Priority-Based Compilation

Project Overview

Responsible:

- Caroline Butschek, Christian Motika, Steven Smyth

Related Theses:

- not finished yet

Related Papers:

- Reinhard von Hanxleden and Björn Duderstadt and Christian Motika and Steven Smyth and Michael Mendler and Joaquín Aguado and Stephen Mercer and Owen O'Brien. SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14), Edinburgh, UK, June 2014. ACM. (pdf)
- Reinhard von Hanxleden and Michael Mendler and Joaquín Aguado and Björn Duderstadt and Insa Fuhrmann and Christian Motika and Stephen Mercer and Owen O'Brien. Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation. In Proc. Design, Automation and Test in Europe Conference (DATE'13), page 581–586, Grenoble, France, March 2013. IEEE. (pdf)

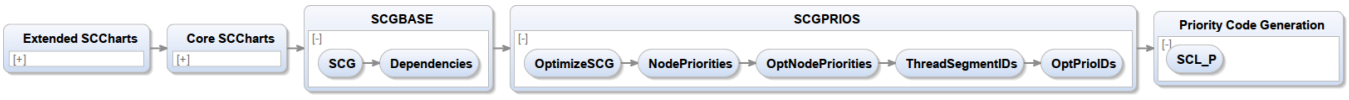
The Priority-Based Compilation

The priority-based low-level compilation approach uses the established compiling chain for SCCharts until the dependency analysis of the SCG is finished. From this point, either the netlist low-level compilation approach or the priority-based low-level compilation approach can be used to generate C code. Because of the different approaches, it is possible, that one of the compiler finds a valid schedule for an SCChart, while the other fails.

General

In contrast to the netlist compiler, this compiler targets only software. It is able to schedule cycles, as long as they only contain transition edges. This restriction is necessary to ensure, that the sequentially constructive model of computation is not violated.

The priority-based low-level compilation approach compiles an SCG enriched by the results of the dependency analysis to SCL_P, whose basis is the programming language C and which is enriched by the SCL_P macros. Therefore the regions of the SCG are fragmented into threads, whose priority determines the order in which the threads are executed. A thread might change its priority, which results in a context switch. The adiministration of the threads is done by the SCL_P macros. As the calculation of the thread priorities is not trivial, it done stepwise, which should help the user to understand how it is done. This is provided by a KiCo compilation chain, which can be called from the SCCharts editor or from the SCG editor.The image below shows the compilation chain as shown by the SCChart editor:



The target language SCL_P:

SCL_P is a leaner variant of Synchronous C. It also provides a deterministic thread administration for C which is implemented as macros. As the grammar of an SCG is simpler, less macros are required than for Synchronous C. The macros provided to a programmer are shown below.

Each thread is identified by a unique prioID, which acts as identifier and as priority for the scheduling. The prioID can be changed, e.g. if a thread waits for a result from another thread. The threads are scheduled in descending order of their corresponding prioID. SCL_P uses cooperative scheduling.

Macros	
`tickstart(p)`	Starts the program, the main thread gets the priority p
`forkn(label1, p1, ..., labeln, pn)`	Forks n processes, which start at label label*n* and have priority p*n*
`par`	Acts as barrier between two threads, deactivates the thread before the barrier and removes it
`joinn(p1, ...)`	Waits until all prioIDs between the braces have finished. It is necessary to consider every prioID, which a thread might have during its execution.
`prio(p)`	Changes the prioID of the current thread, which usually results in a context switch
`pause`	pauses the current thread, as a result, the next thread is started
`tickreturn()`	Returns if program has finished

Requirements for a translation to SCL_P:

- Each thread needs at least one unique prioID.
- The first child inherits the prioID of the parent thread
- The parent threads inherits the prioID of the thread, which performs the join afterwards.
- The thread whose exit node has the lowest prioID has to perform the join.
- The thread, which should perform the join is forked last
- A thread can lower it's prioID during a tick, it should only rise its prioIDs just before a pause.

