Calculating Preconditions for Concurrent Guarded Workflows (Draft)

Thomas Göthel tbd

Abstract

We study the problem of calculating preconditions of concurrent guarded workflows in the context of information systems. It is well understood how preconditions can be calculated automatically for transactions on information system. However, until now there exists no approach for calculating preconditions of guarded workflows where subsequent actions can be arbitrarily interleaved. In this paper, we define a minimalistic workflow language for non-recursive guarded workflows and develop an algorithm that calculates the precondition of a set of concurrent such workflows. The calculated precondition ensures that each of the concurrent workflows terminates and that a given postcondition holds after all concurrent workflows have terminated. In large information systems, several users concurrently try to follow their particular workflows. If different users work on shared data, it could happen that one of the users cannot finish its workflow because of some previous action of some other user. With our approach, we are able to calculate statically a precondition for whether an additional workflow can be started without possibly blocking the already running workflows and without itself being blocked.

1 Introduction

We study the problem of formally calculating preconditions of concurrent guarded workflows. The precondition shall ensure that each of the concurrent workflows will successfully terminate, independent of the possible ordering of the (guarded) actions in their concurrent execution. As we consider workflows with explicit guards, it has to be ensured that there exists at least one terminating path through each of the considered workflows under the calculated precondition.

In [FWD07], it has been shown how the Booster language can be used to generate code from formal models in the context of information systems. A Booster model consists of classes, their attributes and methods, bidirectional associations between classes, and class invariants. Methods of classes are described using predicates relating the state before and after executing it. In information systems, methods usually do not comprise complicated algorithms but are simple operations on the data. However, these methods have to preserve several, possibly complex, system invariants during execution. These invariants can be association invariants, multiplicity invariants, symmetry invariants, and user defined invariants. Since the pre-post-condition relation representing a method is relatively simple in information systems, a candidate program can be obtained by transforming the post condition to commands. Additionally, an appropriate precondition needs to be calculated under which the execution of a command is ensured to preserve all system invariants. In [WFD05, WFD08], it has been shown how these preconditions can be calculated automatically from Booster models. The generated commands together with their calculated preconditions are integrated into guarded commands. The interpretation of guarded commands there is that a command may be executed if its precondition holds and is blocked if it does not hold.

Each method in Booster is assumed to be executed as a transaction. This means, that a method is ensured to preserve all invariants if it is not executed concurrently with other methods. However in practice, different users execute a series of transactions. To calculate the precondition of such a sequential workflow in isolation is easily possible. However, in general, workflows do not run in isolation. Actually, many workflows may be run by different users concurrently and it could happen that a user's workflow cannot be finished because another user worked on shared data making subsequent preconditions false.

In the following, we define different problem classes focussing on different aspects and assumptions of concurrent workflows.

1.1 Problems

In the following, we make the assumption that all considered workflows are finite. This means that none of them involves any sort of recursion.

1.1.1 Concurrent workflows on shared data

Preconditions of concurrent workflows need to ensure that their executions are finished independent of the ordering of their transactions. In this general case, we do not consider independence of guarded actions between concurrent workflows, i.e., all possible interleavings are considered explicitly. One simple solution is to calculate all interleaving traces of both workflows and build the conjunction of the preconditions of all these. This idea was basically followed in [Wan12], though making the calculation more efficient by concepts of dynamic programming, i.e., preconditions of common subtraces are not calculated more than once. The problem with this approach, however, is that it completely ignores guards within workflows. This makes the preconditions unnecessarily strong. Furthermore, the approach focusses on the (more efficient) calculation of preconditions for only two concurrent traces. It is not obvious how to directly extend the approach for arbitrarily many concurrent traces. We consider this problem in Section 5 where we develop an algorithm that is capable of calculating preconditions of concurrent workflows that use guards to decide which branch of a choice is to be taken. It also uses concepts of dynamic programming to calculate preconditions of common concurrent workflows only once.

1.1.2 Concurrent workflows working on separated data

Calculating preconditions of completely independent workflows is almost trivial. The precondition of a concurrent workflow $pre(W_1 \,|||\, W_2, I)$ is simply the precondition of the sequential composition $pre(W_1, pre(W_2, I))$. It does not matter which exact ordering of the workflow's actions we consider. So we can take the ordering of $W_1; W_2$ as representative. This problem becomes more complicated (and more practical) in the case that two (or more) workflows are only partly (completely) independent such that single guarded actions can be prioritised in a subsequent calculation or if there are only some guarded actions that are only locally independent. If we want to consider independent guarded actions in general, we need to be able to decide whether for two guarded actions (g, a) and (h, b) it holds that the state relations $g\&a; h\&b$ and $h\&b; g\&a$ are equal or whether one is a subset of the other. We study the problem of independence (prioritisability) of guarded actions and their utility in Section 6. The main motivation of this study is to reduce the computational effort of the general precondition calculation and to obtain simpler preconditions.

2 Related Work

An approach that is relatively close to ours is presented in [Wan12]. Like in our case, the calculation of preconditions for parallel workflows is focused on. The main difference is that [Wan12] calculates the precondition of two concurrent workflows by considering *all* pairs of traces of the one and the other workflow to calculate the respective precondition by building the conjunction from all these preconditions. This approach is sound, however, also very restrictive because the guards within the workflows are not taken into account. Another difference is that we also consider independent or prioritisable guarded actions to reduce the size of the preconditions and to reduce the amount of computations.

Despite many program logics focussing on concurrent programming logics [...], there exist no approaches that consider weakest preconditions for concurrent programs (or workflows). The advantage of weakest preconditions is that some verification engineer does not have to invent precondition-predicates and to manually show that they are strong enough to prove the necessary postcondition. Instead, our approach allows for automatically calculating preconditions for concurrent workflows that are by construction strong enough to ensure the postcondition to hold after execution.

