Process Restructuring in the Presence of Message-Dependent Variables

Thomas S. Heinze[1], Wolfram Amme[1], and Simon Moser[2]

[1] Friedrich Schiller University of Jena,
07743 Jena, Germany
{T.Heinze,Wolfram.Amme}@uni-jena.de
[2] IBM Software Laboratory Böblingen,
71032 Böblingen, Germany
smoser@de.ibm.com

Abstract. When services interact, issues can be caused by service implementations being stateful because a stateful implementation requires a certain message exchange protocol to be followed. At present, a model of such a message exchange protocol is seldom complete and precise, mainly because the available analysis techniques for its derivation suffer from drawbacks: most prominently the neglect of data. Process restructuring allows for the increase of precision of such a data-unaware analysis by resolving conditional into unconditional control flow in service implementations and hence eliminating the need to consider data. But the restructuring approach so far has been restricted to cases where conditions of data-based choices have been defined over quasi-constant variables only. In this paper we introduce a restructuring technique that also allows us to resolve data-based choices with conditions over variables whose value is determined by the contents of incoming messages.

1 Introduction and Motivation

In *service choreographies* and *service orchestrations* issues can occur as soon as one service has a stateful implementation. This would be the case with services that are implemented as business processes, e.g. in languages like WS-BPEL [11] or BPMN [2]. Stateful services require another service that interacts with them to follow a certain protocol in order to prevent runtime errors, e.g. deadlocks. Consider an example: An auction house service offers an interface with two methods: (1) `Bid()` can be used to place a bid and (2) `Abort()` terminates the auction. It requires a bidding service interface for the auction participants with two methods: (1) `BidRequest()` - a notification that bids can be placed and (2) `BidClosure()` - a notification that the auction has finished. An auction participant service offers the bidding service interface and requires an auction house interface. Hence the services should be compatible since each service offers what the other requires. Considering the service implementation shown as BPMN in Fig. 1, it becomes obvious that an auction participant service must eventually invoke the `Abort()` method or the `Bid()` method (with an argument exceeding

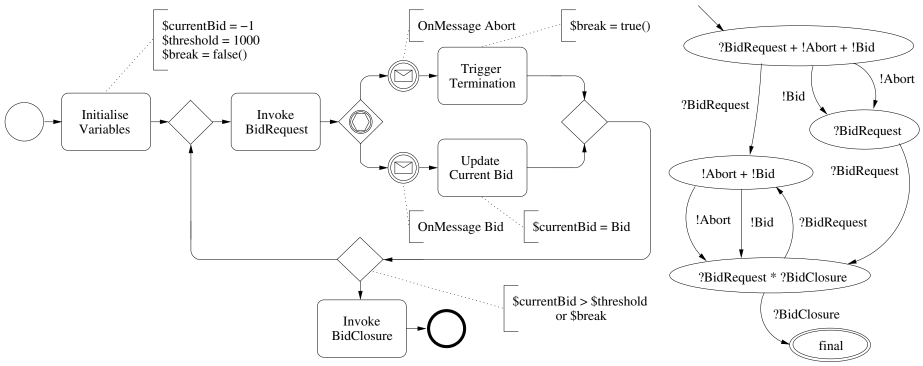

Fig. 1. Service implementation `AuctionHouseService` and its operating guideline

the bidding threshold), i.e. it must follow a certain *message exchange protocol* so that the service interaction terminates. Although there are many forms of this protocol, the underlying idea is the same: It is an automaton describing the externally visible behavior of the service. We will use the term *operating guideline*, coined by [7], for this automaton. Methods calculating an operating guideline analyze the control flow of a service implementation, mainly based on low-level Petri nets, and build the automaton [6,7]. On the right side of Fig. 1, the such derived operating guideline for the auction house service is shown. Therein, all paths are expected to conform to proper interactions with potential partner services. Edges correspond to exchanged messages from the perspective of a partner, i.e. an incoming message is denoted by an edge labeled `"?"` and an outgoing message by an edge labeled `"!"`. Furthermore, additional prerequisites for proper partner services are given as annotations of the automaton's states.