Transformation Steps

The expected input for the compilation chain is an SCG enriched by the results of the depenency analysis. As the compiler does not use any other results from previous compilation steps, any transformation chain, which results in an SCG can be used.The stepwise calculation of the node priorities and the code generation are described in this section.

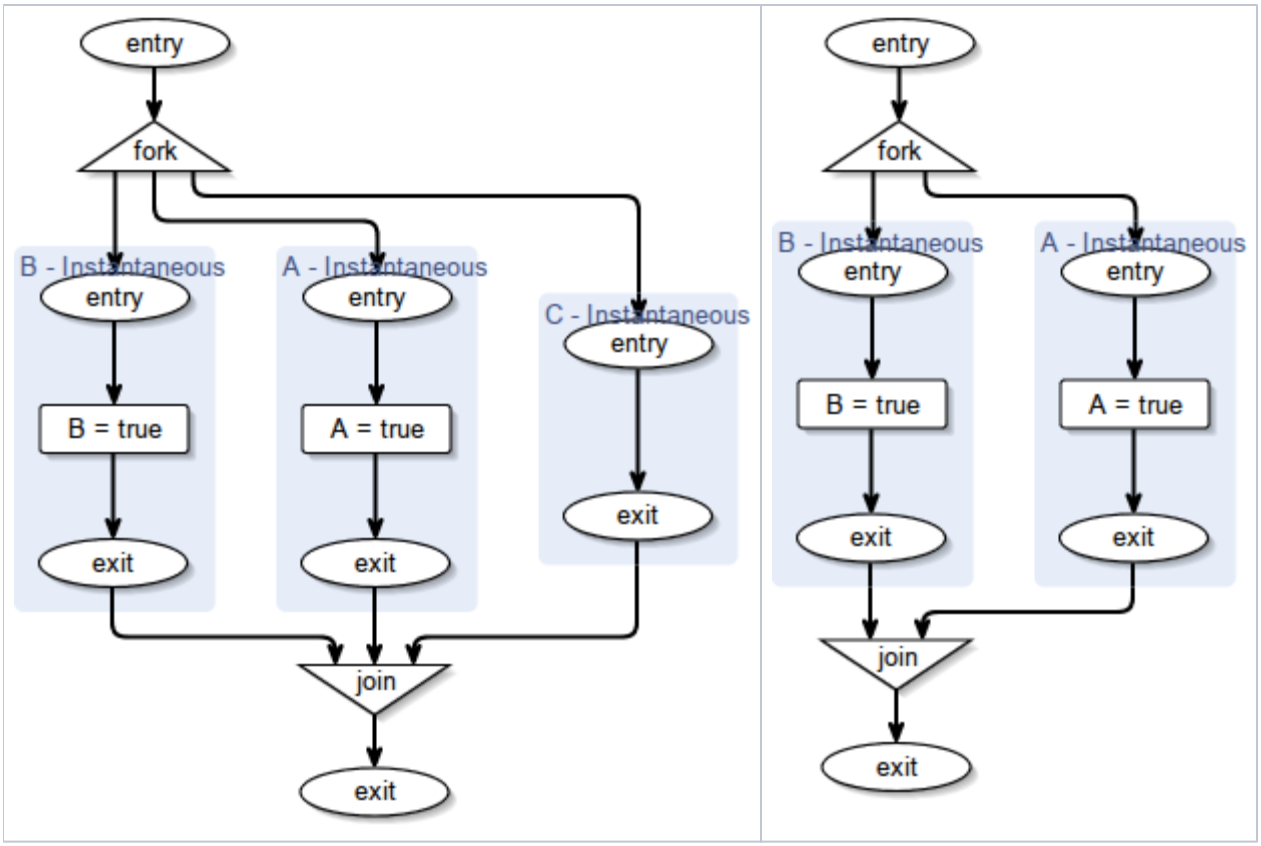
Requirements for a translation to SCL_P:

- Each thread alway needs to have a unique priority, because the priority is used as a priority for the schedule and additionally, the priority is used as identifier for the thread. Therefore the resulting priority for the schedule is called prioID
- The thread, which is forked first should have the highest prioID, because SCL_P only provides cooperate scheduling and the first thread ist started first (in this case the first thread alsway has the same prioID its the parent thread - because of the calculation below)
- The parent threads inherits the prioID of the thread, which performs the join
- The thread whose exit node has the lowest prioID has to perform the join.
- The thread, which should perform the join is forked last
- A thread can lower it's prioID during a tick, it should only rise its prioIDs just before a pause.

