Structural Properties of Questionnaires

Börkur Sigurbjörnsson
M.Sc. Thesis
Institute for Logic, Language and Computation
Universiteit van Amsterdam

October 7, 2002

Abstract

Questionnaires appear in many places around us. Their structure can be quite complex. It is therefore beneficial to reason about them automatically. In this thesis I will define questionnaires; identify their structural properties; and demonstrate how formal methods, namely model checking, can assist in checking structural properties of questionnaires. The SMV model checker is used for the model checking.

Contents

1 Introduction **3**
- 1.1 Work processes . . . 3
- 1.2 Aim of work processes . . . 7
- 1.3 Execution of a work process . . . 8
- 1.4 Properties of work processes . . . 8
  - 1.4.1 Static checks . . . 8
  - 1.4.2 Dynamic checks . . . 10
  - 1.4.3 Property retrieval . . . 10
  - 1.4.4 Post traversal checks . . . 11
  - 1.4.5 Work process equivalence . . . 11
  - 1.4.6 Classification checks . . . 12

2 Formalising work processes **12**
- 2.1 Definition of a work process . . . 12
- 2.2 Definition of a work process execution . . . 16

3 Reasoning about work processes **16**
- 3.1 Examples of model checking applications . . . 17
- 3.2 Model checking work processes . . . 18
- 3.3 Work processes as transition systems . . . 18
- 3.4 CTL . . . 21
- 3.5 Work process properties as CTL formulas . . . 22
  - 3.5.1 Properties of linear- and graph- work processes . . . 23
  - 3.5.2 Properties of linear work processes . . . 24
  - 3.5.3 Properties of graph work processes . . . 24
- 3.6 SMV . . . 24
- 3.7 Model checking in SMV . . . 25
  - 3.7.1 Coding linear work process . . . 26
  - 3.7.2 Coding a deterministic work process . . . 28
  - 3.7.3 Extensions . . . 32
- 3.8 Automatic SMV generation . . . 33
- 3.9 Case studies . . . 34

4 Case study - graph work process **34**
- 4.1 Formulation in SMV . . . 34
- 4.2 Checking properties . . . 35
- 4.3 Runtime results . . . 35
- 4.4 Conclusions . . . 40

5 Case study - linear work processes **40**
- 5.1 Formulation in SMV . . . 41
- 5.2 Checking properties . . . 41
- 5.3 Runtime results . . . 41
- 5.4 Checking work process equivalence . . . 42
- 5.5 Conclusions . . . 44

6 Case study - classification 44
6.1 Formulation in SMV . . . 46
6.2 Runtime results . . . 47
6.3 Conclusions . . . 47

7 Conclusions 47

A SMV examples 50
A.1 Graph work process . . . 50
A.2 Linear work process . . . 55
A.3 Equivalence checking . . . 57
A.4 Classification . . . 61

Preface

Questionnaires are used everywhere around us. Some exist in explicit form, such as surveys performed by a market research company. Others might not exist explicitly, such as the questions a doctor asks a patient to try to determine why he is not feeling well. Yet another type are questionnaires that are normally not referred to as questionnaires, such as a reactive and/or investigative procedure to be performed in case of a melt-down in a nuclear power plant.

All the above examples seem to share similarities. They are sets of questions or instructions, posed either explicitly (Do you eat meat?) or implicitly (If the reactor is on fire then run away). They have instructions concerning which questions to ask and in which order. Again, this might be either explicitly written down (If the answer to meat question was "yes", then ask question nr. 3) or depending on the intuition and experience of the questioner (e.g. in the case of the doctor).

From a mathematical point of view, questionnaires can be viewed as directed graphs. The nodes of the graph represent questions, and are labelled with a question number or text. The edges represent question successor relations, labelled with a constraint. For a question pair connected by an edge, the latter question is to be asked immediately after the former if and only if the constraint holds. Completing the questionnaire is like traversing the graph from some initial node to an end node.

As questionnaires get longer and more complex, it might not be immediately evident which properties hold for them. For example, it might take some effort to answer questions like "Are all parts of the graph reachable?", meaning that all the questions can be asked, that there are no superfluous questions; "Are there parts of the graph which are always traversed?", saying which questions are always posed; "Is the choice of a path through the graph ever ambiguous?", meaning that at some point when answering the questionnaire, the choice of next question to answer is not unique etc. Answering these questions can be viewed as checking certain local/global properties of graphs.

In this thesis I will formalise the language in which we talk about questionnaires. Furthermore I will show how formal methods can be used to analyse explicitly defined questionnaires and automatically address general issues such as those raised above. This thesis will concentrate on the structural properties of questionnaires, and will not try to address their sociological aspects. Hence we will not try to solve problems like "How should questions be phrased?" or "In which order should the questions be asked, in order to get unbiased results", "How do I know that my questionnaire gives me reliable results". Information on the sociological aspects of questionnaires and surveys can for example be found in [17, 18].

The thesis is organised as follows. In Section 1 we introduce questionnaires and list the properties we want to verify. In Section 2 we define questionnaires more formally. In Section 3 we describe how the SMV model checker can be used to verify the structural properties of questionnaires. In Sections 4, 5 and 6 we will analyse three case studies. In Section 4 we will look at how to verify structural properties of a questionnaire defined as a graph. In Section 5 we will look at how to verify structural properties of a questionnaire defined as a list of questions. We also look at how to approach the problem of verifying whether two questionnaires are equivalent. In Section 6 we will look at how to verify properties of a questionnaire that is used to classify its respondents. In Section 7 we will conclude and discuss future work. In Appendix A the SMV code used in the case studies can be found.

1 Introduction

Since our mind has in most cases reserved the word "questionnaire" for a subset of explicitly defined questionnaires, I will use the term "work process" to denote a questionnaire (explicitly or implicitly defined). I will as well replace "questions" with "tasks". I hope that this terminology will do more good than harm.

1.1 Work processes

A work process is a set of tasks, along with a description of the order in which the tasks are to be performed. As a task is performed, zero or more variables are updated. Variable updates can be thought of as answers to questions, or as results of some calculation.

In its simplest form, a work process is an ordered list of tasks. To execute the work process, one simply starts at the beginning of the list and performs the tasks one after another until the end of the list is reached.

Example 1. Let us look at a simple questionnaire used at a hospital for gathering information about a patient's health. Note that this questionnaire is the fiction of the author of this thesis.

1. Do you smoke?
    (a) Yes
    (b) No
2. Do you exercise regularly?
    (a) Yes
    (b) No
3. Have you consumed an alcoholic beverage in the past three days?

(a) Yes
(b) No

⊣

It is common that not all questions of a questionnaire are posed to every user. The appropriate questions are chosen with respect to the previous answers from the user. Example 2 is an example of a work process represented as a list of tasks, with some tasks only performed under conditions.

Example 2. Let us expand the questionnaire in Example 1 by conditional questions

1. Do you smoke?
   (a) Yes
   (b) No
2. (if Yes to question 1) How much do you smoke?
   (a) Less than a half pack a day.
   (b) A half to one pack a day.
   (c) More than one pack a day.
3. (if Yes to question 1) Have you tried to quit smoking?
   (a) Yes
   (b) No
4. Do you exercise regularly?
   (a) Yes
   (b) No
5. (if Yes to question 4) How often do you exercise?
   (a) Six or seven times a week.
   (b) Three to five times a week
   (c) Once or twice a week.
   (d) Not every week.
6. Have you consumed an alcoholic beverage in the past three days?
   (a) Yes
   (b) No

⊣

Let us look at a more complex example of a work process represented as a list, with conditioned tasks. In the following example, let $Q_i(x_j)$ denote the task labelled i, that updates the variable x_j. In the examples, for the sake of simplicity, the tasks update only one variable each, but one could easily imagine how this notation could be extended to include tasks that update more than one variable.