As indicated above, the operating guideline is derived by analyzing a low-level Petri net model of the service implementation's control flow. Data is overall neglected in this model in favor of a feasible analysis. However, by this means, conditional control flow must be mapped to *nondeterminism*, such that the outcome of a data-based choice is independent of its condition, i.e. whether the bidding cycle in the service implementation `AuctionHouseService` is entered or not is chosen arbitrarily within the Petri net model. Consequently, the control flow is over-approximated which hinders precise analysis and the derived operating guideline, shown in Fig. 1, thus provides only a coarse characterization for the behavior of the service. For instance, compatible partners, which send several bid messages `Bid` before awaiting the respective bid request messages `BidRequest`, are not represented and the exit of the bidding is also not explicitly depicted. In [5], we introduced a *process restructuring approach* which allows for the transformation of certain kinds of conditional into unconditional control flow. If effectually applied, there is no need to map the conditional control flow to nondeterminism in the Petri net model because it has been resolved and this source of imprecision can be avoided. As a result, the data-unaware derivation of operating guidelines is able to provide more precise results.

In this paper, we use this approach as the foundation for an advanced technique. Using the new technique then not only broadens the applicability of our restructuring approach, but also refines the derivation of operating guidelines by means of message contents. The remainder of the paper is structured as follows: Sect. 2 introduces the process model used. The restructuring technique is presented in Sect. 3, followed by a discussion of its application to our example in Sect. 4 and related work in Sect. 5. Finally, Sect. 6 concludes the paper.

2 Metamodel for Business Processes

A process model for our purposes must fulfill two requirements: capable of describing more than one business process language and able to precisely reflect the control flow and the data aspects of a service implementation. We therefore introduced *Extended Workflow Graphs* in [5]. Extended Workflow Graphs, abbreviated eWFG, are workflow graphs enriched by a notation of process data. Formally, a workflow graph is a directed graph $WFG = (N, E)$ such that $N = N_{Activity} \cup N_{DivGateway} \cup N_{ConvGateway}$ consists of

- $N_{Activity}$, i.e. nodes to represent activities,
- $N_{DivGateway}$, i.e. nodes to split the control flow in branchings or loops, or mark the beginning of a parallel section - a fork, a decision, or an IOR-split,
- $N_{ConvGateway}$, i.e. nodes to merge the control flow in branchings or loops, or mark the end of a parallel section - a join, a merge, or an IOR-join

and (1) there is exactly one start node $Start \in N$ and one stop node $End \in N$, (2) each diverging control flow node $n \in N_{DivGateway}$ has exactly one incoming edge and two or more outgoing edges, whereas each converging control flow node $n \in N_{ConvGateway}$ has exactly one outgoing edge and two or more incoming edges; each activity has exactly one incoming and exactly one outgoing edge, and (3) each node $n \in N$ is on a path from the start node to the stop node.

Mapping for a well-structured language like WS-BPEL to WFG is straightforward since the points where the control flow diverges and converges are well defined. For BPMN this is more complex because BPMN theoretically allows to model unsound [1] and unstructured processes. However, whether a process is sound can be checked with the methods from [9], and [10] even proposes a refactoring method to make an unstructured process structured. Once a WFG representation has been constructed, it has to be annotated with data flow information. In eWFGs, data is encoded by *Concurrent Static Single Assignment Form (CSSA-Form)*. The key advantage of CSSA-Form is that each variable is defined exactly once, which benefits process analysis. To guarantee this property, variables are renamed: Variable v becomes values $v_1, \ldots, v_n$, one for each of its definitions. As the property is considered static, the definition of a variable inside a control flow cycle is regarded as a single definition, although, the cycle may be executed more than once. Special handling is required if multiple definitions of a variable reach a node via different branches or threads, i.e. at nodes $N_{ConvGateway}$. In these cases, functions $\Phi(v_1, \ldots, v_n)$ are inserted to merge the