OptimizeSCG:

The OptimizeSCG transformation deletes regions from the SCG, which only consist of an enty and exit node. After the code generation, these regions are transformed to empty threads, which only consist of a label and are therefore rejected by a later compilation of the resulting SCL_P/C program. However, for real world examples, it is unlikely, that the user models a region without any further functionalty, therefore this step might be removed.

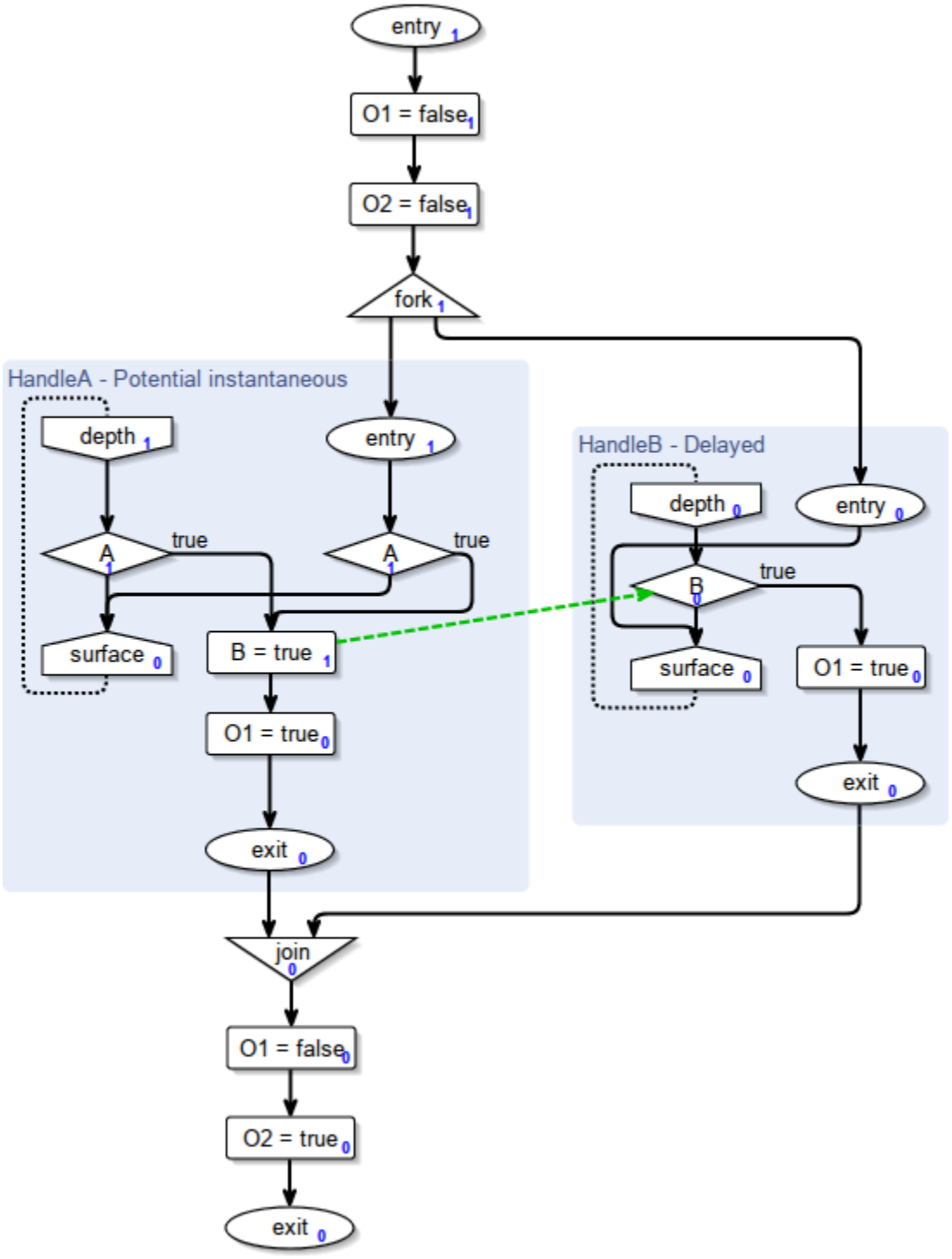
Before optimization:	**After optimization:**