Example 3. Example of a work process, written as a list of tasks. This is an actual survey made by a market research company.
$Q_0(x_0)$
if (x_0=a or x_0=b or x_0=c) then $Q_1(x_1)$
if (x_0=d or x_0=e or x_0=f or x_0=g) then $Q_2(x_2)$
if (x_0=b or x_0=e or x_0=g) then $Q_3(x_3)$
if (x_0=d or x_0=e or x_0=f or x_0=g) then $Q_4(x_4)$
if (x_0=a or x_0=b or x_0=c) then $Q_5(x_5)$
if (x_0=d or x_0=e or x_0=f or x_0=g) then $Q_6(x_6)$
if (x_6=a) then $Q_7(x_7)$
if (x_6=a) then $Q_8(x_8)$
if (x_8=1) then $Q_9(x_9)$
if (x_6=b or x_6=c) then $Q_{10}(x_{10})$
if (x_0=g) then $Q_{11}(x_{11})$
if (x_0=d or x_0=e or x_0=f or x_0=g) then $Q_{12}(x_{12})$
if (x_{12}=1) then $Q_{13}(x_{13})$
$Q_{14}(x_{14})$
The actual values of the Qs are irrelevant to this thesis ⊣

Not all work processes are represented as a list of tasks. Another form in which a work process can be presented is one in which routing is determined after each task has been performed. Upon completion of a task, the routing to the next task is determined by evaluating constraints. One can look at this as a directed graph, where the nodes are tasks and the edges are routing options.
Figure 1 shows how the work process in Example 3, can be represented as a graph. The nodes represent the tasks and are labelled with the task number. The edges represent task successor relation and are labelled with the constraint that must hold for that relation to be traversed. As an example task 2 is performed immediately after task 0 if and only if the variable x_0 was given a value from the set $\{d, e, f, g\}$. Example 4 is another example of a work process represented as a graph.

Example 4. Let us look at an extended version of the questionnaire in Example 2, but now represented as a graph. The questions 1-7 are the nodes of the graph. The routing instructions after each question determine the edges of the graph. As an example node 1 has two edges going out of it, one into node 2 and another into node 4; while node 2 has only one edge going out of it, the one into node 3.

1. Do you smoke?
   (a) Yes
   (b) No

   If Yes then goto question 2
   Else goto question 4
2. How much do you smoke?
   (a) Less than a half pack a day.
   (b) A half to one pack a day.
   (c) More than one pack a day.

   Goto question 3

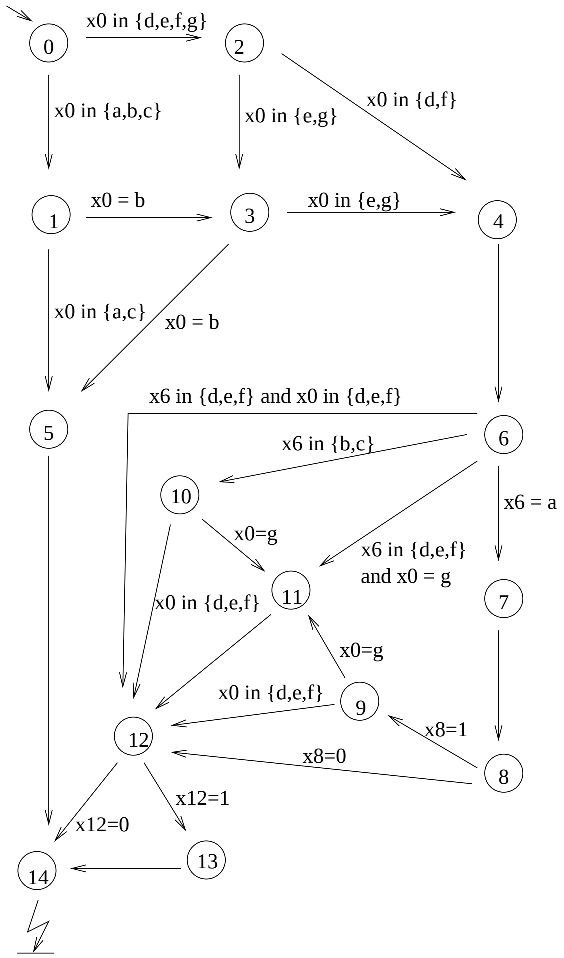

Figure 1: Graph representation of Example 3

3. Have you tried to quit smoking?
   (a) Yes
   (b) No

   Goto question 4
4. Do you exercise regularly?
   (a) Yes
   (b) No

   If Yes then goto question 5
   Else goto question 7
5. How often do you exercise?
   (a) Six or seven times a week.
   (b) Three to five times a week

(c) Once or twice a week.

(d) Not every week.

If Yes to question 3 and Yes to question 4 then goto question 6
Else goto question 7

6. Did you try to quit smoking in order to improve your exercise performance?

   (a) Yes

   (b) No

   Goto question 7

7. Have you consumed an alcoholic beverage in the past three days?

   (a) Yes

   (b) No

   End of process

⊣

While Examples 1, 2 and 4 are relatively simple, Example 3 is more complicated. When work processes reach the complexity level of Example 3, their structural properties might not be trivially decided, hence we might want to be able to derive them automatically. Before we list the properties that might be of interest, let us look at the aims of work processes.

1.2 Aim of work processes

Not only do work processes differ in representation, but they might also have several different purposes. Here we list the most common ones.

Data acquisition Collection of data for statistical analysis. E.g. a survey made by a market research company.

Classification Collection of data from a subject in order to determine a class in which it falls to. E.g. a program for helping people making decisions [16] (voting, choosing a career, choosing an education path etc.) Another example is provided by patient care systems for determining which disease a patient has, or at least narrowing down the list of possible diseases.

Guidance A list of tasks to guide someone through some procedure. E.g. list of tasks to perform in case of an emergency situation in a nuclear power plant.

Guidance and classification E.g. routine checks and maintenance for aircrafts. The work process guides the maintenance team through the process and can afterwards give some analysis (classification) on the state of the aircraft.

1.3 Execution of a work process

Independent of whether a work process is represented as a graph or a list, its execution can be looked at as a graph. The graph describes all possible ways in which the work process can be executed. Each task is a node in the graph and there is an edge from task t_i to task t_j if and only if task t_j can be performed immediately after task t_i. For the work processes already represented as a graph, the execution graph is the same. For the work processes that are represented as a list of tasks, the execution graph need to be built. Each graph has an initial node and a set of final nodes.

Definition 1.1 (Run). A *run* in a work process is one execution of it. I.e. a path in the graph from the beginning node to a final node. $\dashv$

In terms of questionnaires, filling out a questionnaire is a run.

As mentioned before, a task can update zero or more variables. Updates can be either an answer to a question or results of some calculations. As an example, if a questionnaire has the question "What is your birth date?", one variable might be updated with the answer given by the user and another variable might be updated with the calculated age of the user. Updates can be performed by assigning a fixed value to a variable or by incrementing (decrementing) the value of a variable by some constant. The incrementing (decrementing) feature is, as an example, useful when implementing some sort of scoring. To motivate the scoring feature, one can think of the following example. In a patient check-in questionnaire, an integer variable is increased every time the patient's answer might indicate that he is suffering from depression. Later on, this counter can be used to determine whether there is a need for the depression part of the questionnaire to be asked.

If the aim of a work process was to classify users, the classification is done after the work process has been executed. The classes are identified by constraints on the variables of the work process. The runs that make a class constraint valid, are said to belong to that particular class.

1.4 Properties of work processes

Depending on their particular format and aim, work processes may be required to satisfy certain properties. The properties can be split up in six categories; *static checks*, *dynamic checks*, *property retrieval*, *post traversal checks*, *work process equivalence*, and *classification checks*. Below we provide a brief description of each of them.

1.4.1 Static checks

Static checks are made on what may be called the work process frame, that is on the work process structure without any specific values given to variables. They are aimed to help the designer of a work process to better understand her design and detect flaws.

Let us start with checks that are useful, independent of the format used to represent the work process, i.e., whether as a list or as a graph.

1. *Which tasks can be performed?* We would like to verify that all tasks can be performed. Or more precisely, that there do not exist tasks that

are constrained in such a way that they can never be performed. Suppose a designer makes a mistake when creating a constraint such that the constraint cannot evaluate to true. This would leave all the tasks after this constraint unreachable. Checking this is equivalent to checking which parts of the execution graph are reachable.

2. *Which tasks are always performed?* Knowing this, a work process designer can detect if the work process contains constraints that are so weak that they always evaluate to true. Such constraints are of course useless. Checking this is equivalent to checking which nodes in the execution graph are part of every run going through it.

3. *Are variables initialised before they are used in constraints?* Consider the situation in Figure 2. Suppose that an arrow going out of task F has a

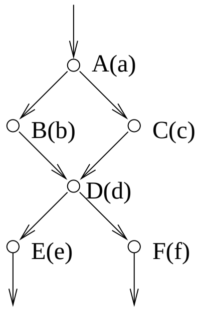

Figure 2: Part of a work process. X(x) represents that task X updates variable x

constraint containing the variable c. Given the information in the figure it is not clear whether or not the variable c has been initialised as we get to task F. It depends on the constraints on edges (A, C) and (D, F). If the constraint on (A, C) evaluates to true whenever the constraint on (D, F) evaluates to true then we are safe. Otherwise we might get into trouble. We would therefore like a mechanism to check if variables are always initialised before used in constraints.