- [Weakest Precondition Semantics for Time and Concurrency-1992]: focus on time; no focus on automatisation; focus on wp-semantics; very limited sharing of resources
- work on concurrent Hoare logic? : [Towards Concurrent Hoare Logic (MsC thesis)-2012] is probably be a good pointer to relevant related work
- Owicki/Gries?
- Have a look on [An Approach to Compositional Reasoning about Concurrent Objects and Futures-2012]
- [A Process-Algebraic Approach to Workflow Specification and Refinement-2007]: workflows patterns described in CSP; composition of workflow patterns; refinement and verification through CSP theory; no underlying data model; no precondition calculation
- [Models for Data-Flow Sequential Processes-2005]: sequential processes working on input and output streams; no precondition calculation
- [Delay-Insensitive Processes: A Formal Approach to the Design of Asynchronous Circuits, Verification and Implementation of Delay-Insensitive Processes in Restrictive Environments-2003]
- [A WEAKEST PRECONDITION SEMANTICS FOR COMMUNICATING PROCESSES-1982]: disjoint variables concurrent processes; explicit communication between processes; only rudimentary underlying data model

3 Background

3.1 Weakest Precondition Calculus

The weakest precondition [Dij76] $wp(c, I)$ of a (guarded) command c with respect to a predicate I ensures that the command successfully terminates starting from states s satisfying the precondition and that the reached states satisfy predicate I. Furthermore, this precondition is the weakest such precondition: if a condition P ensures that command c successfully

terminates reaching a state satisfying I then $P \longrightarrow wp(c, I)$. In other words, the Hoare triple [Hoa69] $\{P\}c\{Q\}$ holds *iff* $P \longrightarrow wp(c, Q)$.

Important properties of the weakest precondition calculus include the following:

- if $f \longrightarrow f'$ then $wp(c, f) \longrightarrow wp(c, f')$
- $wp(c, \bot) \longrightarrow \bot$
- $wp(c, f_1 \wedge f_2) \longleftrightarrow wp(c, f_1) \wedge wp(c, f_2)$
- $wp(c, f_1) \vee wp(c, f_2) \longrightarrow wp(c, f_1 \vee f_2)$
- if c is deterministic then $wp(c, f_1 \vee f_2) \longrightarrow wp(c, f_1) \vee wp(c, f_2)$

Especially the last property is important because it allows a case distinction on disjunctions occurring in the wp calculation. However, note that it is only valid for deterministic commands c.

An important corollary is that if $wp(a, A)$ holds for some arbitrary predicate, then also $wp(A, \top)$ holds. This follows directly from the congruence property of the wp-function w.r.t. logical implication.

4 Definition of concurrent workflows

In this section, we define a minimalistic guarded workflow language, which is still expressive enough to cope with realistic situations. We restrict ourselves to a minimalistic workflow language in order to study the essence of precondition calculation for concurrent guarded workflows. However, we show that common workflow constructions can be expressed in this language.

The most severe restrictions are that we assume that all workflows are finite, i.e., they do not contain any recursion, and that parallel workflows are only allowed on the top-level. This means that we can currently not cope with the common $AND-split$ and $AND-fork$ except for the outermost level. While the first restriction is often acceptable, we plan to get rid of the second restriction in future work.

4.1 A minimalistic workflow language

The simplest workflow is simply the terminated workflow. Furthermore, we consider indexed choices over guarded choices between two subsequent workflows. The idea is that in the later precondition calculation, that the indexed choices are interpreted as conjunction and that the guarded choices are basically interpreted as disjunction. This means that guarded choices represent the choices of some user of the workflow who only wants at least one of the choices to be enabled (to successfully finish the workflow). The indexed choices represent that each of the different paths needs to be possible. In this sense it is related to parallel composition. Therefore, a user usually doesn't want to use the $\Box^{nc}$ operator to construct a sequential workflow. Instead, a user would set I to some one-elementary set to define sequential workflows and combine these using the $|||$ operator defined below. The $\Box^{nc}$ operator is then used in order to normalise such a concurrent workflow to a sequential workflow.

Definition 1 (Sequential Workflow) *We define sequential guarded workflows according to the following grammar.*

$$\begin{array}{lcl} W & := & Skip \\ & | & \Box^{nc}_{i \in I} \, (g_{i.1} \& a_{i.1} \rightarrow W_{i.1} \, \Box \, g_{i.2} \& a_{i.2} \rightarrow W_{i.2}) \end{array}$$

We put the side condition that the index set I is non-empty.

Definition 2 (Concurrent Workflow) *To define concurrent workflows, sequential workflows can be placed in parallel on the top-level.*

$$\begin{array}{rcl} S & := & W \\ & | & W \,|||\, S \end{array}$$

4.2 Abbreviations

Although this is a very reduced workflow language, it comprises much of the workflow language as given in [Wan12].

- simple choice: $g_1 \& a_1 \to W_1 \,\Box\, g_2 \& a_2 \to W_2 := \Box^{nc}_{i \in \{\bullet\}} \, (g_1 \& a_1 \to W_1 \,\Box\, g_2 \& a_2 \to W_2)$
- stop: $Stop := \bot \& skip \to Skip \,\Box\, \bot \& skip \to Skip$
- guarded prefix: $g \& a \to W := g \& a \to W \,\Box\, \bot \& skip \to Skip$
- unguarded prefix: $a \to W := \top \& a \to W \,\Box\, \bot \& skip \to Skip$
- guarded workflow: $g \& W := g \& skip \to W \,\Box\, \bot \to Skip$
- sequential composition: $W_1; W_2 :=$ replace all $Skip$ nodes in W_1 by W_2

The main restrictions in our language are that parallelism may not be nested and that recursion is not allowed.

4.3 Normalising concurrent workflows [1]

To calculate the precondition of a sequential workflow is relatively straight-forward, our aim is to translate concurrent workflows to sequential ones. In the subsequently presented algorithm for the precondition calculation of concurrent workflows, we take advantage of this normalisation procedure to transform and calculate the precondition on the fly.

At first we present the normalisation of the binary case for the sake of a better comprehensibility before we consider the general case.

