Guided Control Flow Unfolding for Workflow Graphs Using Value Range Information

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

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

Abstract. In our previous work, we have introduced a technique to unfold the control flow in workflow graphs based upon static information about constant data values. Using this technique allowed us to safely transform certain kinds of conditional into unconditional control flow, and thus to support a usually data-unaware verification of business processes by more accurate process models. In this paper, a generalisation of this technique is discussed which can be employed in combination with arbitrary information about data values. This way, we show how statically derived value range information is beneficial for unfolding and therefore eliminating conditional control flow in a wider range of cases.

1 Introduction and Motivation

Verification of business processes today is often done using a Petri-net-based process model in which data aspects are being neglected. Prominent examples are the verification of soundness [8], and the verification of its counterpart controllability [6] in case of distributed business processes. The advantage of these data-unaware approaches lies in the feasible and often efficient analysis that is possible when process data is not considered. However, while such a verification is supposed to provide correct results in most cases, in certain circumstances, false-positive as well as false-negative verification results may occur [4, 10].

Given that properties like soundness or controllability relate to control flow, this kind of wrong verification results is mainly due to an imprecise modelling of business processes' control flow. Reasoned by the ommitance of data aspects within the Petri-net-based process model, conditional control flow is therein over-approximated by nondeterminism, resulting in an abstraction too coarse. In [4], we advocated the use of static analysis and a process restructuring technique to safely transform certain types of a process's conditional control flow into unconditional control flow, before translating the process into its Petri net model. Consequently, over-approximating the such resolved conditional control flow can be avoided, which yields a more precise process model and thus verification.

The restructuring technique presented in [4] is based on the observation, that a branching or loop condition can be statically evaluated if therein referenced variables are assigned constant values only. Since the values of variables, and therefore the value of the condition, correlate with the control flow path taken

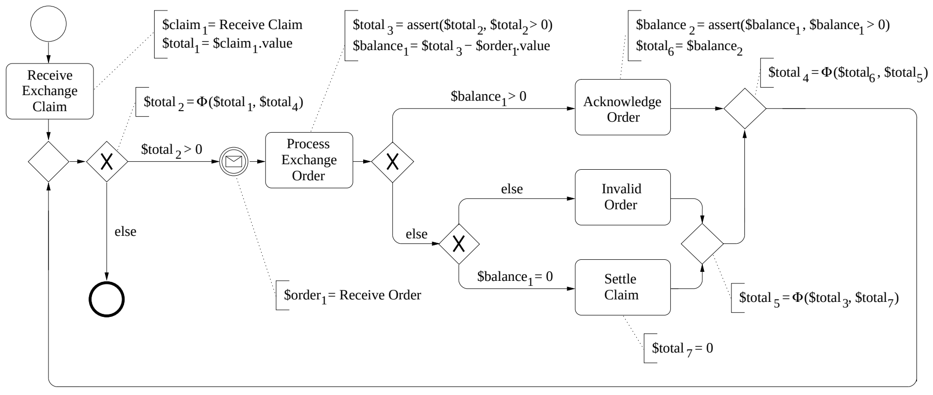
Receive Exchange Claim
$claim_1 = Receive Claim
$total_1 = $claim_1.value
$total_2 = Φ($total_1, $total_4)
$total_2 > 0
else
$order_1 = Receive Order
Process Exchange Order
$total_3 = assert($total_2, $total_2 > 0)
$balance_1 = $total_3 − $order_1.value
$balance_1 > 0
else
else
$balance_1 = 0
Acknowledge Order
Invalid Order
Settle Claim
$balance_2 = assert($balance_1, $balance_1 > 0)
$total_6 = $balance_2
$total_4 = Φ($total_6, $total_5)
$total_5 = Φ($total_3, $total_7)
$total_7 = 0

Fig. 1. Running example: Extended workflow graph

at process runtime, separating and duplicating the control flow for each combination of assigned values then allows for the evaluation and substitution of the condition with unconditional control flow in each duplicate. Since the restructuring technique in its current form is thus restricted to analysis information about constant values, only conditions with variables defined over constants, or single messages, can be resolved. In this paper, we discuss a generalisation of the technique to relax this limitation. For that purpose, the generalised technique is enabled to be used in combination with an arbitrary static analysis which yields an abstraction for the values of process data. In particular, we will show how a value range analysis helps in resolving branching or loop conditions in cases condition variables are not necessarily restricted to constant values.