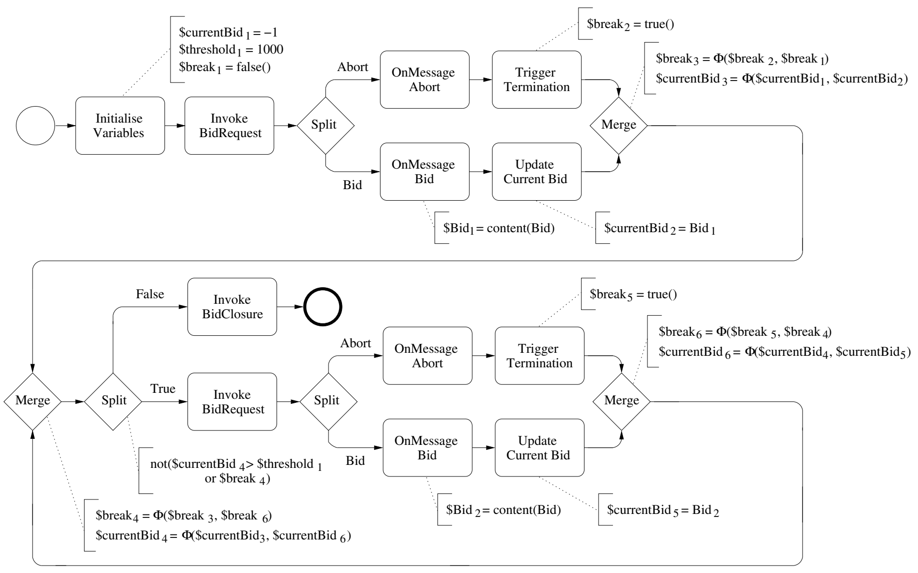

Fig. 2. Extended workflow graph of `AuctionHouseService` process from Fig. 1

confluent definitions $v_1, \ldots, v_n$ of a variable v. The value of such a *Φ-function* is one of its operands: v_i, if the i-th incoming edge denotes the branch taken at process runtime or the parallel thread whose associated operand is defined last.

The eWFG for our example is shown in Fig. 2. For each activity, a distinct node is created and interconnected according to the process's control flow. More complex control flow structures are mapped using pairs of nodes for diverging and converging control flow, i.e. `Split` and `Merge`. Thereby, the cycle in `AuctionHouseService`, which depicts a repeat-until loop, has been transformed into a while loop, due to technical reasons.[1] This can be done safely by extracting the first iteration of the loop and negating the loop condition. Process data are now represented by means of CSSA-Form, such that each static definition of a variable is assigned a unique name, e.g. $\$break_1, \ldots, \$break_6$ for variable `$break`. Furthermore, several Φ-functions have been introduced to merge confluent definitions of variables, e.g. $\$break_4 = \Phi(\$break_3, \$break_6)$.

3 Process Restructuring Technique

In [5], we presented a method for partially eliminating conditional control flow in business processes. More specifically, our *process restructuring approach* allows for the transformation of certain kinds of data-based choices such that data dependences become control dependences, while keeping the execution semantics of

[1] CSSA-Form relies on the presence of while loops. However, every structured loop can be transformed beforehand into a semantically equivalent while loop.

processes unchanged. As a result, conditions of the data-based choices are statically evaluable and can be replaced by unconditional control flow. To this end, the approach relies on a certain class of conditional control flow, characterized by conditions whose variables are defined over constants only. In our example in Fig. 1, the variable `$break` depicts such a *quasi-constant* variable, since it is only defined by the constants `false` and `true`. The condition of the data-based choice in the process is further based on a variable, i.e., `$currentBid`, which is defined by messages. In the following, we will describe a restructuring technique which can be also used in cases where condition variables are defined by messages.

3.1 Basic Restructuring Approach

The restructuring method in [5] has been tailored toward well-structured processes. This means it can cover WS-BPEL processes and a good subset of BPMN, but in its current form is not useable for unsound or unstructured processes. The method can also not be applied to data-based choices with data dependences crossing the boundaries of threads, that is, parallel sections of a process. Furthermore, we will focus on loops, i.e. well-structured control flow cycles, in the subsequent description, by the reason that the restructuring of a structured acyclic branching can be seen as special case of restructuring a loop.