4. *Does there exist a path between two given tasks?* Suppose a work process designer wants to make sure that all users that perform task t_a do not need to perform task t_b. The designer can then check if there exists a path from task t_a to task t_b. If the check is positive, the designer knows that her design is wrong. Similarly if the designer wants to make sure that all users that perform task t_a also perform task t_b, she can check if all paths going from task t_a travel through t_b.

5. *How is the flow of control through the work process?* When working with long and complex work processes it is useful for a work process designer to have an image of the flow of control through the work process.

Although the above properties are helpful for both list-based and graph-based work processes, some are more useful when the work process is represented as a graph, e.g. 2 and some are more useful when the work process is represented as a list, e.g. 5. Some checks are only useful when work processes are defined as a graph.

6. *Does there exist an ambiguous run in the work process?* We want to know if there exist a task where, under a given variable assignment, there exist more than one possible tasks to route to. That is, we want to verify that for all possible variable assignments, constraints on edges, going out of the same task, are mutually exclusive.
7. *Are there deadlocks in the work process?* We want to know if there exists a task such that for some valuation there exists no task to route to. That is, we want to verify that for all tasks and all possible variable assignments, at least one of the constraints on outgoing edges evaluates to true. Combined with the property above, this is the same as saying that whenever we have performed a task, there exists one and only one task that is to be performed next.
8. *Are there cyclic runs in the work process?* Suppose a designer of a work process accidently inserts wrong task number when giving routing instructions, resulting in a route to a previously performed task. The designer has created a cycle in the work process. Usually work processes used for data acquisition are required to be acyclic.

Having verified whether the above properties hold, the designer of a work process should have a much clearer image of whether her design is as intended. Static checks will be considered in Sections 4 and 5.

1.4.2 Dynamic checks

Dynamic checks are made as the work process is being traversed.

1. *Longest self contained run* What is the longest self contained run starting in at task t, given some valuation of the process variables? A self contained run is a sequence of tasks that can be performed without the need to check any constraints. To see the use of this consider a Thin-Client/Fat-Server application operating on an occasionally connected network.
2. *How many tasks are left undone?* At some time during the execution of a work process it might be informative for the user know how many tasks remain to be performed. We would like information of the maximum and minimum number of tasks.

This thesis will focus on static properties of work processes. Dynamic checks will not be considered further.

1.4.3 Property retrieval

There are some interesting properties that can be extracted from the work process.

1. *Length of routes* A work process designer would like to know what is the longest, shortest or average path through the work process. The results of these checks can give valuable information, both to designer and user of a work process, about how long time it is likely to take to execute the work process.

2. *Constraint retrieval* What are the conditions under which specific tasks are performed? For a work process designer the answer to this question can give information about the characteristics of the people that will perform a particular task or the external condition under which the task is performed. She can use the information to realise if her design is as she had imagined.

Properties retrieval will be considered in Sections 4 and 5.

1.4.4 Post traversal checks

There are also some checks that can be made on a single traversal of the process, or on a set of traversals.

1. *Constraint satisfaction* Do the traversals have the property P. (Example: Do all people born in 1976 have pancakes for breakfast?).
2. *Variable setting* Were values given to all the variables in Y?
3. *Task traversals* Were all the tasks in Y traversed?

Observe that in some cases, post traversal checks boil down to mining the data observed by the work process. Analysing this is beyond the scope of this thesis, and will not be considered further.

1.4.5 Work process equivalence

If we have more than one way in which work processes can be represented, we are naturally lead to questions about work process equivalence or similarity. We would like to ask ourselves when two work processes are equivalent. Note that there can be several definitions of equivalence. We will use a strong notion, saying that two work processes are equivalent if given some user input, the two processes perform the same tasks in the same order.

Definition 1.2 (Strong equivalence). Work processes W_1 and W_2 are equivalent if

1. W_1 and W_2 have same tasks and variables.
2. W_1 and W_2 have the same initial task.
3. If the variables of W_1 and W_2 have the same values, and task i was the last task to be performed in both processes, then task j is the next task to be performed in both processes.

$\dashv$

Let us postpone the precise definition of sameness of tasks and variables. For the purpose of this thesis it does not matter how sameness is defined, as long as it is defined. Sameness can be defined as tasks having the same identifier, update the same variables etc.

A weaker definition of equivalence is to say that two work processes are equivalent if they perform the same tasks but not necessarily in the same order.

Work process equivalence will be considered in Section 5.4.

1.4.6 Classification checks

For classification work processes there are several checks we might want to perform.

1. *Are there superfluous classes?* We want to know if we have defined a class in such a way that no runs have a chance of belonging to it.

2. *Is the classification total* We want to know if every run is guaranteed to belong to some class. That is if there are no runs unclassified.

3. *Is the classification unique?* We want to know if each run only belongs to one single class.

Classification checks will be dealt with in Section 6

2 Formalising work processes

Let us now define more precisely what we mean by a work process. There is no standard way of defining a questionnaire or a work processes. We will in this section define work processes in a fashion that is suitable for the purpose of this thesis.

2.1 Definition of a work process

The basic building blocks of work processes are the tasks.

Definition 2.1 (Task). A *task* is a tuple (i, Y), where i is an identifier and Y is a set of variables. Each task can *update* the variables in Y (usually due to user input or internal calculations). A variable update can involve setting the variable to a fixed value, incrementing it or decrementing it. ⊣

The identifier of a task is used to distinguish between two task that update the same variables. In its simplest form, a work process is a set of tasks.

Definition 2.2 (Work Process). A *work process* is a tuple (T, X), where T is a set of *tasks* and X is a set of *variables*. For each task (i, Y) in T, Y is a subset of X. ⊣

Example 5. As an example we can describe the questionnaire in Example 1 as the tuple

$$\Big(\{(Q_{smoke}, s), (Q_{exercise}, e), (Q_{alcohol}, a)\}, \{s, e, a\}\Big)$$

where Q_{smoke}, $Q_{exercise}$ and $Q_{alcohol}$ are task identifiers; and s,e and a are variables for keeping the answers. ⊣

Defining a work process as a set of tasks is too simple for definitions of real world work processes. We need means to define the order in which tasks are performed. To introduce different routing methods we define different types of work processes, starting with the "list of tasks" approach, used in Examples 1, 2 and 3.

Definition 2.3 (Constraint). A *constraint* on a set of variables X is a formula, made up from elements of X, constants and the symbols $=, <, \wedge, \vee$ and $\neg$. $\dashv$

Definition 2.4 (Linear Work Process). A *linear work process* is a 5-tuple $(T, X, C, \sigma, \gamma)$, where (T, X) is a *work process*, C is a set of *constraints*, $\sigma : T \to \{0, \ldots, |T| - 1\}$ is an ordering of the tasks and $\gamma : T \to C$ is a partial function, associating constraints with tasks. Note that σ is a bijection. $\dashv$

Let us clarify this definition with an example

Example 6. The questionnaire in Example 2 can be defined as the linear work process $(T, X, C, \sigma, \gamma)$, where [1]

$$\begin{aligned}
T &= \{(Q_1, x_1), (Q_2, x_2), (Q_3, x_3), (Q_4, x_4), (Q_5, x_5), (Q_6, x_6)\} \\
X &= \{x_1, x_2, x_3, x_4, x_5, x_6\} \\
C &= \{x_1 = yes, x_4 = yes\} \\
\sigma &= \{(Q_1, 0), (Q_2, 1), (Q_3, 2), (Q_4, 3), (Q_5, 4), (Q_6, 5)\} \\
\gamma &= \{(Q_2, x_1 = yes), (Q_3, x_1 = yes), (Q_5, x_4 = yes)\}
\end{aligned}$$

$\dashv$

The notion of a linear work processes can be extended to allow for nested conditionals.

Definition 2.5 (Conditional Work Process). A *conditional work process* is a sequence of *tasks* and *control structures*. Control structures have the format IF c THEN t1 ELSE t2 FI, where c is a constraint and t1 and t2 are sequences of tasks and control structures. $\dashv$

Example 7. As an example

```
(Q_0,x_0)
IF x_0=3 THEN
  (Q_1,x_1)
  (Q_2,x_2)
ELSE
  (Q_3,x_3)
  IF x_3=7 THEN
    (Q_4,x_4)
  ELSE
    (Q_5,x_5)
  FI
FI
(Q_6,x_6)
```

is a conditional work process. $\dashv$

Note that the linear work processes form a subset of the conditional work processes. The linear version of a work process is probably the most widely used in practice. An alternative means of representation for work processes are graphs.