In principle, our restructuring technique can be seen as an unfolding of a process's control flow. However, existing unfolding techniques restructure the entire process with all variables, though, it is only necessary to unfold those parts and variables related to a branching or loop condition. Further, unfolding at the value level is infeasible in case of infinite data domain such that an abstraction for variables' values is required. Our restructuring approach overcomes these issues by guiding the control flow unfolding based on static analysis information.

The remainder of the paper is structured as follows: The next section introduces the process representation format and analysis used to derive value range information. In Section 3, we describe our generalised technique for guided control flow unfolding and its use in combination with value range information. Related work is discussed in Section 4. Finally, Section 5 concludes the paper.

2 Workflow Graphs and Value Range Analysis

In order to allow for the static derivation of information essential to our guided control flow unfolding approach, a process representation format is required which is capable of representing both, control as well as data aspects of a business process. We therefore use an extension of workflow graphs [4, 8]. Workflow

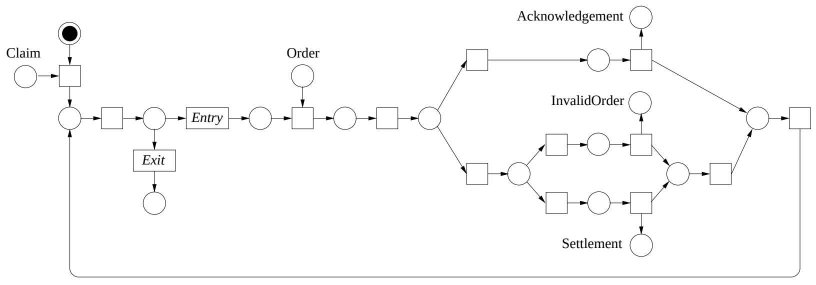

Fig. 2. Petri-net-based process model (Open workflow net [6])

graphs support a simple and flexible modelling of a process's control structure. However, since workflow graphs only map control flow, we augment them with a notation of process data. Thus, nodes and edges of a workflow graph are annotated with data manipulation statements in Concurrent Static Single Assignment (CSSA-)Form [5], which yields an easy to analyse model for a business process's control and data flow. Since we here refer to the verification of fully-specified, i.e., executable, business processes, processes can be translated into their process representation through extended workflow graphs in an automated fashion [2].

In Figure 1, an example process is shown in its representation as extended workflow graph (whose visualisation here closely follows the Business Process Model and Notation). The depicted process models the action of item exchange. A customer therein first specifies the item for exchange by sending message `Claim`. Afterwards, the customer is allowed to order exchange items via message `Order`, where each is acknowledged by message `Acknowledgement`, as long as the total value of orders does not exceed the value of the item for exchange. Otherwise, the last order is rejected, indicated via message `InvalidOrder`. In case the total value of ordered items eventually equals the value of the item for exchange, the claim is settled and message `Settlement` is sent to the customer.

For its realisation, the process is based on a loop whose execution is conditioned. In the Petri-net-based process model, as shown in Figure 2, the loop is mapped to the nondeterministic choice of transitions *Entry* and *Exit*, such that the loop condition is not precisely represented. In consequence, verifying the process using this Petri net model yields erroneous results for properties like controllability, e.g., the process is verified to be non-controllable although it is.