4.3.1 The binary case

The basic idea of our normalisation procedure is to represent possible interleavings of concurrent workflows in terms of indexed choices ($\Box^{nc}$) and to preserve the guarded choices ($\Box$) of all involved sequential workflows.

$$\begin{array}{lcl} norm(Skip \,|||\, W) & = & W \\ norm(W \,|||\, Skip) & = & W \end{array}$$

$$\begin{array}{lcl} norm(\Box^{nc}_{i \in I} \, (g_{i.1} \& a_{i.1} \to W_{i.1} \,\Box\, g_{i.2} \& a_{i.2} \to W_{i.2}) & & \\ \quad ||| \, \Box^{nc}_{j \in J} \, (g_{j.1} \& a_{j.1} \to W_{j.1} \,\Box\, g_{j.2} \& a_{j.2} \to W_{j.2})) & = & \Box^{nc}_{i \in I \uplus J} \, (g_{i.1} \& a_{i.1} \to norm(P_{I,J,i.1}) \\ & & \quad \Box\, g_{i.2} \& a_{i.2} \to norm(P_{I,J,i.2})) \end{array}$$

case: $\underline{i \in I}$

[1] When I started thinking about precondition calculation of concurrent workflows this was kind of a starting point. In the meantime, I think the value of normalising concurrent workflows is only very limited. The only reason to still consider this is that we could base correctness issues on normalised workflows: $pre(|||_i W_i, I) = pre(norm(|||_i W_i), I)$ and if $s \models pre(W, I)$ where W is some sequential workflow then for all paths $(s, W) \longrightarrow^* (s', W')$ we either have that (s', W') can somehow proceed (no deadlock) or $W' = Skip$ and $s' \models I$.

$P_{I,J,i.m} = W_{i.m} \,|||\, \Box^{nc}_{n\in J}\ (g_{n.1}\&a_{n.1} \to W_{n.1} \,\Box\, g_{n.2}\&a_{n.2} \to W_{n.2})$

case: $i \in J$

$P_{I,J,i.m} = \Box^{nc}_{n\in I}\ (g_{n.1}\&a_{n.1} \to W_{n.1} \,\Box\, g_{n.2}\&a_{n.2} \to W_{n.2}) \,|||\, W_{i.m}$

4.3.2 The general case

The binary case can be generalized to the general case where we normalise arbitrarily many concurrent workflows.

$$
\begin{array}{lcl}
norm(|||_{j\in J} W_j) & = & norm(|||_{j\in J\setminus\{i\}} W_j) \quad \textbf{if } W_i = Skip \\
norm(|||_{j\in J}(\Box^{nc}_{i\in I_j}(g_{i.1}\&a_{i.1} \to W_{i.1} \,\Box & & \\
\qquad g_{i.2}\&a_{i.2} \to W_{i.2}))) & = & \\
\Box^{nc}_{i\in \biguplus_{I_j}}(g_{i.1}\&a_{i.1} \to norm(|||_{j\in J} \textbf{ if } i \in I_j \textbf{ then } W_{i.1} & & \\
\qquad \textbf{else } \Box^{nc}_{x\in I_j}(g_{x.1}\&a_{x.1} \to W_{x.1} \,\Box\, g_{x.2}\&a_{x.2} \to W_{x.2})) & & \\
\quad \Box\, g_{i.2}\&a_{i.2} \to norm(|||_{j\in J} \textbf{ if } i \in I_j \textbf{ then } W_{i.2} & & \\
\qquad \textbf{else } \Box^{nc}_{x\in I_j}(g_{x.1}\&a_{x.1} \to W_{x.1} \,\Box\, g_{x.2}\&a_{x.2} \to W_{x.2}))) & &
\end{array}
$$

5 Calculating preconditions of concurrent workflows

We focus on the problem of calculating preconditions for concurrent workflows that ensure that each of the sequential workflows is ensured to terminate successfully. In contrast to [Wan12], we explicitly consider guards in workflows. This means that we do not insist that every possible execution of a workflow must be possible but that there exists at least one path through each workflow.

5.1 Precondition calculation of sequential workflows

Since we can normalise concurrent workflows in our language to sequential ones, we only need to consider the precondition calculation for sequential ones. This calculation takes exactly into account, which choices are to be considered as guarded choices and which choices belong indexed choices (original interleavings).

$$
\begin{array}{lcl}
pre(Skip, I) & = & I \\
pre(\Box^{nc}_{i\in I}\ (g_{i.1}\&a_{i.1} \to W_{i.1} \,\Box\, g_{i.2}\&a_{i.2} \to W_{i.2}), I) & = & \bigwedge_{i\in I}(g_{i.1} \longrightarrow wp(a_{i.1}, pre(W_{i.1}, I)) \\
 & & \qquad \wedge \\
 & & \qquad g_{i.2} \longrightarrow wp(a_{i.2}, pre(W_{i.2}, I)) \\
 & & \qquad \wedge \\
 & & \qquad (g_{i.1} \vee g_{i.2}))
\end{array}
$$

The conditions occurring above are equivalent to

$$
\begin{array}{l}
\quad g_{i.1} \wedge \neg g_{i.2} \wedge wp(a_{i.1}, pre(W_{i.1}, I)) \\
\vee\ \neg g_{i.1} \wedge g_{i.2} \wedge wp(a_{i.2}, pre(W_{i.2}, I)) \\
\vee\ g_{i.1} \wedge g_{i.2} \wedge wp(a_{i.1}, pre(W_{i.1}, I)) \wedge wp(a_{i.2}, pre(W_{i.2}, I))
\end{array}
$$

This means that we calculate the precondition of only one subsequent path if only $g_{i.1}$ or only $g_{i.2}$ respectively holds. In the case that the guards overlap, i.e., the user has both choices, we need to ensure that both paths terminate and finally reach a state satisfying I.