[1] Here we cheat in the definition of σ and γ by saying Q_i when we mean (Q_i, x_j)

Definition 2.6 (Graph Work Process). A *graph work process* is a 7-tuple $(T, X, E, C, \gamma, t_0, t_{end})$, where (T, X) is a work process; $E \subseteq T \times T$; $(T \cup \{t_{end}\}, E)$ is a directed graph; t_{end} is a constant task to mark the end of a work process; C is a set of constraints; $\gamma : E \to C$ assigns constraints to edges; $t_0 \in T$ is the initial task. *Routing* through the graph is determined by the constraints (see below). ⊣

Let us clarify this definition with an example.

Example 8. We can define the questionnaire in Example 4 as the graph work process $(T, X, E, C, \gamma, t_0, t_{end})$ where, [2]

$$\begin{array}{rcl}
T & = & \{(Q_1, x_1), (Q_2, x_2), (Q_3, x_3), (Q_4, x_4), (Q_5, x_5), (Q_6, x_6), (Q_7, x_7)\} \\
X & = & \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\} \\
E & = & \{(Q_1, Q_2), (Q_1, Q_4), \\
 & & (Q_2, Q_3), \\
 & & (Q_3, Q_4), \\
 & & (Q_4, Q_5), (Q_4, Q_7), \\
 & & (Q_5, Q_6), (Q_5, Q_7), \\
 & & (Q_6, Q_7), \\
 & & (Q_7, t_{end})\} \\
C & = & \{\top, \\
 & & x_1 = yes,\ x_1 = no, \\
 & & x_4 = yes,\ x_4 = no, \\
 & & x_3 = yes \wedge x_4 = yes,\ x_3 = no \vee x_4 = no\} \\
\gamma & = & \{(e_{12}, x_1 = yes), (e_{14}, x_1 = no) \\
 & & (e_{23}, \top), \\
 & & (e_{34}, \top), \\
 & & (e_{45}, x_4 = yes), (e_{47}, x_4 = no), \\
 & & (e_{56}, x_3 = yes \wedge x_4 = yes), (e_{57}, x_3 = no \vee x_4 = no), \\
 & & (e_{67}, \top), \\
 & & (e_{7end}, \top)\} \\
t_0 & = & Q_1
\end{array}$$

⊣

In the definition of a graph work process, nothing is mentioned about how routing is performed on the graphs. Let us now define a few different routing strategies, by defining a few different types of work processes.

Definition 2.7 (Deterministic Work Process). A *deterministic work process* is a graph work process. Routing through the graph is determined by the constraints. The constraints must be formulated so that one and only one of the constraints, on edges leading from a task, can be satisfied at the same time. ⊣

[2] Here we cheat by writing Q_i instead of (Q_i, x_j), e_{ij} instead of (Q_i, Q_j) and e_{7end} instead of (Q_7, t_{end})

Example 4 is a deterministic work process.

Definition 2.8 (Weak Deterministic Work Process). A *weak deterministic work process* is a graph work process. Constraints on outgoing edges need not be mutually exclusive. Edges leading from a task and their constraints are ordered by importance. The most important constraint is checked first, only if that constraint does not hold, the second most important constraint is checked etc. ⊣

A weak deterministic work process can be quite easily translated into a deterministic work process by making the constraints, leading from a task, mutually exclusive. This can for example be done by adding to each constraint the conjunction of negations of all constrains that have higher importance than the constraint in question.

Definition 2.9 (Depth First Routing Work Process). A *depth first routing work process* is a graph work process. The underlying graph is a tree. Routing through the tree is again determined by the constraints, but the constraints leading from a task need not be mutually exclusive. After a task has been performed, the outgoing constraints are evaluated in turn, according to some predefined ordering. When a constraint evaluates to true, the route following the constraint is taken. When no constraints evaluate to true, backtracking, to the last branching of the tree, is made and checking of constraints at that point is continued. ⊣

A depth first routing work process can be translated into deterministic work process. We apply the following algorithm to the initial task and an empty queue

```
link(task t, queue E){
  out[] = all edges leaving t.
  for all edges (u,v) in E{
    add an edge (t,v) to the graph
    gamma((t,v)) = gamma((u,v))
  }
  for i=0 to i<out.length{
    link(pi2(out[i]), out[i+1...out.length]+E)
  }
}
```

where the function `pi2` returns the second element of a pair; `gamma` is the constraint assignment function of Definition 2.6; and `out[i+1...out.length]+E` adds the elements `out[i+1]` ... `out[out.length]` to the front of the queue `E`. The result is a weak deterministic work process, which can be translated into a deterministic work process as mentioned above.

In this thesis only two of the work process types will be analysed, linear and deterministic work processes. Linear is chosen because it is very commonly used. It is the format of paper based work processes and since computer based work processes are very often a digital copy of the old paper version, they tend to inherit the linear format. Graph formats are also common for computer based work processes. As for graph work processes, we will only look at deterministic work processes. For handling the other graph formats, one can translate them into a deterministic format and then use the methods mentioned in this thesis. I do however believe that a better solution is to extend the methods to cope with other graph formats.

2.2 Definition of a work process execution

Let us now define more precisely what we mean by an *execution*, a *run* and a *classification*.

Definition 2.10 (Execution graph). An *execution graph* of a work process is a graph that describes the execution of a work process. The nodes of the graph are the tasks of the work process. There is an edge (t, t') in the graph if and only if t' can be performed immediately after task t is performed. Associated with each edge is a constraint, defining the conditions under which task t' is performed immediately after task t is performed. A set of nodes in the graph are called *end-nodes*. A task in a linear work process is an end-node if all the tasks appearing later in the list are performed under constraints. A task in a graph work process is an end-node if it has an edge to t_{end}. ⊣

Definition 2.11 (Run). A *run* trough a work process is a sequence $(t_i, a_i)_{i=0}^k$ where t_i is task in the work process and a_i is a function $x \mapsto y \in Domain(x) \cup \{undef\}$ assigning values to variables (the value $undef$ is used say that a variable has not been given a value). The task t_0 is the initial task and valuation a_0 is the initial valuation of variables. For all, $0 \leq i < k$, (t_i, t_{i+1}) is an edge in the execution graph, a_i makes the constraint on (t_i, t_{i+1}) true and a_i and a_{i+1} differ only in the variables updated by task t_{i+1}. The task t_k is an end-node. ⊣

As an example [3]

$$\big((Q_1, \{(x_1, no)\}), (Q_4, \{(x_1, no), (x_4, no)\}), (Q_6, \{(x_1, no), (x_4, no), (x_6, yes)\})\big)$$

is a run through the work process in Example 6.

Definition 2.12 (Classification). A *classification* is a set of constraints C. A run $(t_i, a_i)_{i=0}^k$ belongs to a class $c \in C$ if and only if the valuation a_k makes c true. ⊣

Example 9. The classification $\{x_1 = yes, x_1 = no\}$ is a classification of the runs of the work process in Example 6. The example run above belongs to the class identified with the constraint $x_1 = no$. ⊣

3 Reasoning about work processes

We have defined work processes and listed numerous properties one would like to check. We now need to decide how to go about verifying that the properties hold.

Approaches to verification are either proof-based or model-based. A detailed description of both approaches can be found in [8]. In the proof-based approach, the system is described by a set of formulas Γ and the property is described as a formula ϕ. The verification method consist of finding a proof for $\Gamma \vdash \phi$, i.e.

[3] In the example we only mention variable assignments that are not $undef$

the formula ϕ can be deduced from the formulas in Γ (Γ proves ϕ). In the model-based approach, the system is represented as a finite model $\mathcal{M}$ and the property is again described as a formula ϕ. The verification method consists of computing whether $\mathcal{M} \vDash \phi$ (ϕ is true in $\mathcal{M}$).

Both the proof based and the model based approach are compositions of three parts:

- Describing the system. I.e. the set Γ for proof based approach or the model $\mathcal{M}$ for the model based approach.
- Describing the property ϕ.
- Verifying that the system satisfies the property.

The two verification part of the two approaches differs in many ways. While the degree of automation is high for model checking, it is low for theorem proving. Theorem proving typically requires guidance from theorem-proving experts, domain experts or both. Hence theorem proving can take a long time. Model checking is done on a finite state machine. Exhaustive case analysis is done on the machine's set of states. Efficient representation of states has enabled model checkers to handle very large state spaces, e.g. 10^{20}.

For the purpose of this thesis, that is, for analysing work processes, I find model checking more suitable than theorem proving. The inductive arguments, which are the foundations of theorem proving, fail here due to discontinuity in value spaces. We would resort to large case analysis in the proof and only local application of inductive proofs. Furthermore, for a given instance of a work process it is easy to build a model of its execution. It is therefore more straight forward to do model checking on that model, rather than axiomatise it and, in a sense, consider more models.

Model checking can handle the case analysis more quickly and automatically. Furthermore, model checkers give valuable feedback if a property is not satisfied. If a bug appears in the design, model checkers are able to give a counterexample.

