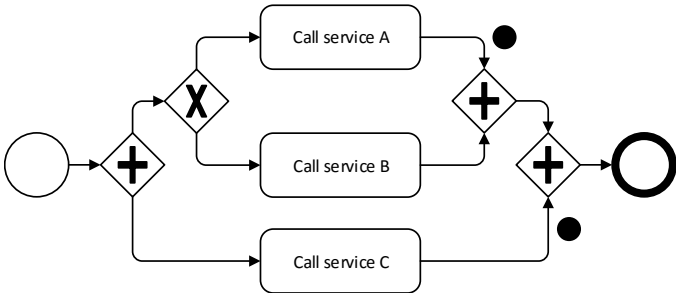

Figure 6. One fault blocks another fault

a appears always before b.

Conversely and as motivated in our case study, most analyses for service compositions do use accurate fault finding techniques, which can only find first appearing faults since afterwards the program is within a dirty state. Furthermore, the accurate finding of faults is more time consuming then performing safe over-approximations as done by classical compiler algorithms. Finally, as we have shown previously in our case study, such accurate fault finding techniques like state space exploration have some serious disadvantages during the reparation of malformed service compositions.

Summarized, we plead for an adaption of fast and well-known analysis techniques of modern IDEs to the development of service compositions. Furthermore, we argue for the development of advanced 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 [22]. Such runnable applications can be executed, e.g., in a virtual machine [23].

V. Conclusion and Future Work

In this paper, we have recommended for the introduction of advanced analyses for service compositions. In this context, we have shown in a case study that accurate analysis techniques can result in an unprecise and time consuming error detection. This makes it difficult to repair a defect composition.

Our proposal is to apply well-known algorithms of compiler theory to the theory of service compositions. Therefore, some basic problems have to be solved in the future: (1) Some compiler algorithms have to be adapted to unstructured graphs. Furthermore, (2) there is the need for an algorithm to find races between two variables in unstructured workflow graphs. Then, it is possible to perform the CSSA building algorithm on service compositions, which enables other algorithms from compiler theory in a parallel context. Finally, (3) there must be a research infrastructure like a compiler to collect and apply new developed algorithms to service compositions. Such a compiler has to be connected with an IDE to support the building of compositions.

References

[1] Course Evaluation Service, “coa.st,” website, visited on January 30th, 2017. [Online]. Available: http://www.coast.uni-jena.de

[2] 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.

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

[4] 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

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

[6] 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.

[7] 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, iSBN 0-521-46519-2.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

 ISBN: 978-1-61208-528-9

[14] 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.

[15] I. C. Society, “IEEE Standard Glossary of Software Engineering Terminology,” IEEE Standard 610.12-1990, Dec 1990, pp. 1–84.

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

[17] 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.

[18] 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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

 ISBN: 978-1-61208-528-9