This figure shows that thread C has been removed. If only one thread is left, then the fork and join nodes are removed. If no thread is left, then the parent node from the fork node is connected to the child node of the join node.

NodePriorities:

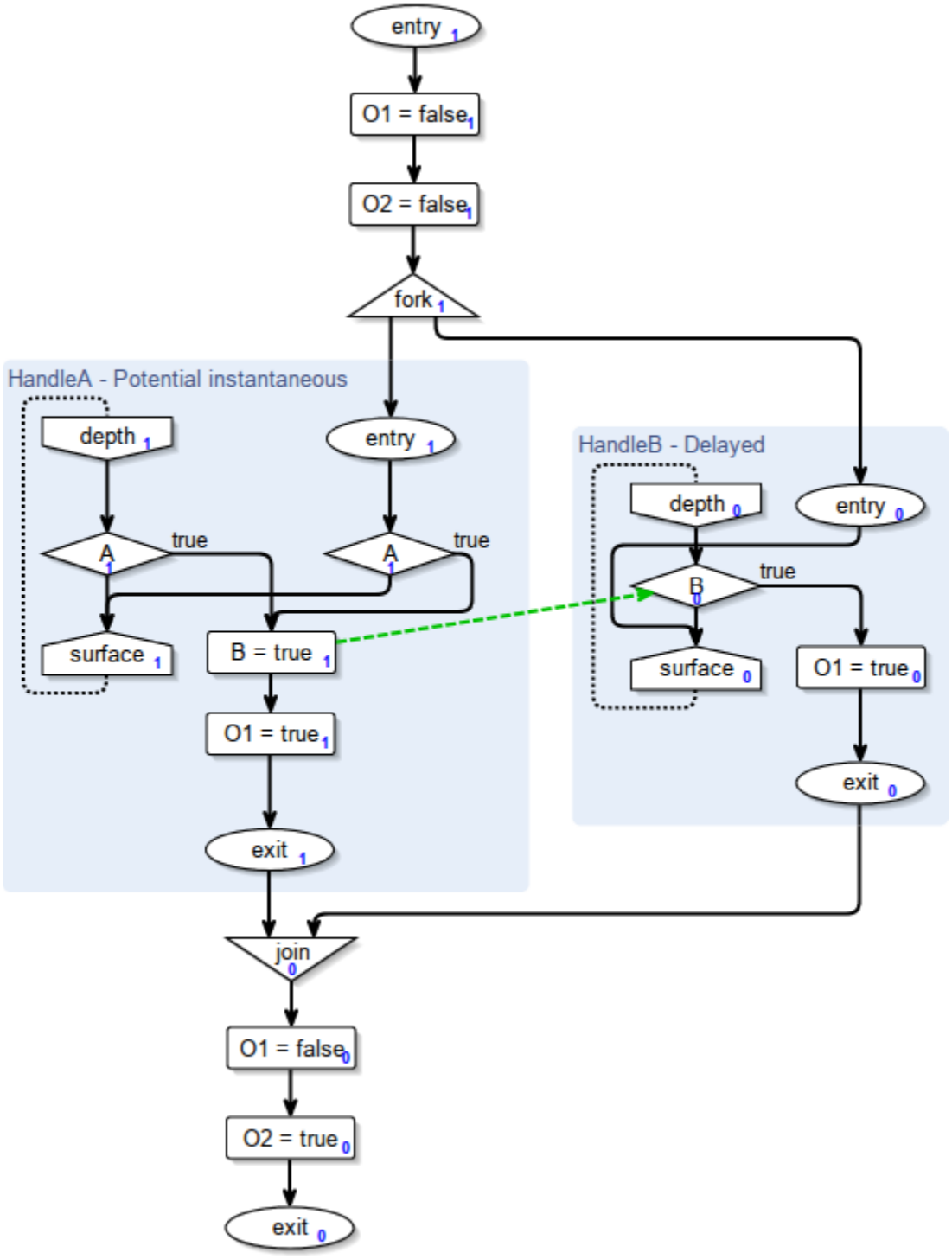
This transformation step uses the results from the dependency analysis. It checks, whether a valid schedule for the SCG exists and calculates the node priorities afterwards. Therefore, the strongly connected components of the SCG are calculated, where the nodes of the SCG are the nodes of directed the graph which is connected by dependency and transition edges. Pause edges are ignored. If such a strongly connected component contains a dependency edge, the SCG is not schedulable. Otherwise, the SCG is schedulale and the node priorities, which are crucial for the schedule, can be determined. The priority of a node is the longest path originating from that node, where strongly connected components are considered as a single node and transition edges have weight 0 and dependency edges have weight 1. Again, pause edges are ignored. The theoretical foundation of this transformation step can be found here.