3.1 Examples of model checking applications

Model checking has proved to be helpful in verifying hardware and software, thanks to the development of tools such as CAESAR [4], CWB [3], MURPHI [2], SMV [11], SPIN [7] and VeriSoft [5].

Hardware verification typically involves model checking of circuits. Kaivola and Kohatsu [9] examine the challenges presented by large-scale formal verification of industrial-size circuits, based on their experience in verifying the class of all micro-operations executing on the floating-point division and square root unit of the Intel IA-32 Pentium® 4 microprocessor.

Many software verification projects involve model checking components of UML. Schäfer et al.[15] describe a prototype tool that is designed to automatically verify whether the interactions expressed by a collaboration can indeed be realized by a set of state machines. State machines are compiled into a PROMELA model and collaborations into sets of Büchi automata. The model checker SPIN is called upon to verify the model against the automata. Chandra et al.[1] use model checking to analyse Lucent Technologies' CDMA call-processing library, using VeriSoft.

In this thesis neither software nor hardware is being analysed. We will model check work processes, a product of a human, with or without the help of work process creation software. In a way the problem here bears more resemblance with hardware verification because the subject, the work process, can be looked at as a kind of a circuit.

3.2 Model checking work processes

Verifying, by model checking, that a system satisfies a given property is divided into three steps.

- Coding the system in the description language of the model checker, resulting in a model $\mathcal{M}$.
- Coding the property in the specification language of the model checker, resulting in a formula ϕ.
- Ask the model checker whether $\mathcal{M}$ satisfies ϕ.

We must decide which language to use for representing the work process execution and its properties. Work process executions have a clear temporal behaviour. Tasks are performed one after another and we can use the words before and after to compare execution times of different tasks. Work process executions also have a clear branching behaviour. Execution of a work process can differ from one run to another, depending on how the variables were updated. Hence the obvious choice is to choose a transition system to represent the execution of a work process and a branching time temporal language to represent its properties.

3.3 Work processes as transition systems

A transition system is a pair (W, R), where W is a non-empty set and $R \subseteq W \times W$. Visually we can look at the elements of W as points in plane and for each two points $u, v \in W$, if $(u, v) \in R$ then there is an arrow connecting u and v in the plane. Figure 3 shows the transition system $(\{a, b, c, d\}, \{(a, b), (b, c), (b, d)\})$. We will describe the execution of a work process as a transition system. Ad-

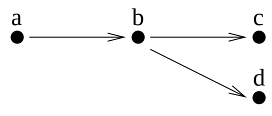

Figure 3: Simple transition system

ditionally we introduce a set of variables and assign values to them for each element in W. More formally we will look at the tuple $(W, R, X, \{L_x | x \in X\})$, such that (W, R) is a transition system, X is a set of variables and $L_x : W \to D_x$ is a mapping from W to the domain of variable x. The mapping L_x determines the value of x at each point w in the transition system.

The execution of a work process is split up in two types of actions, routing decisions and performing of tasks. In our models we distinguish between those actions by a boolean variable r, which is true if routing is being decided and false if a task is being performed. Only one task is handled at a time. We use the integer variable t to denote which task is being handled at the moment. We describe, by examples, two different transition systems, one for linear work processes and another for deterministic work processes.

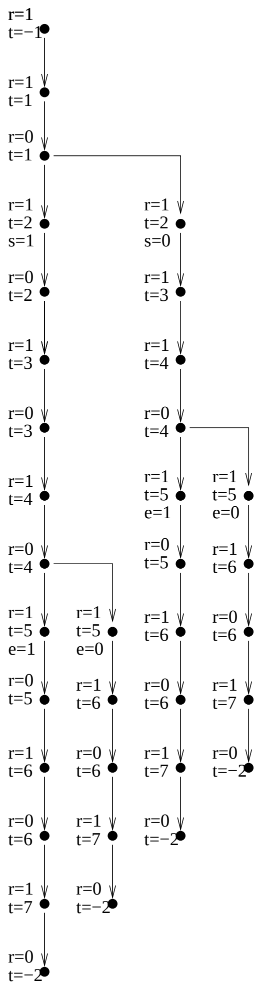

Figure 4: The computational tree for the work process in Example 2

Linear work process The execution of a linear work process can be described by the following pseudo code

```
for t = 1 to n
  if (evaluate(c[t])) then perform task t
```

where we assume that the tasks are enumerated from 1 to n by σ (see Definition 2.4) and c[t] is the constraint in front of task t.

Figure 4 shows how the execution of the work process in Example 2 (page 4) is modelled as a transition system. Each branch from the top to the bottom represents a possible run through the work process. We introduce two additional variables. The boolean variable s is true if the answer to the question *Do you smoke?* was positive, and false otherwise. The boolean variable e is true if the answer to the question *Do you exercise regularly?* was positive, and false otherwise. The handling of each task involves two actions; first evaluating its constraint and if the constraint evaluates to true then the task is performed. In the tree we see that on each path there are one or two nodes marked with each task number. In case of one node, the task's constraint did not evaluate to true and the task is not performed. In case of two nodes, the former node is for constraint evaluation and the latter for performing the task. The tasks are handled in continuously increasing order. Hence the task variable t is increased by one at a time, and increased whenever routing is to be determined. A value -1 of t denotes the beginning of the work process. A value -2 of t means that the end of the process has been reached. The variable t can take the value 7 although the actual tasks are labelled from 1 through 6. This value of the routing variable is used as an overflow at the end of the work process and is never performed. I.e. this value is used to determine routing to the end. In Figure 4, only the value of relevant variables are displayed at each point. If a value of a variable is not displayed, it is the same as it was at the previous point in the system.

Deterministic work process The execution of a deterministic work process can be described by the following pseudo code

```
t = initial task
while (t <> -2)
  perform task t
  t := route(t)
```

where the function `route` evaluates the constraints on edges going out of task t and determines which task is to be performed next. A value -2 of t means the end of the work process.

Figure 5 shows how the work process in Example 4 (page 5) is modelled as a transition system. Each branch from the top to the bottom represents a possible run through the work process. We introduce three additional variables. The boolean variable s is true if the answer to the question *Do you smoke?* was positive, and false otherwise. The boolean variable e is true if the answer to the question *Do you exercise regularly?* was positive, and false otherwise. The boolean variable q is true if the answer to the question *Have you tried to quit smoking?* was positive, and false otherwise. The handling of tasks involves two actions; first performing the task and then determining the routing to the next task. In the tree we see that on each path there are either zero or two nodes marked with each task number. In case of zero nodes, the task is not performed in that particular run. In case of two nodes, the former node is for performing the task and the latter for determining the routing to a next task. The task variable t can bear the values -1 and -2, denoting respectively the beginning and end of the work process. As before, Figure 5 only displays the value of relevant variables at each point.

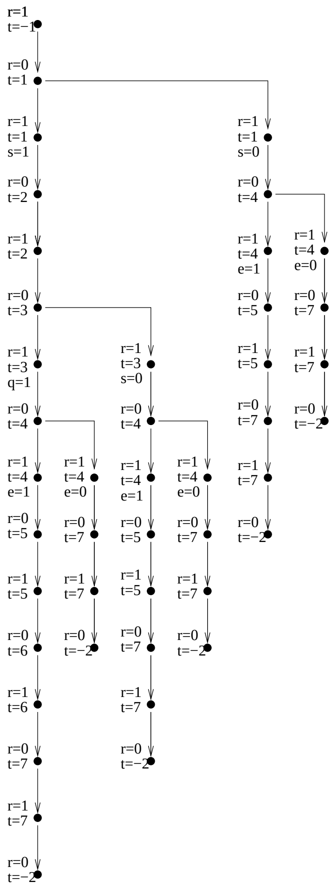

Figure 5: The computational tree for the work process in Example 4

Having built a model of the work process execution, we can start verifying properties. There are some properties that are meaningful for both models, while others only apply to one or the other. Let us now look at how to formulate the properties.