In contrast, the extended workflow graph explicitly models the loop condition: Loop execution is controlled by an integer variable $\texttt{\$total}_2$, representing the difference of the value of the item for exchange and the total value of all already ordered exchange items. Accordingly, if the value of the variable is greater than zero, the loop is executed, and otherwise exited. Therefore, the value of $\texttt{\$total}_2$ is initially set to the value of the item for exchange (message part $\texttt{\$claim}_1\texttt{.value}$), and afterwards updated for each loop iteration with the difference of its current value and the value of an accepted exchange item (message part $\texttt{\$order}_1\texttt{.value}$). As can be seen, all variables are statically only defined

Variable	Derived information	Variable	Derived information
`$claim`$_1$	*undefined*	`$order`$_1$	*undefined*
`$claim`$_1$`.value`	$(0, +\infty]$	`$order`$_1$`.value`	$(0, +\infty]$
`$balance`$_1$	$[-\infty, +\infty]$	`$balance`$_2$	$(0, +\infty]$
`$total`$_1$	$(0, +\infty]$	`$total`$_2$	$[0, +\infty]$
`$total`$_3$	$(0, +\infty]$	`$total`$_4$	$[0, +\infty]$
`$total`$_5$	$[0, +\infty]$	`$total`$_6$	$(0, +\infty]$
`$total`$_7$	$[0, 0]$		

Table 1. Derived value range information

once, as is indicated by the variables' subscripts. This is the main characteristic of CSSA-Form and vitally supports analysis since variables then coincide with their definition statements. Although, special handling is required if multiple variable definitions have to be joined at a single node of the extended workflow graph. In these cases, statements with so-called Φ-functions are used to merge the confluent definitions into a single value, as is done for the definitions of variables `$total`$_1$ and `$total`$_4$ by statement `$total`$_2 = \Phi($`$total`$_1$`,$total`$_4$`)`.

To further improve analysis, the implications of branching and loop conditions are annotated in terms of assertion statements. For instance, since the loop in the example process is only executed if the value of variable `$total`$_2$ exceeds zero, we know that the variable must be a positive integer within the body of the loop. Therefore, uses of `$total`$_2$ in the loop body are substituted with a reference to a new variable which is defined by `$total`$_3 =$ `assert($total`$_2$`,$total`$_2 > 0$`)`, indicating that `$total`$_3$ equals `$total`$_2$ and has a value greater than zero.

Based on the representation of business processes through extended workflow graphs, various static analysis become available. In particular, the use of CSSA-Form facilitates the transfer of analysis techniques from the area of compiler optimisation, like, e.g., constant propagation, global value numbering [5], or value range analysis. Especially the latter analysis provides an abstraction for the values of process data which benefits a guided control flow unfolding.

Value range analysis is a textbook data flow analysis technique which can be used to derive an interval for each integer or floating-point variable and point of a program, such that the variable is guaranteed to take a value in the interval at the given program point. In [2], we have implemented such an analysis for extended workflow graphs and show how it can be used to derive value range information for processes of a small subset of the WS-BPEL language.

An application of the analysis to the example process of Figure 1 yields the value range information shown in Table 1. Note that, therein, each variable is assigned a single interval which is valid at each point of the process, due to the static single definition of variables in CSSA-Form. Further, the analysis defined in [2] is able to exploit data type definitions in business processes for deriving more precise value range information. In case of the example process, the derived intervals for `$claim`$_1$`.value` and `$order`$_1$`.value` therefore only comprise positive integers since the corresponding message types are set to `xsd:positiveInteger`.

```
// let $eWFG$ be an extended workflow graph and let $loop$ denote a loop therein
$inf = analyse(eWFG)$;              // $inf\colon Variables \rightarrow AnalysisInformation$
normalise $loop$ in $eWFG$ and derive instance pattern as is explained in [4];
while ($\exists\ guard \in eWFG$ such that $guard$ is an instance guard) do
    $assertion = \emptyset$;              // $assertion\colon Variables \rightarrow AnalysisInformation$
    let $values$ be the assignment of condition variables valid at $guard$;
    foreach single assignment $(variable_{condition} \leftarrow variable)$ in $values$ do
        $assertion = assertion \cup \{variable_{condition} \mapsto \inf(variable)\}$;
    end for;
    if $(evaluate(guard,\ assertion) == true)$ then
        let $instance$ be the loop instance for $assertion$;
        if $(instance \nsubseteq eWFG)$ then $eWFG = eWFG \cup instance$;
        end if;
        replace $guard$ with a control flow edge to $instance$;
    else replace $guard$ with a control flow edge to the exit node of $loop$;
    end if;
end while;
```