The image shows the resulting node priorities for ABO (blue).

OptNodePriorities:

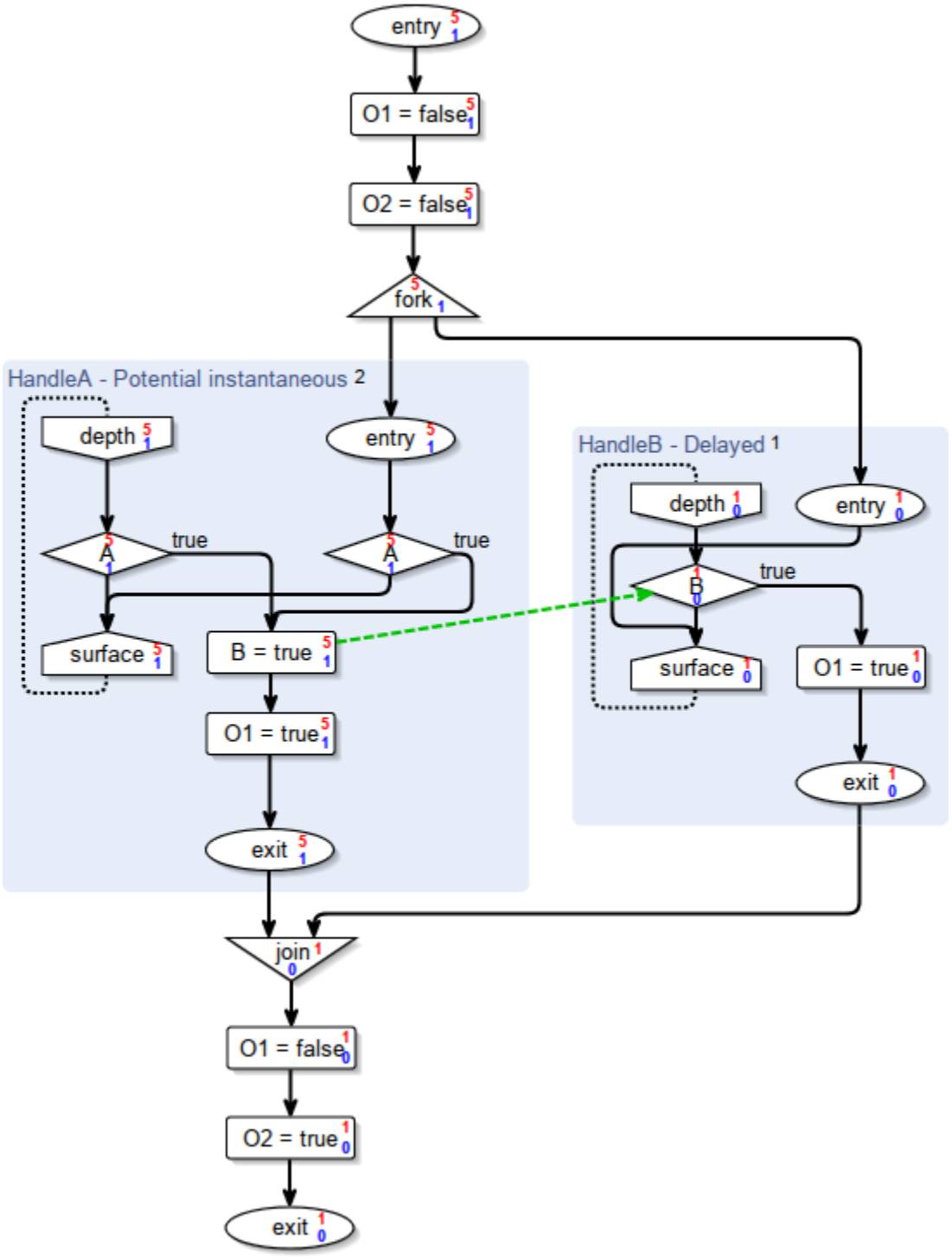
This transformation step is optional. It reduces the number of context switches. Surface nodes usually have the priority 0. Additionally, exit nodes usually have the same priority as the exit nodes of their sibling threads. The resulting node priorities of the ABO example shown in the previous transformation step illustrates this. However it is often unnecessary to perform a context switch, just because an exit or surface node appears, especially because in this case, the order in which the exit/surface nodes are scheduled does not matter. Therefore the exit and surface nodes are taken as starting point. The algorithm moves upward from those nodes and checks, whether any parent node with a higher priority exists. If more than one parent exists, the minimal priority is taken. The search terminates immediatly, if an entry, join or depth node is found and additionally, if a node has an incoming dependency. This is important, because otherwise, the order of the threads might be corrupted. Another restriction is, that the node, which is forked first, cannot perform a join, so if the node which is forked first is the node with the lowest exit priority after this optmization, the exit priority of another thread has to be reduced.