A loop can be effectually processed by our method if the static value of any variable appearing in its loop condition correlates with particular paths in the control flow. Exploiting this property allows the transformation of the loop such that these implicit control dependences become explicit and can be used as substitutes of the loop condition. For instance, the value of a variable which is only defined by constants is in correlation with certain control flow paths, since any path determines the constant in effect. In [5], we called such variables *quasi-constant* and restricted our restructuring approach to loops whose conditions where defined over quasi-constant variables only. Separating the paths which are associated to different assignments of constants to condition variables enabled us to statically evaluate and remove loop conditions for this class of loops.

Restructuring is done in two steps: First, the loop is converted into a *loop normal form*. This normal form is characterized by the separation of all static control flow paths defining different values for variables of the loop condition. To obtain the normal form, nodes, excluding the loop header, where different definitions of condition variables converge, are resolved. After normalization, all definitions of condition variables only converge at the loop header node. Second, the remaining control flow paths which define differing values for condition variables, i.e. the dynamic paths coalesced at the loop header, are separated. The loop is divided into copies of its body, called *loop instances*. Each instance represents the execution of the loop for a certain assignment of condition variables. For loops with conditions of quasi-constant variables, the assignments consist of constants only. Thus, the loop condition is evaluable for each instance and can be replaced by unconditional control flow. Since the number of possible assignments, and so the number of loop instances, is bound by the amount of constants assigned to variables, the transformation is guaranteed to terminate.

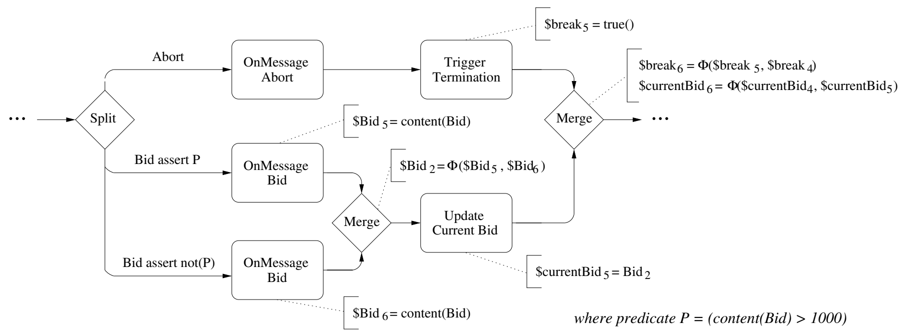

Fig. 3. Splitting incoming message activity `OnMessage Bid`

3.2 Message-Dependent Variables

We now enlarge the applicability of our restructuring approach to cases where a loop condition contains, besides quasi-constant variables, variables that are defined by messages. We call variables of this kind *message-dependent*. To restructure loops with conditions over message-dependent variables, we need a way to associate the possible values of such variables to the control flow.

Depicting messages not only as events, but rather distinguishing the contents of messages, allows for a more fine-grained representation of message-dependent variables. We refine the representation of a message-dependent variable by help of *assertions* for its defining messages. The assertions are based on propositions over predicates, which characterize the possible contents of messages. The definition of a variable by an incoming message is then split into multiple definitions, one for each assertion. This *predicate-based abstraction* allows us to link the values of message-dependent variables as distinct variable definitions to the control flow.

Since our goal is to eliminate conditional control flow of loops, the assertions for incoming messages are derived from the respective loop condition. To this end, the basic predicates of the loop condition are analyzed. Identified predicates can be categorized into quasi-constant predicates, i.e. predicates where only quasi-constant variables are used, and message-dependent predicates, i.e. predicates where additionally message-dependent variables are used. For each message-dependent predicate P, the predicate P and its negation $\neg P$ are used as assertions for the respective incoming message. If a message-dependent variable is used in more than one predicate, all propositions over these predicates form assertions for the message: $P \wedge Q$, $\neg P \wedge Q$, $P \wedge \neg Q$, $\neg P \wedge \neg Q$ for predicates P and Q.[2] The such derived assertions are then utilized for splitting the definitions of those message-dependent variables that are used in the loop condition. Note that, in case of two or more message-dependent variables used in a single predicate, this approach only provides a conservative approximation since we do not examine the actual logical functions realized by condition predicates.