3.4 CTL

Computational Tree Logic (CTL) is a branching time temporal language. CTL has proven to be very useful in verifying hardware and communication protocols, and is being widely used in verification of software. CTL formulas are defined inductively by Backus Naur form:

$$\begin{aligned}\phi ::= \quad & \bot \mid \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \\ & AX\phi \mid EX\phi \mid AG\phi \mid EG\phi \mid AF\phi \mid EF\phi \mid A[\phi U \phi] \mid E[\phi U \phi]\end{aligned}$$

For defining the semantics of CTL, let us consider as a model, the transition system $M = (W, R, L)$, where W is a set of states, R is a transition relation $W \times W$, and a labelling function $L : W \rightarrow \mathcal{P}(Atoms)$, where $Atoms$ is a set of proposition letters, in our case the proposition letters represent $s = yes$, $t = 5$ etc. The meaning of CTL is

1. $\mathcal{M}, w \models \top$ and $\mathcal{M}, w \not\models \bot$ for all $w \in W$
2. $\mathcal{M}, w \models p$ iff $p \in L(w)$
3. $\mathcal{M}, w \models \neg\phi$ iff $\mathcal{M}, w \not\models \phi$
4. $\mathcal{M}, w \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, w \models \phi_1$ and $\mathcal{M}, w \models \phi_2$
5. $\mathcal{M}, w \models \phi_1 \vee \phi_2$ iff $\mathcal{M}, w \models \phi_1$ or $\mathcal{M}, w \models \phi_2$
6. $\mathcal{M}, w \models \phi_1 \rightarrow \phi_2$ iff $\mathcal{M}, w \not\models \phi_1$ or $\mathcal{M}, w \models \phi_2$
7. $\mathcal{M}, w \models AX\phi$ iff for all w' such that wRw' we have $\mathcal{M}, w' \models \phi$
8. $\mathcal{M}, w \models EX\phi$ iff there exists a state w' such that wRw' and $\mathcal{M}, w' \models \phi$
9. $\mathcal{M}, w \models AG\phi$ iff for all paths $w_1Rw_2Rw_3R\ldots$, where w equals w_1, we have for each w_i along the path that $\mathcal{M}, w_i \models \phi$
10. $\mathcal{M}, w \models EG\phi$ iff there exists a path $w_1Rw_2Rw_3R\ldots$, where w equals w_1, and for each w_i along the path we have $\mathcal{M}, w_i \models \phi$
11. $\mathcal{M}, w \models AF\phi$ iff for all paths $w_1Rw_2Rw_3R\ldots$, where w equals w_1, there is some w_i along the path such that $\mathcal{M}, w_i \models \phi$
12. $\mathcal{M}, w \models EF\phi$ iff there exists a path $w_1Rw_2Rw_3R\ldots$, where w equals w_1, and for some w_i along the path we have $\mathcal{M}, w_i \models \phi$
13. $\mathcal{M}, w \models A[\phi_1 U \phi_2]$ iff for all paths $w_1Rw_2Rw_3R\ldots$, where w equals w_1, there is some w_i along the path such that $\mathcal{M}, w_i \models \phi_2$ and for all $j < i$ we have $\mathcal{M}, w_j \models \phi_1$
14. $\mathcal{M}, w \models E[\phi_1 U \phi_2]$ iff there exists a path $w_1Rw_2Rw_3R\ldots$, where w equals w_1, and for some w_i along the path we have $\mathcal{M}, w_i \models \phi_2$ and for all $j < i$ we have $\mathcal{M}, w_j \models \phi_1$

Cases 1-6 are the propositional connectives. Cases 7-14 are temporal connectives, always composed of a pair of letters. The prefix A is universal, meaning that the formula should hold for all outgoing paths. The prefix E is existential, meaning that the formula should hold on some outgoing path. The letter X refers to the next state. The letters G and F refer to all future states and some future state, respectively. The letter U says that some time in the future ϕ_2 is satisfied, and until then, ϕ_1 is satisfied. A more detailed description of CTL can be found in [8].

3.5 Work process properties as CTL formulas

Now that we have built a model of the execution of the work process, that is the transition system, let us see how we can use CTL to express the properties of work processes, listed in Section 1.4.

3.5.1 Properties of linear- and graph- work processes

Some properties are useful to check for both linear and graph work processes.

Which tasks can be performed? In our model, task i can be performed if there exists a branch of the computation tree on which there exists a point where $t = i$ and $r = 0$. I.e. task i can be performed if the formula

$$EF(t = i \wedge r = 0) \tag{1}$$

holds. This property should of course hold for every task.

Which tasks will always be performed? In our model, task i is always performed if on every branch of the computation tree, there exists a point where $t = i$ and $r = 0$. Task i is always performed if the formula

$$AF(t = i \wedge r = 0) \tag{2}$$

holds. This might seem a trivial question because we need only look at the work process and see which tasks do not have any "if" statement in front of them. Checking this might however assist the designer to spot constraints that are always satisfied and hence useless.

Extracting information about shortest and longest path We can do this indirectly by checking for each i, $0 \leq i \leq number\_of\_tasks$, whether there exists a path in the computation tree where the number of states, having $r = 0$, is $i + 1$. This might result in quite complicated formulas. We can also do this by introducing a new variable k, which is initially zero and increased by one every time a task is performed. Then we need only check whether for each i, $0 \leq i \leq number\_of\_tasks$, the formula

$$EF(t = -2 \wedge k = i) \tag{3}$$

holds. The statement `t=-2` says that we are at the end of execution and `k=i` says that `i` tasks have been performed. Then we have a list of all possible path lengths. We do not however get any information on average path lengths or number of distinct paths of each length. Furthermore we do not get any information about what each path looks like. We can, however, try to verify the negation,

$$\neg EF(t = -2 \wedge k = i) \tag{4}$$

and the counterexamples, provided by the model checker, would give us one example of each possible path length.

Is there a path between two given tasks, i and j? Then we verify that the formula

$$EF((\neg r \wedge t = i) \wedge EF(\neg r \wedge t = j)) \tag{5}$$

holds. The statement $(\neg r \wedge t = i)$ says that task i is being performed and similarly $(\neg r \wedge t = j)$ says that task j is being performed.

3.5.2 Properties of linear work processes

Let us now look at some properties that are particularly useful, when working with linear work process.

Successor relations In Figure 1 we can see the control flow through the process in Example 3. This picture can give the designer valuable information about how the process works. Creating this picture manually is time consuming and error prone. Here, model checking can help. We can ask the model checker whether it is possible that task j follows immediately after task i. In CTL it goes like this:

$$EF((\neg r \wedge t = i) \wedge EX(E[rU(\neg r \wedge t = j)])) \tag{6}$$

The part $(\neg r \wedge t = i)$ means that task i is being performed. The whole formula says hence that there exists a computation path, such that after task i has been performed, the routing variable is set until task j is performed. In other words, there are no tasks performed between the performing of tasks i and j. We can check formula 6 for each pair (i, j) such that $0 \leq i < j \leq number\_of\_tasks - 1$. Additionally we need to know which tasks can be the final tasks of the work process. We get this information by checking which tasks have the state "-2" as its successor. Or in terms of CTL, we check for all tasks:

$$EF((\neg r \wedge t = i) \wedge EX(E[rU(\neg r \wedge t = -2)])) \tag{7}$$

Then we have enough information to draw the successor relationships in Figure 1. We do not get any information from the model checker about the constraints associated with the edges.

Although perhaps not as useful, the successor relation test can be adapted to graph work processes. In that case, one needs to consider that the successor relation might not be increasing on the task number, as it is in the case of linear work process.

3.5.3 Properties of graph work processes

Let us now look at some properties that are particularly useful, when working with graph work process.

Is an ending reached? We can verify that the process always comes to an end position, by verifying that the formula

$$AF(t = -2) \tag{8}$$

holds. The formula says that every path in the transition system, eventually has a node where the task identifier is -2. This test verifies that there do not exist cyclic runs in the process. This works indeed also for linear work processes, but there it makes little sense because it will always hold.

3.6 SMV

One or the most common verifier for CTL is SMV, symbolic model verifier, [10, 12]. SMV takes as input a program describing the model and CTL formulas.

The model checker returns a value 'true' if the formulas hold for all initial states. If a formula does not hold, SMV returns a counter example, a trace showing why the formula fails to hold.

SMV programs are built up from modules (procedures) and one of them must be called `main`. Modules declare signals (variables) and assign values to them. SMV offers three simple data types, boolean, enumerated and subrange. The following code contains examples of the data types respectively.

```
foo : boolean;
bar : {yes,no,maybe};
count : 0..7;
```

Furthermore SMV offers arrays and structs. The signals are infinite sequences of values of a given type. Signals are assigned values through the `init` and `next` commands. The command `init` is used to define the initial value of the sequence. The command `next` is used to iteratively define the next value of the sequence. As an example we can define a signal x to be a running sum of the values of signal y

```
init(x) := 0;
next(x) := x + y;
```

Signals can be assigned values in a non-deterministic fashion. The assignment

```
next{bar} := {yes,no};
```

assigns the values 'yes' and 'no' to 'bar' non-deterministically.

In the following SMV programs, we make use of the conditionals `if` and `case`; and the constructor loop `for`.

Execution of SMV modules is either synchronous or asynchronous. In the former case, the modules execute in parallel. In the latter, they run at different 'speeds' and interleave arbitrarily.