5.2 Example

As an abstract example consider the following concurrent workflow with g_1, g_2 non-overlapping and h_1, h_2 non-overlapping.

$$g_1 \& a_1 \to Skip \,\Box\, g_2 \& a_2 \to Skip$$
$$|||\, h_1 \& b_1 \to Skip \,\Box\, h_2 \& b_2 \to Skip$$

First, we normalise this process, which results in the following (sequential) workflow.

$$\begin{array}{l} (g_1 \& a_1 \to (h_1 \& b_1 \to Skip \,\Box\, h_2 \& b_2 \to Skip) \\ \quad \Box\, g_2 \& a_2 \to (h_1 \& b_1 \to Skip \,\Box\, h_2 \& b_2 \to Skip)) \\ \Box^{nc} \\ (h_1 \& b_1 \to (g_1 \& a_1 \to Skip \,\Box\, g_2 \& a_2 \to Skip) \\ \quad \Box\, h_2 \& b_2 \to (g_1 \& a_1 \to Skip \,\Box\, g_2 \& a_2 \to Skip)) \end{array}$$

Second, we calculate the precondition of this sequential workflow, which is

$$\begin{array}{l} (g_1 \wedge wp(a_1, h_1 \wedge wp(b_1, I) \\ \qquad\qquad\qquad \vee \\ \qquad\qquad\qquad h_2 \wedge wp(b_2, I)) \\ \quad \vee \\ \quad g_2 \wedge wp(a_2, h_1 \wedge wp(b_1, I) \\ \qquad\qquad\qquad \vee \\ \qquad\qquad\qquad h_2 \wedge wp(b_2, I))) \\ \wedge \\ (h_1 \wedge wp(b_1, g_1 \wedge wp(a_1, I) \\ \qquad\qquad\qquad \vee \\ \qquad\qquad\qquad g_2 \wedge wp(a_2, I)) \\ \quad \vee \\ \quad h_2 \wedge wp(b_2, g_1 \wedge wp(a_1, I) \\ \qquad\qquad\qquad \vee \\ \qquad\qquad\qquad g_2 \wedge wp(a_2, I))). \end{array}$$

Using the approach given in [Wan12], the result would be the following.

$$\begin{array}{l} wp(a_1, wp(b_1, I)) \wedge wp(a_1, wp(b_2, I)) \\ \wedge wp(a_2, wp(b_1, I)) \wedge wp(a_2, wp(b_2, I)) \\ \wedge wp(b_1, wp(a_1, I)) \wedge wp(b_1, wp(a_2, I)) \\ \wedge wp(b_2, wp(a_1, I)) \wedge wp(b_2, wp(a_2, I)). \end{array}$$

Assume, for example, that after executing a_1 the precondition of b_1 does not hold, i.e., $wp(a_1, wp(b_1, I)) = \bot$. Furthermore assume that $wp(a_1, wp(b_2, I))$ was satisfiable. Then the approach of [Wan12] would consider both workflows not to be executable concurrently. However, when explicitly considering guards, our tree-based calculation (that respects guards) would give a satisfiable precondition (if at least one of the orderings $b_1; a_1$, $b_1; a_2$, $b_2; a_1$, $b_2; a_2$ had a precondition in the intersection).

5.3 An "efficient" algorithm for calculating preconditions of concurrent workflows

The procedure of first normalising the concurrent workflow and second calculating the precondition of the normalised workflow is not very efficient because of two reasons. The first reason is that the normalised version of a concurrent workflow usually gets very big. The second reason is that preconditions of syntactically equal concurrent workflows is calculated more than once. Assume that in a concurrent workflow the i^{th} sequential workflow can perform an a and that the j^{th} workflow can perform a b. Then we perform the precondition calculation (we omit the guards here) $wp(a, wp(b, C'))$ and $wp(b, wp(a, C''))$. Note however that $C' = C''$, i.e., the same subsequent concurrent workflow is reached regardless whether

```
-- We don't have to calculate anything for the empty composition.
  pre(|||_{i∈{}} _, lut) = lut
-- delete all top-level "Skip"s
  pre(|||_{i∈I} W_i, lut) = pre(|||_{i∈(I\{j})} W_i)  |  if W_j = Skip
-- the general case:
-- The algorithm is given informally. However, it can easily be seen that the three
-- parts with ... can easily be implemented using some iteration over I.
-- We assume that I={1 .. n} for some arbitrary n to make the algorithm more readable.
  pre(pW ≡ |||_{i∈I} W_i
         ≡ |||_{i∈I} (g_{i.1}&a_{i.1} → W_{i.1}
                    □ g_{i.2}&a_{i.2} → W_{i.2}, lut) =
    if pW def? lut then lut
    else let lut1   = pre(|||_{i∈I} i = 1?W_{i.1} : W_i, lut)
             lut1'  = pre(|||_{i∈I} i = 1?W_{i.2} : W_i, lut1)
             ...
             lut#I  = pre(|||_{i∈I} i = #I?W_{i.1} : W_i, lut(#I-1)')
             lut#I' = pre(|||_{i∈I} i = #I?W_{i.1} : W_i, lut#I)

             p11 = g_{1.1} --> wp(a_{1.1}, lut#I'!(|||_{i∈I} i = 1?W_{i.1} : W_i))
             p12 = g_{1.2} --> wp(a_{1.2}, lut#I'!(|||_{i∈I} i = 1?W_{i.2} : W_i))
             p13 = (g_{1.1} \/ g_{1.2})
             ...
             p#I1 = g_{#I.1} --> wp(a_{#I.1}, lut#I'!(|||_{i∈I} i = #I?W_{i.1} : W_i))
             p#I2 = g_{#I.2} --> wp(a_{#I.2}, lut#I'!(|||_{i∈I} i = #I?W_{i.2} : W_i))
             p#I3 = (g_{#I.1} \/ g_{#I.2})
         in
             def(lut#I', pW := (p11  /\ p12  /\ p13) /\
                                 ...              /\
                               (p#I1 /\ p#I2 /\ p#I3) )
```

Figure 1: Algorithm for calculating preconditions of concurrent guarded workflows

$\langle a, b\rangle$ or $\langle b, a\rangle$ is performed. That the already calculated precondition on, say, the $\langle a, b\rangle$ path can be reused on the other path.