ABO with optimized node priorities.

ThreadSegmentIDs:

SCL_P requires unique priorities for the schedule. In case, that nodes of different threads have the same priority, the thread segment IDs decide, which thread comes first. As described above, a thread, which forks other threads hands it's own prioID, and therefore it's thread segment ID over to the first child and inherits the prioID and therefore the thread segment ID from the child, which performs the join. The assignment of the thread segment IDs is done by a modified depth-first search, which stops at each join node, until its predecessors have received their thread segment ID. Because a prioID should be only lowered, the assignment algorithm starts with the highest thread segment ID, which is calculated by (number of entry nodes) - (number of forknodes). Although the thread segment ids are shown within the region, additionally the unoptimized prioIDs are shown within the graph. This is because a thread segment ID changes after a join and it should help the user to understand, which node belongs to which thread segment ID.



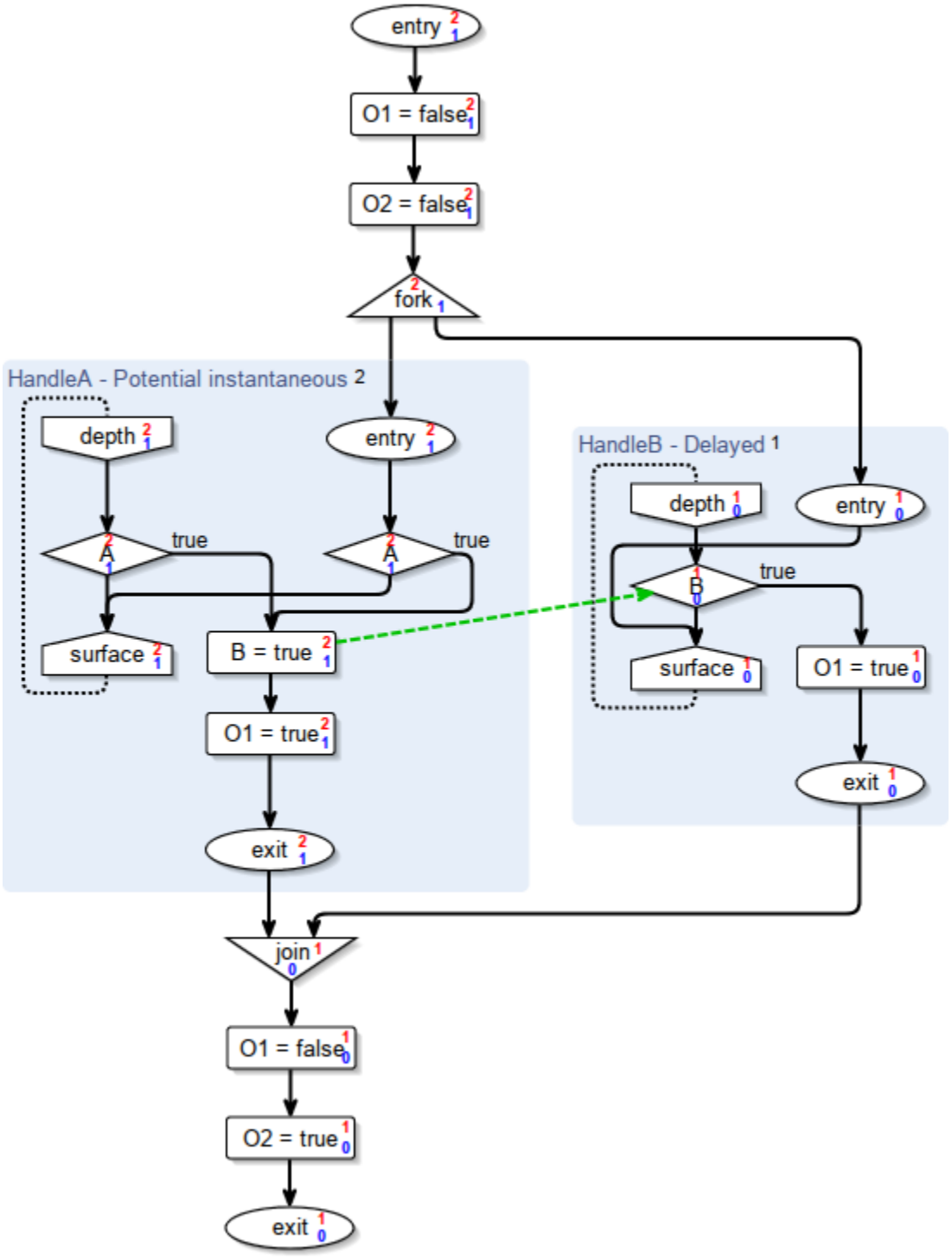
ABO with thread segment IDs (black numbers in the region) and unoptimized prioIDs (red numbers at the nodes)

OptPrioIDs:

The unique prioIDs are calculated by the following formula:

prioID = (node priority) * (number of thread segment ids) + (thread segment id)

The drawback of this formula is, that many prioIDs remain unused. This transformation compresses the prioIDs in use. If any prioID is unused, the thread with the next higher prioID gets that prioID. This makes the bookkeeping for SCL_P more compact.



ABO with optimized prioIDs.