[2] If predicates are logically correlated, e.g. $P = (X > 5)$ and $Q = (X < 10)$, certain assertions are contradictory and can be therefore omitted, e.g. $\neg P \wedge \neg Q$.

Fig. 3 shows the result of splitting one of the incoming message activities contained in the eWFG shown in Fig. 2. The activity is split into two distinct activities, which are separated using message events based on complementary assertions for the incoming message, i.e. *assert* P and *assert* $not(P)$. The assertions are grounded on predicate P, which corresponds to expression $currentBid_4 > threshold_1$ of the loop condition. Therein, referenced variables have been updated according to their actual values, i.e. $threshold_1$ has been replaced by constant 1000 and $currentBid_4$ by the contents of message `Bid`.

3.3 Extended Process Restructuring

A loop with condition over *quasi-constant* and *message-dependent* variables, which respects the restrictions listed in Sect. 3.1, can be restructured in an automated fashion using three steps. First, data dependences for the loop condition are identified. Utilizing the derived information, the definition of each message-dependent condition variable is split into multiple definitions, as described above. Next, the loop is transformed into its *loop normal form* by employing the normalization algorithm in [5]. Finally, the loop is divided into its *loop instances* by using the adaption of the instantiation algorithm introduced below.

In the instantiation process, loop instances are iteratively generated and used as loop replacement. Each instance is guarded by a copy of the data-based choice containing the loop condition, called *instance guard*. If the condition can be evaluated, the guard is replaced by an edge to the exit node of the loop, when evaluation yields false, or, when evaluation yields true by an edge to the instance. The instance itself points to subsequent guards, which are processed in the following iterations. In order to assure the static evaluation of guards, a new notion of loop instance is used. So far, an instance has been considered an execution of the loop with respect to an assignment of constants to variables. When also allowing condition variables defined by messages, this concept is not enough. Instead, we now consider an instance to be an execution of the loop body with respect to an arbitrary *assertion* for the state of condition variables. The state of a quasi-constant variable is then described as value assignment, depicting the constant in effect. The state of a message-dependent variable is characterized by exploiting the introduced assertions for incoming messages. Since these assertions were chosen so that they matched the predicates of the loop condition, the condition can be evaluated for each instance guard by help of elementary logical reasoning and constant expression evaluation. Due to the finite number of predicates, condition variables and assigned constants, the number of loop instances is bound and the instantiation algorithm is again guaranteed to terminate.

The adapted instantiation algorithm is shown in Fig. 4. Procedure *instantiate* consecutively processes instance guards, i.e. data-based choices containing the loop condition, until no further guard exist. To evaluate the condition of a guard, a new assertion A is generated based on the assignment of variables valid at this guard. A constant assignment is thereby added to A for quasi-constant condition variables and assertions of the incoming messages are taken over for message-dependent condition variables. Evaluating the guard based on assertion A allows

```
procedure $instantiate(eWFG = (N, E))$ is
    while ($\exists$ $guard \in N$ such that $guard$ is instance guard) do
        let $values$ be the assignment of condition variables valid at $guard$;
        $A$ = create empty assertion;
        foreach single assignment $v_{c_i} \leftarrow v_i$ in $values$ do
            if ($v_i$ is a message-dependent variable) then
                let $M$ be the incoming message defining $v_i$;
                let $A_M$ be the assertion valid for message $M$;
                add $((v_{c_i} == content(M)) \wedge A_M)$ to assertion $A$;
            else add $(v_{c_i} == v_i)$ to assertion $A$;
            end if;
        end for;
        if $(evaluate(condition(guard), A) == true)$ then
            let $Instance_A$ be the loop instance associated to assertion $A$;
            if (not($Instance_A \subseteq eWFG$)) then
                $eWFG = eWFG \cup Instance_A$;
            end if;
            let $entry_{Instance_A}$ be the entry node of $Instance_A$;
            $E = (E \setminus \{(predecessor(guard), guard)\})$
                    $\cup \{(predecessor(guard), entry_{Instance_A})\}$;
        else $E = (E \setminus \{(predecessor(guard), guard)\})$
                    $\cup \{(predecessor(guard), exit)\}$;
        end if;
    end while;
end.
```

Fig. 4. Adapted algorithm for loop instantiation

us to replace it by an edge to the loop exit, when evaluation yields false, or by an edge to the loop instance $Instance_A$ associated to assertion A, otherwise. If no such instance yet exists, a new one is created. The restructured workflow graph for our example is shown in Fig. 5. Only a single instance has been created while instantiating the loop of the example since only the guard of the instance with assertion $(break_4 == false \wedge currentBid_4 == content(Bid) \wedge not\, content(Bid) > 1000) \wedge threshold_1 == 1000)$ resulted true. Obviously, the data-based choice of the eWFG shown in Fig. 2 was replaced by unconditional control flow.