For these two reasons, we perform the normalisation of some concurrent workflow in our algorithm on the fly and directly calculate the formula representing the overall precondition. Furthermore, we keep track of the already visited concurrent workflows in a map to calculate preconditions only once. We assume that the single sequential workflows with the concurrent workflow do not use the $\Box^{nc}$ operator in the sense that the corresponding index set I is always a one-element-set.

Our algorithm is shown in Figure 1. To calculate the precondition of a concurrent workflow $|||_{i\in I} W_i$ with respect to some post condition I, $pre(|||_{i\in I} W_i, \{|||_{i\in\{\}} \_ \mapsto I\})!(|||_{i\in I} W_i)$ has to be invoked.

We have prototypically implemented this algorithm in Haskell[2] as a proof of concept. As expected the calculation of preconditions of several concurrent workflows is effectively possible and the corresponding preconditions get huge. In the next section, we show how the presented algorithm can be extended to consider independent guarded actions between concurrent workflows. By this, we can sometimes considerably reduce the visited state space and we can very often get simpler (smaller) formulas as preconditions.

[2]The original algorithm is realised in Precond.hs. Currently, it is not running because I changed some basic representations. However, Precond3.hs, Precond4.hs, and Precond5.hs do basically the same if none of the guarded actions is declared as independent to some other guarded action.

5.4 Correctness

Given a parallel Workflow $W := |||_{i \in I} W_i$ and a postcondition I. We need to show that $pre(norm(W), I) = pre(W, \{|||_{i \in \{\}^-} \mapsto I\})!W$ (under the current assumption that none of the W_is contains the $\Box^{nc}$ operator).

6 Preconditions of concurrent workflows with independent (prioritisable) guarded actions

In this section, we consider prioritisable guarded actions within concurrent workflows. Our aim is to thereby reduce the computational effort in our precondition calculation as given in the last section. The main idea is to leave out certain precondition paths when it basically makes no difference which ordering of two (or more) actions is considered.

6.1 Prioritisable guarded actions

In the most general case (possibly overlapping guards), we have to make sure that the subsequent weakest precondition calculation for one ordering of the guarded action pairs already contains the precondition of the other ordering. This means that we have basically to consider all possibilities of subsequent guards to be true or not resulting in 16 cases. We will, however, discuss below how these conditions can be simplified by strengthening them or assuming that the guards in a guarded action pair are non-overlapping.

Definition 3 (Prioritisable Guarded Actions) *A guarded-action-pair $((g_1, a_1), (g_2, a_2))$, where a_1 and a_2 are deterministic, is prioritisable over $((h_1, b_1), (h_2, b_2))$ if the following conditions hold for all predicates I:*

- $g_1 \wedge \neg g_2 \wedge wp(a_1, h_1 \wedge \neg h_2 \wedge wp(b_1, I)) \longrightarrow h_1 \wedge \neg h_2 \wedge wp(b_1, g_1 \wedge \neg g_2 \wedge wp(a_1, I))$
- $g_1 \wedge \neg g_2 \wedge wp(a_1, \neg h_1 \wedge h_2 \wedge wp(b_2, I)) \longrightarrow \neg h_1 \wedge h_2 \wedge wp(b_2, g_1 \wedge \neg g_2 \wedge wp(a_1, I))$
- $g_1 \wedge \neg g_2 \wedge wp(a_1, h_1 \wedge h_2 \wedge wp(b_1, I)) \longrightarrow h_1 \wedge h_2 \wedge wp(b_1, g_1 \wedge \neg g_2 \wedge wp(a_1, I))$
- $g_1 \wedge \neg g_2 \wedge wp(a_1, h_1 \wedge h_2 \wedge wp(b_2, I)) \longrightarrow h_1 \wedge h_2 \wedge wp(b_2, g_1 \wedge \neg g_2 \wedge wp(a_1, I))$

- $\neg g_1 \wedge g_2 \wedge wp(a_2, h_1 \wedge \neg h_2 \wedge wp(b_1, I)) \longrightarrow h_1 \wedge \neg h_2 \wedge wp(b_1, \neg g_1 \wedge g_2 \wedge wp(a_2, I))$
- $\neg g_1 \wedge g_2 \wedge wp(a_2, \neg h_1 \wedge h_2 \wedge wp(b_2, I)) \longrightarrow \neg h_1 \wedge h_2 \wedge wp(b_2, \neg g_1 \wedge g_2 \wedge wp(a_2, I))$
- $\neg g_1 \wedge g_2 \wedge wp(a_2, h_1 \wedge h_2 \wedge wp(b_1, I)) \longrightarrow h_1 \wedge h_2 \wedge wp(b_1, \neg g_1 \wedge g_2 \wedge wp(a_2, I))$
- $\neg g_1 \wedge g_2 \wedge wp(a_2, h_1 \wedge h_2 \wedge wp(b_2, I)) \longrightarrow h_1 \wedge h_2 \wedge wp(b_2, \neg g_1 \wedge g_2 \wedge wp(a_2, I))$

- $g_1 \wedge g_2 \wedge wp(a_1, h_1 \wedge \neg h_2 \wedge wp(b_1, I)) \longrightarrow h_1 \wedge \neg h_2 \wedge wp(b_1, g_1 \wedge g_2 \wedge wp(a_1, I))$
- $g_1 \wedge g_2 \wedge wp(a_1, \neg h_1 \wedge h_2 \wedge wp(b_2, I)) \longrightarrow \neg h_1 \wedge h_2 \wedge wp(b_2, g_1 \wedge g_2 \wedge wp(a_1, I))$
- $g_1 \wedge g_2 \wedge wp(a_1, h_1 \wedge h_2 \wedge wp(b_1, I)) \longrightarrow h_1 \wedge h_2 \wedge wp(b_1, g_1 \wedge g_2 \wedge wp(a_1, I))$
- $g_1 \wedge g_2 \wedge wp(a_1, h_1 \wedge h_2 \wedge wp(b_2, I)) \longrightarrow h_1 \wedge h_2 \wedge wp(b_2, g_1 \wedge g_2 \wedge wp(a_1, I))$