Fig. 3. Generalised algorithm for guided control flow unfolding

3 Guided Control Flow Unfolding

We now describe how the derived analysis information is used to guide control flow unfolding in such a way that conditional control flow can be effectively resolved. In our previous work [4], we only considered branching or loop conditions whose condition variables could be analysed to be only defined by constant values. As a result, it was always possible to resolve conditions using our technique since knowing the constant value for each condition variable allows for inferring the value of the condition. This may not hold true if arbitrary analysis information is used instead. For instance, it is not possible to infer the value of condition `x > 10` if, e.g., an interval $(0, +\infty]$ has been derived for `x`. However, using a conservative criteria we are able to check beforehand whether the derived analysis information is sufficient to completely resolve a branching or loop condition.

In principle, unfolding a loop or branching so that conditional control flow is transformed into unconditional control flow is done using two steps. First, the loop or branching is converted into a normal form [4], which is characterised by the separation of all static paths of the control flow, ending in the respective loop header or branching node, that convey distinct values for condition variables. To this end, nodes with Φ-functions merging alternative definitions of condition variables are resolved by duplicating these nodes and their successors for each definition. Thus, after normalisation, definitions of condition variables only converge at the loop header or branching node. Second, in case of a conditional branching, splitting the branching node for each of its predecessors allows for evaluating the branching condition based on derived analysis information. According to the result, the branching is replaced with unconditional control flow leading either to the then- or to the else-part of the branching. For loops, a further step is needed to also separate the remaining paths of the control flow

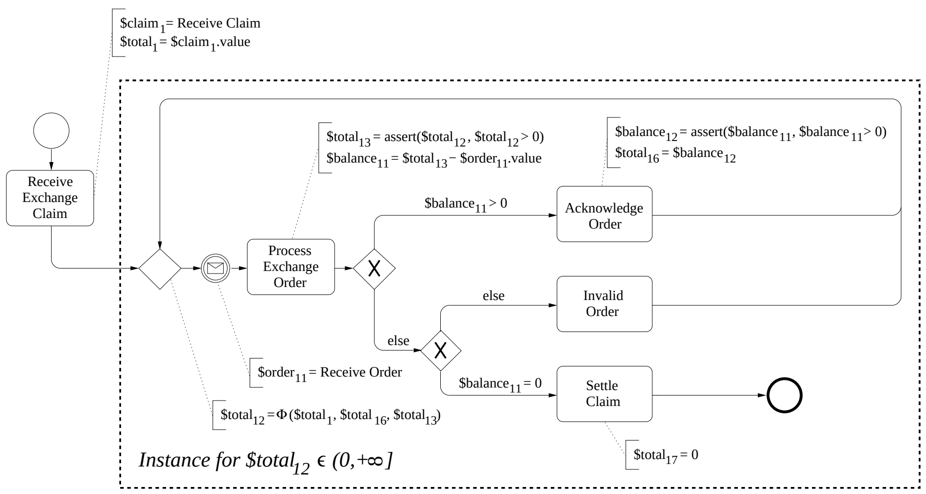
$claim_1 = Receive Claim
$total_1 = $claim_1.value
Receive Exchange Claim
$total_13 = assert($total_12, $total_12 > 0)
$balance_11 = $total_13 − $order_11.value
Process Exchange Order
$balance_12 = assert($balance_11, $balance_11 > 0)
$total_16 = $balance_12
$balance_11 > 0
Acknowledge Order
else
Invalid Order
else
$balance_11 = 0
Settle Claim
$order_11 = Receive Order
$total_12 = Φ($total_1, $total_16, $total_13)
$total_17 = 0
Instance for $total_12 ∈ (0,+∞]