4 Application to Business Process Analysis

In this section we present the application of our process restructuring technique to the derivation of *operating guidelines*. Existing methods for the construction of operating guidelines are based on a low-level Petri net model of business processes [6,7]. Despite being conservative, such an abstraction provokes imprecise analysis results. In the following, we will show how our process restructuring technique is able to remedy this kind of imprecision for data-based choices with conditions of *message-dependent* and *quasi-constant* variables.

The `AuctionHouseService` example from Fig. 1 is an instance of a process whose derived operating guideline was incomplete. It is to be expected that the

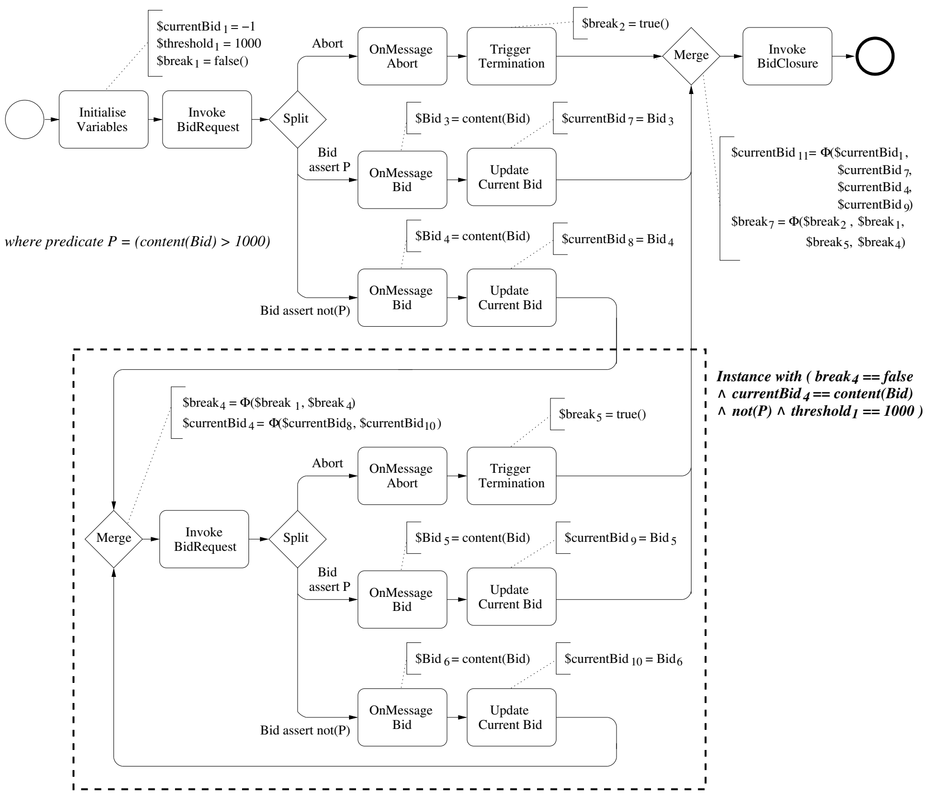

Fig. 5. Restructured workflow graph of `AuctionHouseService`

operating guideline derived from the restructured eWFG will include the missing partner processes. As mentioned, derivation of operating guidelines requires a *Petri net model*. The restructured eWFG of `AuctionHouseService` hence must be mapped onto that model. Since a similar translation from WFG into Petri nets has already been defined in [1], a slightly adopted mapping can be used in our case. Analyzing the Petri net derived from the restructured eWFG results in the operating guideline shown in Fig. 6.[3] Compared to the conventional operating guideline in Fig. 1, the missing compatible partner processes are now represented. For example, the path which traverses edges `!Bid[assert not P]`, `!Bid[assert not P]`, `?BidRequest`, `?BidRequest`, `?BidRequest`, `!Bid[assert P]`, `?BidClosure` depicts a partner that sends two bids not exceeding the threshold value, receives the associated plus an additional bid request, submits a bid exceeding the threshold value, and finally picks up the bid closure message.