- $g_1 \wedge g_2 \wedge wp(a_2, h_1 \wedge \neg h_2 \wedge wp(b_1, I)) \longrightarrow h_1 \wedge \neg h_2 \wedge wp(b_1, g_1 \wedge g_2 \wedge wp(a_2, I))$
- $g_1 \wedge g_2 \wedge wp(a_2, \neg h_1 \wedge h_2 \wedge wp(b_2, I)) \longrightarrow \neg h_1 \wedge h_2 \wedge wp(b_2, g_1 \wedge g_2 \wedge wp(a_2, I))$
- $g_1 \wedge g_2 \wedge wp(a_2, h_1 \wedge h_2 \wedge wp(b_1, I)) \longrightarrow h_1 \wedge h_2 \wedge wp(b_1, g_1 \wedge g_2 \wedge wp(a_2, I))$
- $g_1 \wedge g_2 \wedge wp(a_2, h_1 \wedge h_2 \wedge wp(b_2, I)) \longrightarrow h_1 \wedge h_2 \wedge wp(b_2, g_1 \wedge g_2 \wedge wp(a_2, I))$

These conditions are difficult to check in general. This is mainly because the conditions have to be checked for arbitrary predicates I. To reduce the quantified conditions we define another slightly stronger condition:

Definition 4 (Strongly Prioritisable Guarded Actions) *A guarded actions (g, a) is strongly prioritisable over (h, b) if the following conditions hold:*

- $\forall I.wp(a, wp(b, I)) \longrightarrow wp(b, wp(a, I))$
- $wp(a, h) \longrightarrow h$
- $wp(a, \neg h) \longrightarrow \neg h$
- $g \wedge wp(b, T) \longrightarrow wp(b, g)$
- $\neg g \wedge wp(b, T) \longrightarrow wp(b, \neg g)$

Note that we in the last two conditions, e.g., $g \longrightarrow wp(b, g)$ would be too strong because it would mean that g also would need to contain the termination condition of action b.

It can easily be shown that if (g_1, a_1) and (g_2, a_2) are strongly prioritisable over each of (h_1, b_1) and (h_2, b_2) then $((g_1, a_1), (g_2, a_2))$ is prioritisable over $((h_1, b_1)(h_2, b_2))$. By this, we have in total 20 checks, however only 4 quantifying over arbitrary predicates I. Thus, 16 of the conditions can be checked fully automatically.

We can simplify the checks above if we assume that g_1, g_2 and that h_1, h_2 are non-overlapping, i.e., $g_1 \wedge g_2 \longleftrightarrow \bot$ and $h_1 \wedge h_2 \longleftrightarrow \bot$.

Definition 5 (Weakly Prioritisable Guarded Actions) *A guarded action (g, a) is weakly prioritisable over (h, b) if $g \wedge wp(a, h \wedge wp(b, I)) \longrightarrow h \wedge wp(b, g \wedge wp(a, I))$ for all predicates I.*

If g_1 and g_2 are non-overlapping and h_1 and h_2 are non-overlapping then $((g_1, a_1), (g_2, a_2))$ is prioritisable over $((h_1, b_1), (h_2, b_2))$ *iff* $(g_1, a_1), (g_2, a_2)$ are mutually weakly prioritisable over $(h_1, b_1), (h_2, b_2)$.

In the next section, we apply these definitions in the context of our precondition calculation.

6.2 Locally prioritisable guarded actions

In the calculation of concurrent workflows, local prioritisability of guarded actions can be exploited to get simpler preconditions. Consider the following concurrent guarded workflow with $((g_1, a_1), (g_2, a_2))$ prioritisable over $((h_1, b_1), (h_2, b_2))$.
$LEFT \,|||\, g_1 \& a_1 \rightarrow Wf_{11} \,\Box\, g_2 \& a_2 \rightarrow Wf_{12} \,|||\, MIDDLE \,|||\, h_1 \& b_1 \rightarrow Wf_{21} \,\Box\, h_2 \& b_2 \rightarrow Wf_{22} \,|||\, RIGHT$

Using our algorithm, we get the following precondition for this concurrent workflow.
$g_1 \longrightarrow wp(a_1, h_1 \longrightarrow wp(b_1, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT))$
$\qquad\qquad \wedge h_2 \longrightarrow wp(b_2, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT))$

$$
\begin{array}{l}
\qquad\qquad \wedge(h_1 \vee h_2) \\
\qquad\qquad \wedge A_1) \\
\wedge \\
g_2 \longrightarrow wp(a_2, h_1 \longrightarrow wp(b_1, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT)) \\
\qquad\qquad \wedge h_2 \longrightarrow wp(b_2, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT)) \\
\qquad\qquad \wedge(h_1 \vee h_2) \\
\qquad\qquad \wedge A_2)) \\
\wedge \\
(g_1 \vee g_2) \\
\wedge \\
h_1 \longrightarrow wp(b_1, g_1 \longrightarrow wp(a_1, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT)) \\
\qquad\qquad \wedge g_2 \longrightarrow wp(a_2, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT)) \\
\qquad\qquad \wedge(g_1 \vee g_2) \\
\qquad\qquad \wedge B_1) \\
\wedge \\
h_2 \longrightarrow wp(b_2, g_1 \longrightarrow wp(a_1, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT)) \\
\qquad\qquad \wedge g_2 \longrightarrow wp(a_2, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT)) \\
\qquad\qquad \wedge(g_1 \vee g_2) \\
\qquad\qquad \wedge B_2)) \\
\wedge \\
(h_1 \vee h_2) \\
\wedge R
\end{array}
$$

The condition A_1, for example, stands for the preconditions that actions from $LEFT$, Wf_{11}, $MIDDLE$, or $RIGHT$ are chosen after the action a_1 (and b_1, b_2 are chosen subsequently). The condition R stands for the possible preconditions of paths beginning with some other events apart from a and b events.