SCL_P:

The translation from the SCG with prioIDs to SCL_P is straightforward. Assignments and Conditionals are translated to their corresponding C-Code. Labels and gotos are used, if such a node is visited twice. Surface and their corresponding depth nodes are translated to pause statements. If the assigned prioID changes from one node to another, a prio(p) statement added. However, join, fork and entry nodes need more attention. For each fork node it is important to ensure, that the node, which has the highest prioID is translated first and the node which performs the join, which is the node with the lowest prioID assigned to its exit node is translated last. The labels for the threads are the names of the corresponding regions, if they exist and are unique. Otherwise a number is added to the region name or a new label is created. For each fork*n* with n < 1 a corresponding macro has to be generated. Likewise, a macro needs to be generated, if join*n* joins more than one prioID. If another thread has a higher prioID than the exit node of the joining thread, this prioID will be scheduled first and therefore, join does not have to wait for that prioID to finish. However it might happen, that a thread stops because of a pause statement, then the prioID indicated by the corresponding depth node has to be considered by the join. If the prioID of a parallel thread is lower than the prioID of the joining node, it has to be considered as well. Entry nodes hand the corresponding labels of the threads over to the next node, if this is not an exit node or a surface node with a depth node, which results in a prioID switch. This avoids the generation of unnecessary labels.

```
/*****************************************************************************/
/*                G E N E R A T E D      C    C O D E                        */
/*****************************************************************************/
/* KIELER - Kiel Integrated Environment for Layout Eclipse RichClient        */
/*                                                                           */
/* http://www.informatik.uni-kiel.de/rtsys/kieler/                           */
/* Copyright 2014 by                                                         */
/* + Christian-Albrechts-University of Kiel                                  */
/*   + Department of Computer Science                                        */
/*     + Real-Time and Embedded Systems Group                                */
/*                                                                           */
/* This code is provided under the terms of the Eclipse Public License (EPL). */
/*****************************************************************************/

#define _SC_ID_MAX 2
#include "scl_p.h"
#include "sc.h"

bool A;
bool B;
bool O1;
bool O2;

int tick()
{
    tickstart(2);
    O1 = false;
    O2 = false;
    fork1(HandleB,1){
        HandleA:
        if (A){
            label_0:
            B = true;
            O1 = true;
        } else {
            label_1:
            pause;
            if (A){
                goto label_0;
            } else {
                goto label_1;
            }
        }
    } par {
        HandleB:
        pause;
        if (B){
            O1 = true;
        } else {
            goto HandleB;
        }
    } join1(2);
    O1 = false;
    O2 = true;
    tickreturn();
}
```

The compilation result for ABO.

Packages belonging to this Project:

de.cau.cs.kieler.scg.prios:

- Contains the calculation of the priorities
- de.cau.cs.kieler.scgprios.extensions
  - provides some extensions for the priority calculations
- de.cau.cs.kieler.scgprios.extensions.export
  - provides an interface to extensions, which are also used by the de.cau.cs.kieler.scg.prios.sclp package
- de.cau.cs.kieler.scgprios.optimizations
  - provides the classes to optimize the node priorities and the prioIDs
- de.cau.cs.kieler.scgprios.priority
  - provides the classes to check the schedulablility and calculate the node priorities, thread segment ids and prioids (all unoptimized)
- de.cau.cs.kieler.scgprios.results
  - provides result types for the KielerCompilerContext
- de.cau.cs.kieler.scgprios.transform
  - provides the transformation methods for KiCo

de.cau.cs.kieler.scg.prios.sclp

- provides transformation, which translates the SCG with the priority information to SCL_P

de.cau.cs.kieler.kexpressions.c

- Contains translation from KExpressions to C (used for the translation of the SCG to SCL_P)

de.cau.cs.kieler.sclp

- A modified copy of de.cau.cs.kieler.sc, adapted for SCL_P (used by the simulation)
- The SCL_P macros can be found here

de.cau.cs.kieler.sccharts.sim.sclp

- A modified copy of de.cau.cs.kieler.sccharts.sim.c, used for simulation the SCChart or SCG in SCL_P

de.cau.cs.kieler.sccharts.sim.sclp.test

- A modified copy of de.cau.cs.kieler.sccharts.sim.c.test, adapted for regression testing

Packages modified for this project:

de.cau.cs.kieler.scg.klighd: SCGraphDiagramSynthesis.xtend

- The node priorities, thread segment ids and prioIDs are now shown in the SCG