3.7 Model checking in SMV

We illustrate how SMV can be used to reason about work processes, by means of two detailed examples. The first example shows how to code a linear work process. The second example shows how to code a deterministic work process.

The SMV programs are split up into four parts and looks something like this:

```
module main(){
  Signal definitions ...
  Signal initialization ...
  Description of the transition system ...
  Specifications ...
}
```

We will describe each part in detail for both examples. In the following we will be discussing two kinds of variables. First of all there are variables within the work process. I.e. variables updated as tasks are performed, such as answers to questions. Those variables will be referred to as *process variables*. Secondly, there are variables used directly by the SMV program. Those will simply be referred to as *signals*.

3.7.1 Coding linear work process

As an example of a linear work process we will use Example 2. The structure of that work process was like this:
$Q_0(x_0)$
if ($x_0 = yes$) then $Q_1(x_1)$
if ($x_0 = yes$) then $Q_2(x_2)$
$Q_3(x_3)$
if ($x_3 = yes$) then $Q_4(x_4)$
$Q_5(x_5)$
We have replaced the actual question with an identifier Q_i and replaced the answer choices with a variable name x_i. The only process variables relevant to routing are x_0 and x_3. Both variables have domain {yes,no}. Let us now look at each part of the program.

Signal definitions We begin with signal definitions

```
task : -2..6;
route : boolean;
vars : array 0..1;
vars[0] : {undef,no,yes};
vars[1] : {undef,no,yes};
performed : 0..6;
```

The signals `task` and `route` are replacements of the variables t and r, mentioned in section 3.3. The signal `task` can take the values -2 and -1, representing end and beginning of execution respectively; the values 0 through 5 to represent the six tasks; and 6 is used to represent overflow after the last task has been performed (see further page 20). The signal `route` is a boolean signal to indicate how the task is handled. A value of 1 means that routing is being determined, and value of 0 means that the task is being performed. The process variables x_0 and x_3, used previously, have been transformed into the array `vars` of length 2, used to keep updates of the process variables that are relevant to routing. The variables' domain has been extended to include the value 'undef'. This value is used to identify that the process variable has not yet been given a value. The signal `performed` is used for counting how many tasks have been performed.

Signal initialisation After defining the signals, they must be initialised

```
init(task) := -1;
init(route) := 1;
init(performed) := 0;
for(i=0;i<2;i=i+1){
  init(vars[i]) := undef;
}
```

The `task` signal is set to represent the beginning of the execution. The `route` variable is set to routing mode. The `performed` variable starts at 0 since no task has been performed. All process variables are initialised with the value undef. This is necessary because otherwise the model checker would non-deterministically choose one value from its domain as an initial one.

Description of the transition system Having initialised the signals, we are ready to describe the transition system

```
if(task~=-2){
  if(~route){
    next(performed) := performed + 1;
    next(task) := task + 1;
    next(route) := 1;
    case{
      task=0 : next(vars[0]) := {no,yes};
      task=3 : next(vars[1]) := {no,yes};
    }
  } else {
    case{
      task=-1 : next(task) := 0;
      task=0 : next(route) := 0;
      task=1 : {
        if (vars[0]=yes) next(route) := 0;
        else next(task) := task + 1;
      }
      task=2 : {
        if (vars[0]=yes) next(route) := 0;
        else next(task) := task + 1;
      }
      task=3 : next(route) := 0;
      task=4 : {
        if (vars[1]=yes) next(route) := 0;
        else next(task) := task+1;
      }
      task=5 : next(route) := 0;
      default : {
        next(task) := -2;
        next(route) := 0;
      }
    }
  }
}
```

The first conditional `if(task~=-2)`, meaning that `task` is not equal to `-2`, is to tell that nothing is to be done after the process has reached its end. If the end has not been reached we must distinguish between routing mode and task performing mode. This is done by the conditional `if(~route)`. In case of task performing, `~route`, we increase the counter of performed tasks by one; we increase the task signal by one and we switch over to routing mode by setting the `route` signal. If there there are process variables to be updated in this task, we non-deterministically choose one value from their domains, that is other than 'undef'. In case of routing, we do a case distinction on the task signal. If the signal is set at the beginning of the process, -1, we must set it to the first task, which is 0 in this case. If the task signal is at a task without a constraint in front of it, we go into task performing mode by setting `route` to 0 (see tasks 0, 3 and 5). If the task signal is at a constrained task, we must check whether the constraint holds. If it does, we switch to task performing mode, otherwise we increase the task signal by one but stay in routing mode (see tasks 1, 2 and 4). If the task signal is at a non-existing task (overflow), the control is caught by the `default` case. In that case we are at the end of the process and hence we set the task signal to `-2`.

Specifications Finally after describing the transition system, we list the specifications, that we want check whether hold in the model:

```
SPEC AF task=-2;
for(i=0;i<6;i=i+1){
  SPEC EF (~route & task=i);
}
for(i=0;i<6;i=i+1){
  SPEC AF (~route & task=i);
}
for(i=0;i<6;i=i+1){
 for(j=i+1;j<6;j=j+1){
   SPEC EF((~route & task=i) & EX(E[route U (~route & task=j)]));
 }
}
for(i=0;i<6;i++){
 SPEC EF((~route & task=i) & EX(E[route U (~route & task=-2)]));
}
for(i=0;i<7;i=i+1){
  SPEC EF (performed=i & task=-2);
}
```

First of all we check whether end is always reached, `AF task=-2`. We check whether each task can possibly be performed, `EF(~route & task=i)`. We check whether each task is always performed, `AF(~route & task=i)`. We check whether one task can follow another,

```
EF((~route & task=i) & EX(E[route U (~route & task=j)]))
EF((~route & task=i) & EX(E[route U (~route & task=-2)]))
```

Finally, we check possible route lengths, `EF(performed=i & task=-2)`.

Model checking Feeding this program to the SMV model checker gives no surprising results. All paths result in an ending positions. All tasks can be performed. Tasks 0, 3 and 5 are the only tasks always to be performed. And for l, $3 \leq l \leq 6$, there exists a path through the work process whose length is l. Hence the shortest path contains three tasks, and the longest path contains six tasks.

3.7.2 Coding a deterministic work process

Let us now look at how we code a deterministic work process. We use here the work process in Example 4. The structure of the work process was like this:
$Q_0(x_0)$
- if $(x_0 = yes)$ goto Q_1

- if $(x_0 = no)$ goto Q_3
$Q_1(x_1)$
- goto Q_2
$Q_2(x_2)$
- goto Q_3
$Q_3(x_3)$
- if $(x_3 = yes)$ goto Q_4
- if $(x_3 = no)$ goto Q_6
$Q_4(x_4)$
- if $(x_2 = yes$ and $x_3 = yes)$ goto Q_5
- if $(x_2 = no$ or $x_3 = no)$ goto Q_6
$Q_5(x_5)$
- goto Q_6
$Q_6(x_6)$
- goto end

We have replaced the actual questions with an identifier Q_i and the answer choices with a variable name x_i. The process variables relevant to the routing are x_0, x_2 and x_3. All those variables have domain {yes,no}. The program has the same structure as the former one.

Signal definitions Let us start with signal definitions

```
task : -2..6;
route : boolean;
vars : array 0..2;
vars[0] : {undef,no,yes};
vars[1] : {undef,no,yes};
vars[2] : {undef,no,yes};
performed : 0..7;
```

The `task` signal is defined as before except that there is no overflow. The signals `route` and `performed`; and the process variables are defined as before.

Signal initialisation The signals are initialised as before

```
init(task) := -1;
init(route) := 1;
for(i=0;i<3;i=i+1){
 init(vars[i]) := undef;
}
init(performed) := 0;
```

Description of the transition system The transition system is somewhat different from the previous one:

```
if(task~=-2){
 next(route) := ~route;
 if(~route){
  next(performed) := performed + 1;
  case{
   task=0 : next(vars[0]) := {no,yes};
```

```
   task=2 : next(vars[2]) := {no,yes};
   task=3 : next(vars[1]) := {no,yes};
  }
 }else{
  case{
   task=-1 : next(task) := 0;
   task=0 :
    if(vars[0]=yes) next(task) := 1;
    else if(vars[0]=no) next(task) := 3;
   task=1 : next(task) := 2;
   task=2 : next(task) := 3;
   task=3 :
    if(vars[1]=yes) next(task) := 4;
    else if(vars[1]=no) next(task) := 6;
   task=4 :
    if(vars[2]=yes & vars[1]=yes) next(task) := 5;
    else if(vars[2]=no | vars[1]=no) next(task) := 6;
   task=5 : next(task) := 6;
   task=6 : next(task) := -2;
  }
 }
}
```

The `route` signal alternates between 0 and 1 in each step. The `task` signal is not increased continuously as before. Previously routing was decided before a task was performed. I.e. for each task it was decided whether it should be performed. Here routing is decided after a task is performed. I.e. after a task is performed, it is decided which task is to be performed next, by evaluating the constraints on the current task's outgoing edges. Again we distinguish between routing mode and task performing mode. In case of task performing, the `performed` counter is increased by one; and non-deterministically a value is chosen for each process variable that is updated by the current task. In case of routing, we do a case distinction on the task signal. If the task signal is -1 we set it to the initial task, in this case as 0. If the task signal is at a task with constrained routing, the constraints are evaluated and the task signal is set to the appropriate task (see tasks 0, 3 and 4). If the task signal is at a task without constrained routing, the task signal is set to the appropriate task (see tasks 1, 2, 5 and 6). There is no default case because the routing to the end of the process is dealt with in the routing after performing the last task of the run. Note that the last task needs not be the same for different runs, there might be instructions to route to the end of the process after several tasks in the process.