From the condition

$$
\begin{array}{l}
g_1 \longrightarrow wp(a_1, h_1 \longrightarrow wp(b_1, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT)) \\
\qquad\qquad \wedge h_2 \longrightarrow wp(b_2, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT)) \\
\qquad\qquad \wedge(h_1 \vee h_2)) \\
\wedge \\
g_2 \longrightarrow wp(a_2, h_1 \longrightarrow wp(b_1, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT)) \\
\qquad\qquad \wedge h_2 \longrightarrow wp(b_2, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT)) \\
\qquad\qquad \wedge(h_1 \vee h_2)) \\
\wedge \\
(g_1 \vee g_2)
\end{array}
$$

we can conclude that

$$
\begin{array}{l}
h_1 \longrightarrow wp(b_1, g_1 \longrightarrow wp(a_1, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT)) \\
\qquad\qquad \wedge g_2 \longrightarrow wp(a_2, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT)) \\
\qquad\qquad \wedge(g_1 \vee g_2)) \\
\wedge \\
h_2 \longrightarrow wp(b_2, g_1 \longrightarrow wp(a_1, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT)) \\
\qquad\qquad \wedge g_2 \longrightarrow wp(a_2, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT)) \\
\qquad\qquad \wedge(g_1 \vee g_2)) \\
\wedge \\
(h_1 \vee h_2)
\end{array}
$$

holds. To enable this reasoning we need the prioritisability as above and especially that a_1, a_2 are deterministic. The latter is necessary to enable the disjunction property of the wp-calculus: $wp(a, X \vee Y) \longrightarrow wp(a, X) \vee wp(a, Y)$, which is only valid for deterministic commands a. We use this property in order to make case distinction in, for example,

$wp(a_1, h_1 \vee h_2)$. Otherwise, we would not be able to match the premises of the prioritisability clauses (see Definition 3).
This means that we can simplify the original precondition to the following one.

$$
\begin{array}{l}
g_1 \longrightarrow wp(a_1, h_1 \longrightarrow wp(b_1, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT)) \\
\qquad\qquad \wedge h_2 \longrightarrow wp(b_2, pre(LEFT \,|||\, Wf_{11} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT)) \\
\qquad\qquad \wedge (h_1 \vee h_2) \\
\qquad\qquad \wedge A_1) \\
\wedge \\
g_2 \longrightarrow wp(a_2, h_1 \longrightarrow wp(b_1, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{21} \,|||\, RIGHT)) \\
\qquad\qquad \wedge h_2 \longrightarrow wp(b_2, pre(LEFT \,|||\, Wf_{12} \,|||\, MIDDLE \,|||\, Wf_{22} \,|||\, RIGHT)) \\
\qquad\qquad \wedge (h_1 \vee h_2) \\
\qquad\qquad \wedge A_2)) \\
\wedge \\
(g_1 \vee g_2) \\
\wedge \\
h_1 \longrightarrow wp(b_1, B_1) \\
\wedge \\
h_2 \longrightarrow wp(b_2, B_2)) \\
\wedge R
\end{array}
$$

If B_1 or B_2 is always true (no subsequent actions apart from a_i possible), then the condition $h_1 \longrightarrow wp(b_1, B_1)$ or $h_2 \longrightarrow wp(b_2, B_2))$ respectively can be omitted because from, e.g., $h_1 \longrightarrow wp(b_1, g_1 \longrightarrow \dots)$ we can infer that also $h_1 \longrightarrow wp(b_1, \top)$ holds (see Section 3.1).

The detailed proofs of all claims above are basically tedious case distinctions, which we have performed in the Isabelle/HOL theorem prover [NPW02].

6.3 Globally prioritisable guarded actions

The approach of reducing preconditions due to locally prioritisable guarded actions can even be used to reduce the visited state space within our algorithm if we assume globally prioritisable guarded actions. Consider the following concurrent workflow.
$Wf_1 \,|||\, \cdots \,|||\, g_{i.1}\&a_{i.1} \rightarrow Wf_{i.1} \,\Box\, g_{i.2}\&a_{i.2} \rightarrow Wf_{i.2} \,|||\, \cdots \,|||\, Wf_n$.
If $(g_{i.1}, a_{i.1}), (g_{i.2}, a_{i.2})$ are prioritisable over *all* (also subsequent) guarded action pairs of the other workflows, the precondition can be reduced to the following.

$$
\begin{array}{l}
g_{i.1} \longrightarrow wp(a_{i.1}, pre(W_1 \,|||\, \cdots \,|||\, Wf_{i.1} \,|||\, \cdots \,|||\, Wf_n)) \\
\vee \\
g_{i.2} \longrightarrow wp(a_{2.1}, pre(W_1 \,|||\, \cdots \,|||\, Wf_{i.2} \,|||\, \cdots \,|||\, Wf_n)) \\
\vee \\
(g_{i.1} \vee g_{i.2})
\end{array}
$$

It is not necessary to consider, for example, a $\langle b, c, a_{i.1} \rangle$ path because $a_{i.1}$ can first be prioritised over c and then over b. This argument extends to arbitrary depths because in each case the guarded a actions can be prioritised until we reach the top-level.

6.4 Extending the previous algorithm

In the last section, we have shown how prioritisable guarded actions can be exploited to reduce the precondition formula of some concurrent workflow and how in some cases the visited state space can be reduced in our precondition calculation. In this section, we give some details how we actually implemented these results.

6.4.1 Locally prioritisable guarded actions

In the calculation of pre($W_1 \,|||\, W_2 \,|||\, \cdots \,|||\, W_n$) within our algorithm, we exploit the results w.r.t. locally independent actions in the following way. At first, the precondition calculation w.r.t. the initial actions of W_1 is identical to the original calculation. However, when we consider W_2, we check whether the initial guarded action pair of W_1 is prioritisable over the initial guarded action pair of W_2. If this holds, we mark the initial actions of W_1 as delayed because the previously calculated precondition (where we consider the initial actions of $W1$) already contain the precondition of first performing the initial actions of W_2 and then W_1. Delayed workflows are not considered in the current step but (possibly) again subsequently.

Within our calculation, we have the following.