Fig. 4. Unfolded workflow graph

which define distinct values for condition variables, in particular, the dynamic paths coalesced in the loop header node. For that purpose, a loop is divided into duplicates of its loop body, i.e., loop instances, where the execution of each instance is guarded by a copy of the loop condition, i.e., an instance guard. An instance thereby represents a loop iteration for a certain assertion for the values of condition variables. In our previous work [4], assertions constrained condition variables to constant values or messages. However, for a generalised unfolding, we now basically allow arbitrary assertions about variables' values. Eventually, in an iterative procedure, all instance guards are evaluated based on the derived analysis information and, like is done in case of a branching, replaced with unconditional control flow leading either to an instance or to the loop exit node.

In Figure 3, a consolidated view of unfolding a loop is given in terms of an algorithm. Therein, having derived static analysis information for a process's variables using function *analyse*, the loop is first converted into its normal form as explained in [4]. Afterwards, instance guards are iteratively processed by creating an assertion *assertion* for the values of condition variables valid at an instance guard *guard* based on the derived analysis information inf. This assertion is then used to evaluate the guard with function *evaluate* and to replace it with unconditional control flow according to the result of the evaluation.

Note that, in the algorithm, the use of analysis information is parameterised using functions *analyse* and *evaluate*. Thus, it is possible to instantiate this general algorithm for exploiting information provided by an arbitrary static analysis by merely declaring implementations for functions *analyse* and *evaluate*, with respect to the analysis. In case of our running example, function *analyse* denotes the value range analysis as described in [2]. Function *evaluate* realises the evaluation of condition expressions based on information derived by function *analyse* and can be implemented using the semantic transfer functions listed in [2].

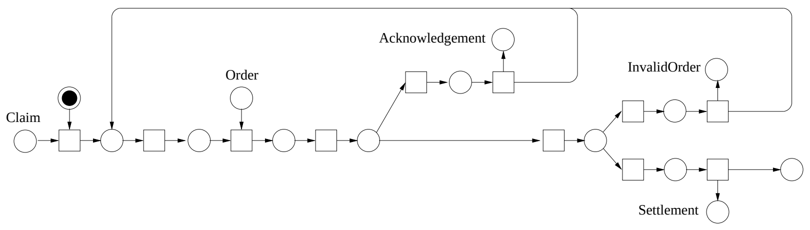

Fig. 5. Petri-net-based process model after unfolding

An application of the algorithm to the loop contained in the example process of Figure 1 allows us to exploit the value range information given in Table 1 for unfolding the loop such that the loop condition is finally resolved. The thus unfolded workflow graph is shown in Figure 4. As can be seen, it was only necessary to create a single instance for assertion $\texttt{\$total}_{12} \in (0, +\infty]$ while unfolding, where variable $\texttt{\$total}_2$ has therein been renamed to $\texttt{\$total}_{12}$.

In Figure 5, the unfolded workflow graph is mapped to a Petri net based on the Petri net semantics for workflow graphs stated in [8]. In so doing, process data, i.e., data annotations in the extended workflow graph, can be discarded without loosing precision in representing the loop's control flow since the loop condition is now properly modelled by unconditional control flow. Verifying the thus refined Petri net model with respect to controllability gives then the correct verification result for the example process, i.e., the process is controllable.

In general, our technique provides in this way a heuristics for improving verification in that the more conditional control flow is resolved, the more precise is the Petri-net-based process model and therefore the verification. Furthermore, if termination can be shown, e.g., using termination analysis [10], the verification result is guaranteed to be safe with respect to properties like controllability.