As can be furthermore seen, the new operating guideline explicitly models the exit condition of the process. A partner can either send `Abort` or a bid exceeding

[3] Derivation of operating guidelines is only possible for bounded communication channels [7]. Therefore, we depict the operating guideline for channels of size $k = 3$.

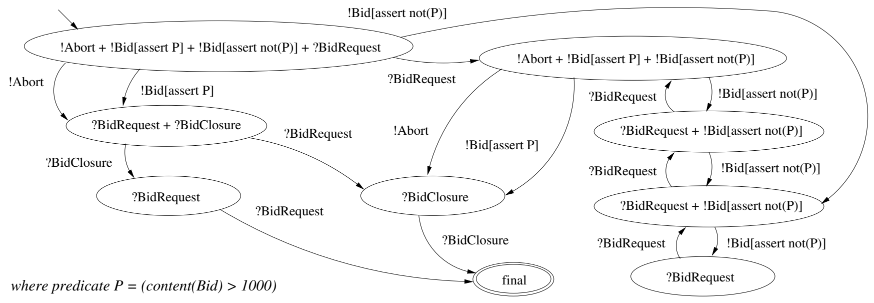

Fig. 6. Refined operating guideline of `AuctionHouseService`

the threshold value, i.e. `Bid[assert P]`, to reach the final state. In comparison, the operating guideline in Fig. 1 masks this exit condition. A partner is required here to determine whether the bidding is over or not by the receipt of message `BidClosure` or `BidRequest`. Masking the exit condition causes the conventional operating guideline to contain spurious behavior, which is not realized by the process. Thus, a path is included where, albeit `Abort` has been transmitted, further bid requests can be received and bids sent, which is not possible with process `AuctionHouseService`. On the one hand, this spurious behavior does not induce a deadlock, but, on the other hand, might result in a livelock. In contrast, the new operating guideline does not contain spurious behavior.

Generally speaking, the operating guideline in Figure 6 is a refinement of the conventional operating guideline such that the therein hidden data-based choice of process `AuctionHouseService`, defining its exit condition, is now made explicit. This refinement is possible since the data-based choice is based on a condition of *message-dependent* and *quasi-constant* variables and our *process restructuring technique* can be effectually applied. As a result, we are able to transform conditional into unconditional control flow and to provide a precise model of the control flow without the need of considering data dependences, which benefits the subsequent Petri-net-based derivation of operating guidelines.

However, there is a price: The potentially confidential threshold value used in the process needs to be announced, as predicate $P = (\mathit{content}(\mathit{Bid}) > 1000)$. Since the conventional operating guideline already provides a description of some compatible partners, it may be in question whether to use the refined operating guideline. This is not an option if an empty operating guideline has been initially derived. Consider the variation of `AuctionHouseService` shown in Fig. 7. Therein, no bid request is sent, like in Fig. 1, but each bid is acknowledged by a confirmation message. Consequently, the internal choice whether to receive further bids or to exit the bidding is not signaled. The analysis in [7] regards this process as not usable and derives an empty operating guideline, which is obviously wrong. For instance, the interaction with a partner which sends a single bid exceeding the threshold value receives the confirmation and the bid closure message does not deadlock. To derive a non-empty operating guideline, a method which allows for a more precise analysis of the process's conditional control flow