$g_{2.1} \wedge wp(a_{2.1}, pre(\overline{W_1} \,|||\, W_{2.1} \,|||\, \cdots \,|||\, W_n))$
$\vee$
$g_{2.2} \wedge wp(a_{2.2}, pre(\overline{W_1} \,|||\, W_{2.2} \,|||\, \cdots \,|||\, W_n))$

In the subsequent calculation of, for example, $pre(\overline{W_1} \,|||\, W_{2.1} \,|||\, \cdots \,|||\, W_n)$, the initial actions of W_1 are not considered but are "undelayed" when considering initial actions of $W_{2.1}, ..., W_n$ (unless the initial guarded action pair of W_1 is again prioritisable over the currently considered guarded action pair of some other workflows). This means that locally prioritisable guarded actions are delayed as long as they are prioritisable over the considered guarded guarded action pair.

When we consider the initial guarded action pair of W_3, we check for prioritisability of the initial guarded action pairs of W_1 and W_2 and delay them accordingly if possible. And so on.

In general, when considering the initial actions of some workflow W_i, we temporarily delay workflow W_j with $j < i$ if the guarded action pair of W_j is prioritisable over the initial guarded action pair of W_i. By this, we ensure that the precondition calculation that has been performed for the "left" workflows is "more complete".

Consider the following illustrating example where $((g, a), (\bot, skip))$ is prioritisable over $(h_1, b_1), (h_2, b_2)$ and where h_1, h_2 do not overlap.

$pre(g\&a \rightarrow Skip \,|||\, h_1\&b_1 \rightarrow Skip \,\Box\, h_2\&b_2 \rightarrow (h_3\&b_3 \rightarrow Skip))$
$= g \wedge wp(a, pre(h_1\&b_1 \rightarrow Skip \,\Box\, h_2\&b_2 \rightarrow (h_3\&b_3 \rightarrow Skip)))$
$\wedge (h_1 \wedge wp(b_1, pre(\overline{g\&a \rightarrow Skip}))$
$\quad \vee h_2 \wedge wp(b_2, pre(\overline{g\&a \rightarrow Skip} \,|||\, h_3\&b_3 \rightarrow Skip)))$
$= g \wedge wp(a, pre(h_1\&b_1 \rightarrow Skip \,\Box\, h_2\&b_2 \rightarrow (h_3\&b_3 \rightarrow Skip)))$
$\wedge (h_1 \wedge wp(b_1, \top)$
$\quad \vee h_2 \wedge wp(b_2, h_3 \wedge wp(b_3, pre(g\&a \rightarrow Skip)))$
$= g \wedge wp(a, pre(h_1\&b_1 \rightarrow Skip \,\Box\, h_2\&b_2 \rightarrow (h_3\&b_3 \rightarrow Skip)))$
$\wedge (h_1$
$\quad \vee h_2 \wedge wp(b_2, h_3 \wedge wp(b_3, pre(g\&a \rightarrow Skip)))$

Note that the last equality is sound because from the g-conjunct we already know that $h_1 \longrightarrow wp(b_1, T)$ holds (see Section 6.2).

6.4.2 Globally prioritisable guarded actions

Within our algorithm, we check whether the initial guarded-action-pair of some workflow in the composition is independent from all (also subsequent) guarded actions of all other workflows. If we find such a workflow, we prioritise its initial guarded actions as above.

6.4.3 Integrating locally and globally prioritisable guarded actions

We have integrated these procedures w.r.t. locally and globally prioritisable guarded actions in our algorithm. In the case of globally prioritisable guarded actions, we only search for undelayed guarded action pairs that are globally prioritisable and have to ensure that they are also prioritisable over currently delayed guarded action pairs because they could be activated later on. After globally prioritising a guarded action pair, all currently delayed guarded actions are undelayed for the subsequent calculation (where they could be delayed again).

7 Case Study

tbd

8 Discussion/Future Work

In this paper, we have developed an algorithm that calculates the precondition of concurrent guarded workflows. The precondition ensures that each of the sequential workflows is ensured to successfully terminate and that the overall postcondition is reached. In contrast to related work, we explicitly consider guards in order to obtain preconditions that are not too strong. Our algorithm is "efficient" in that it employs concepts of dynamic programming such that preconditions for the same subsequent workflows are only calculated once. Do further reduce the computational effort and to reduce the size of formulas representing preconditions, we have studied how prioritisable guarded actions can be used to do so. All theoretical contributions have been verified using the Isabelle/HOL theorem prover. We have furthermore implemented the algorithm to enable evaluation of our approach. We have considered the case study ... tbd

In future work, we would like to broaden our approach towards the precondition calculation of more sophisticated workflows. This includes that we want to consider nested parallelism, recursive workflows, and synchronization between workflows. The motivation for the latter is that a user of some workflow could be willing to wait for the result of another workflow, which he possibly even initiated.

Another direction for future work could be to look more on input-dependent workflows. This means that control flow is influenced deeper by the input of data that the user enters during executing a particular workflow.

The presented calculation of preconditions for many concurrent workflows is rather complex. The reason for this is the lack of compositionality. This means that we currently cannot infer the precondition of some concurrent workflow $W_1 \mid\mid\mid W_2$ from the precondition of W_1 and the precondition of W_2. A study on compositionality of workflows w.r.t. precondition calculation would therefore be very advantageous. Especially, if this calculation was based on a compositional denotational semantics, we could define a notion of refinement on workflows.

References

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[FWD07] David Faitelson, James Welch, and Jim Davies. From predicates to programs: The semantics of a method language. *Electr. Notes Theor. Comput. Sci.*, 184:171–187, 2007.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Wan12] Chen-Wei Wang. Calculating preconditions for parallel workflows. In *APSEC*, pages 499–504, 2012.

[WFD05] James Welch, David Faitelson, and Jim Davies. Automatic maintenance of association invariants. In *SEFM*, pages 282–292, 2005.

[WFD08] James Welch, David Faitelson, and Jim Davies. Automatic maintenance of association invariants. *Software and System Modeling*, 7(3):287–301, 2008.