4 Related Work

Improving business process verification based on Petri nets by means of incorporating data aspects is an ongoing research topic. In [10], a termination analysis is introduced for WS-BPEL processes to help in justifying the fairness assumption. The use of high-level Petri nets is, e.g., advocated in [9] for detecting deadlocks in acyclic processes. However, the application of high-level nets in case of cyclic control flow is basically hindered by undecidability, if the domain of process data is unrestricted. This holds also true if high-level nets are unfolded into low-level Petri nets, since infinite domains then yield infinite models. In contrast, our guided unfolding approach is always guaranteed to result in finite models.

Control flow restructuring is also utilised by other program analysis and optimisation methods for improving analysis results [7]. Although, none targets the elimination of conditional control flow or value range analysis in particular. A similar static analysis for the derivation of value ranges to the one used here, which is also applied to WS-BPEL processes, has already been described in [3].

5 Conclusion and Future Work

In this paper, we presented a generalised version of our process restructuring technique [4], which allows us to unfold conditional into unconditional control flow. Compared to our previous work, the generalised technique is enabled to exploit information derived by arbitrary static analysis, as is exemplified for value range analysis, and can therefore be effectively applied in a wider range of cases. Using the technique then helps in compiling more precise process models for data-unaware Petri-net-based approaches to business process verification.

Main issue of future work will be the thorough evaluation of our process restructuring technique. Therefore, we want to implement the technique for structured business processes of the WS-BPEL language. As the value range analysis has already been realised, we currently focus on the implementation of the restructuring algorithm itself. Building on that, we further plan to employ other static analysis, i.e., symbolic methods [1], to our guided unfolding approach.

References

1. Fahringer, T., Scholz, B.: Advanced Symbolic Analysis for Compilers: New Techniques and Algorithms for Symbolic Program Analysis and Optimization. No. 2628 in LNCS, Springer (2003)
2. Gebhardt, K.: Entwurf und Implementierung einer Wertebereichsanalyse für WS-BPEL-Prozesse auf Grundlage erweiterter Workflow-Graphen. Diplomarbeit, Friedrich Schiller University of Jena (2011)
3. Görlach, K.: Ein Verfahren zur abstrakten Interpretation von XPath-Ausdrücken in WS-BPEL-Prozessen. Diplomarbeit, Humboldt University of Berlin (2008)
4. Heinze, T.S., Amme, W., Moser, S.: Process Restructuring in the Presence of Message-Dependent Variables. In: Service-Oriented Computing - ICSOC 2010 International Workshops, PAASC, WESOA, SEE, and SOC-LOG, San Francisco, CA, USA, December 7-10, 2010, Revised Selected Papers. pp. 121–132. No. 6568 in LNCS, Springer (2011)
5. Lee, J., Padua, D.A., Midkiff, S.P.: Basic Compiler Algorithms for Parallel Programs. ACM SIGPLAN Notices 34(8), 1–12 (1999)
6. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. Data & Knowledge Engineering 64(1), 38–54 (2008)
7. Steffen, B.: Property-Oriented Expansion. In: Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings. pp. 22–41. No. 1145 in LNCS, Springer (1996)
8. van der Aalst, W.M.P., Hirnschall, A., Verbeek, H.M.W.: An Alternative Way to Analyze Workflow Graphs. In: Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002, Proceedings. pp. 535–552. No. 2348 in LNCS, Springer (2002)
9. Wagner, C.: Partner Synthesis for Data-Dependent Services. In: Services and their Composition, 4th Central-European Workshop on Services and their Composition, ZEUS 2012, Bamberg, February 23-24 2012, On-Site Proceedings. pp. 17–24 (2012)
10. Weißbach, M., Zimmermann, W.: Termination analysis of business process workflows. In: Proceedings of the 5th International Workshop on Enhanced Web Service Technologies, WEWST 2010, Ayia Napa, Cyprus, December 1, 2010. pp. 18–25. ACM Press (2010)