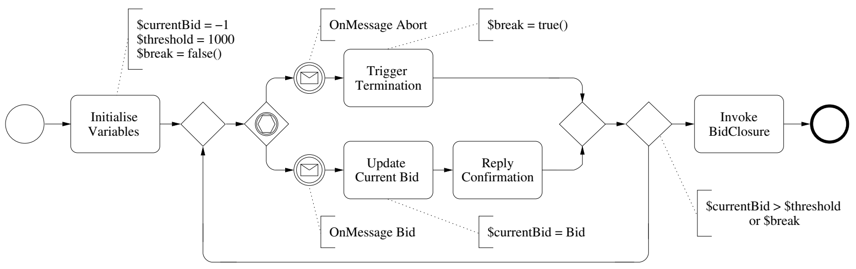

Fig. 7. Variation of `AuctionHouseService`

is mandatory. Moreover, the operating guideline must announce the threshold value used in the branching condition of the process. Thus, describing process behavior by means of a *message exchange protocol* is required here to take data into account, i.e. the contents of exchanged messages.

5 Related Work

Most approaches for control-flow analysis of business processes neglect data, particularly when utilizing Petri nets. If precise analysis of a process's control flow is of concern, a common argument is to use *high-level Petri nets*, which allow for a data-aware process model. In this regard, the inclusion of message contents is sketched in [6] by means of unfolding a high-level into a low-level net, which can be used as analysis input. However, this approach is only feasible for a finite data domain. By help of *predicate abstraction* [4], an infinite domain can be reduced to a finite domain. But, the thereto required predicates need to be properly derived, which is especially challenging in the presence of loops [3]. In our example, predicate `$currentBid` $>$ `$threshold` of the loop condition is apparently a good candidate. The difficulty is to identify and adequately incorporate the data dependences of used variables, i.e. the fact that `$threshold` denotes a constant and `$currentBid` the contents of messages.

In [8], a Petri net model of business processes is extended with a notion of process data, based on an abstract and finite data model. As a result, the such extended process model can also be checked for properties like soundness [1]. However, in contrast to our approach, the used process model is conceptual, i.e. underdefined, and the utilized data abstraction must be provided manually.

6 Conclusion and Outlook

In this paper, we have presented a process restructuring technique and its benefits for the analysis of business processes. The technique allows us to resolve data-based choices with conditions over message-dependent and quasi-constant variables. As a result, data dependences of resolved choices do not need to be taken into account in a subsequent, potentially data-unaware, control-flow analysis. In this way, we were able to demonstrate a refined derivation of message exchange protocols for business processes by means of message contents.

The main issue for future work is the thorough validation and evaluation of our restructuring technique. Having laid its methodological foundation within this paper, we want to assess the practical use of the presented approach by applying it to a collection of real-world processes. Unfortunately, we are not aware of a representative set of business processes, i.e. a common benchmark, which can be used for that purpose. In consequence, we will need to assemble a comprehensive collection of realistic business processes first.

References

1. van der Aalst, W.M.P., Hirnschall, A., Verbeek, H.M.W.: An Alternative Way to Analyze Workflow Graphs. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) CAiSE 2002. LNCS, vol. 2348, pp. 535–552. Springer, Heidelberg (2002)
2. Business Process Model and Notation (BPMN) Version 2.0. OMG Standard, Object Management Group/Business Process Management Initiative (2009)
3. Flanagan, C., Qadeer, S.: Predicate Abstraction for Software Verification. ACM SIGPLAN Notices 37(1), 191–202 (2002)
4. Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
5. Heinze, T.S., Amme, W., Moser, S.: A Restructuring Method for WS-BPEL Business Processes Based on Extended Workflow Graphs. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 211–228. Springer, Heidelberg (2009)
6. Lohmann, N.: A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
7. Lohmann, N., Massuthe, P., Wolf, K.: Operating Guidelines for Finite-State Services. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 321–341. Springer, Heidelberg (2007)
8. Sidorova, N., Stahl, C., Trčka, N.: Workflow Soundness Revisited: Checking Correctness in the Presence of Data While Staying Conceptual. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 530–544. Springer, Heidelberg (2010)
9. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. Data & Knowledge Engineering 68(9), 793–818 (2009)
10. Vanhatalo, J., Völzer, H., Leymann, F., Moser, S.: Automatic Workflow Graph Refactoring and Completion. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 100–115. Springer, Heidelberg (2008)
11. Web Services Business Process Execution Language Version 2.0. OASIS Standard, Organization for the Advancement of Structured Information Standards (2007)