Specifications The specifications part is the same as in the case of a linear work process, but with different indexes

```
SPEC AF task=-2;
for(i=0;i<7;i=i+1){
  SPEC EF (~route & task=i);
}
for(i=0;i<7;i=i+1){
```

```
  SPEC AF (~route & task=i);
}
for(i=0;i<7;i=i+1){
 for(j=i+1;j<7;j=j+1){
   SPEC EF((~route & task=i) & EX(E[route U (~route & task=j)]));
 }
}
for(i=0;i<7;i++){
 SPEC EF((~route & task=i) & EX(E[route U (~route & task=-2)]));
}
for(i=0;i<8;i=i+1){
  SPEC EF (performed=i & task=-2);
}
```

Model checking When feeding the program to the SMV model checker, there turns out to be a problem. The specification `AF(task=-2)` evaluates to false. The model checker provides us with the following counterexample.

State	route	task	var[0](x_0)	var[1](x_2)	var[2](x_3)	Next state
1	1	-1	undef	undef	undef	2
2	0	0	undef	undef	undef	3
3	1	0	no	undef	undef	4
4	0	3	no	undef	undef	5
5	1	3	no	undef	yes	6
6	0	4	no	undef	yes	7
7	1	4	no	undef	yes	6

In state 1 the routing signal in 1 and the task signal is -1, indicating that routing to the first task is being determined. In state 2 the routing signal is 0 and the task signal is 0, saying that task 0 is being performed. In state 3 routing is 1 and task is 0, hence routing after task 0 is being determined. The signal var[0] has been set to 'no' to reflect how the variable x_0 was updated when task 0 was performed. In state 4 routing is 0 and task is 3, saying that task 3 is being performed. In state 5 routing is 1 and task is 3, hence routing after task 3 is being determined. The signal var[2] has been set to 'yes' to reflect how the variable x_3 was updated when task 3 was performed. In state 6 routing is 0 and task is 4, saying that task 4 is being performed. In state 7 routing is 1 and task is 4, hence routing after task 4 is being determined. We can see that the process starts looping after performing task 4. Looking at the signal values we see that the error occurs when signal var[0] (x_0) is set to 'no', due to a negative answer to the question "Do you smoke?". In that case, the question "Have you tried to quit smoking?" is never asked, and hence the signal var[1] (x_2) is never updated. After task 4 we wanted to control the routing by the constraints `vars[2]=yes & vars[1]=yes` and `vars[2]=no | vars[1]=no`. Those constraints complement each other, if we assume that the signals have the domain $\{yes, no\}$. In this formulation we have added the value 'undef' to the domain, and the constraints are not complementary. To fix this kind of error, the designer must either update the undefined variable somewhere along the way, or change the routing to cope with this situation. One way to fix this particular error is to change the routing to

```
task=4 :
 if(vars[2]=yes & vars[1]=yes) next(task) := 5;
 else if(vars[0] = no | vars[2]=no | vars[1]=no) next(task) := 6;
```

When feeding the revised model to the model checker, the specification

```
AF(task=-2)
```

evaluates to true. The model checker verified that all tasks can be performed and tasks 0, 3 and 6 are the only tasks that are always performed. For each l, $3 \leq l \leq 7$, there exists a path through the work process whose length is l. Hence the shortest path contains three tasks and the longest path contains all seven tasks.

3.7.3 Extensions

Let us now see how to extend the above code to be able to do some additional checks.

Uninitialised variables We ran into trouble in Section 3.7.2 because we tried to use in a constraint a variable whose value was not defined. We can easily extend the two above programs to detect if the program is about to use a variable with undefined value. We introduce a boolean signal `var_not_init` and initialise it to zero. Each time we are about to evaluate a constraint, we first check for each variable, appearing in the constraint, whether it is initialised. If it is not initialised we set the signal `var_not_init`. As an example, the routing determination for task 4 in Section 3.7.2 would alter to

```
task=4 : {
 if(vars[1]=undef | vars[2]=undef){
  next(var_not_init) := 1;
  next(task) := -2;
 }
 if(vars[2]=yes & vars[1]=yes) next(task) := 5;
 else if(vars[2]=no | vars[1]=no) next(task) := 6;
}
```

To verify that uninitialised variable are never used we must verify the CTL formula

$$AG(\neg var\_not\_init) \tag{9}$$

by adding the following line to the specification part of the program

```
SPEC AG(~var_not_init);
```

The formula evaluates to true if and only if the variable `var_not_init` is never given the value `1`.

Routing error Routing errors might occur due to other reasons than missing initialisation of variables, for example due to a typo made by the work process designer. We can extend the above model of a deterministic work process to handle the general case where no routing is found. To this end, we introduce a new boolean signal, called `route_error`, and initialise it with value 0. Each time when determining routing we add the code

```
else {
 next(route_error) := 1;
 next(task) := -2;
}
```

to catch the case when non of the routing conditions is fulfilled. To verify that this is never the case we must verify the CTL formula

$$AG(\neg route\_error) \tag{10}$$

and add the following line to the specification

```
SPEC AG(~route_error);
```

The formula evaluates to true if and only if the variable `route_error` is never given the value `1`.

Ambiguous routing We can also extend the deterministic model to include checks for ambiguous routing, that is to check if more than one routing option is available. We introduce a new signal `amb_route` and initialise it to zero. If a constraint evaluates to true, when determining the routing of a specific task, we must ensure that the disjunction of all the other constraints, on edges leaving this task, does not evaluate to true. As an example, the routing determination for task 4 in Section 3.7.2 would be altered to

```
task=4 :
 if(vars[2]=yes & vars[1]=yes){
  if (vars[2]=no | vars[1]=no){
   next(amb_route) := 1;
   next(task) := -2;
  } else next(task) := 5;
 }else if(vars[2]=no | vars[1]=no) next(task) := 6;
```

To verify that ambiguous routing never occurs we must verify the CTL formula

$$AG(\neg amb\_route) \tag{11}$$

by adding the following line to the specification part of the program

```
SPEC AG(~amb_route);
```

The formula evaluates to true if and only if the variable `amb_route` is never given the value `1`.

3.8 Automatic SMV generation

While working on this thesis I created a program that automatically creates the SMV code. The program is written in Java and takes as input an XML description of a work process. It creates the SMV file according to the recipe in Section 3.7. It runs the SMV verifier and returns the results as an XML file. The program is run by the script `wp`

```
wp xmlfile [-e][-pp][-ap][-pl][-s][-uv][-re][-ar][-x spec_file]
Where:
  xmlfile      XML description of a work process
  -e           check ending
  -pp          check which tasks are performed
  -ap          check which tasks are always performed
  -pl          check which path lengths are possible
  -s           check successor relations
  -uv          check for uninitialised variables
  -re          check for routing error
  -ar          check for ambiguous routing
  -x spec_file check the specifications in file spec_file
```

3.9 Case studies

In the next three sections we will apply the methods, introduced in this section, on three different work processes.

In Section 4 we will look at a deterministic work process. We will look at how standard methods can be used to check the structural properties of the work process. We will take a special look at how the model checker deals with typos made by the work process designer.

In Section 5 we will look at a linear work process. We will look at how standard methods can be used to check the structural properties of the work process. We will take a special look at how we can use model checking to verify work process equivalence.

In Section 6 we will look at a classification work process. We will look at how we can verify that the classification works.

4 Case study - graph work process

This case study is devoted to a work process defined as a graph. We will take a look at the deterministic work process in Figure 6. The first column in the table lists the tasks and the second column lists routing instructions to be performed after each tasks. The routing instructions consist either of one definite GOTO command or a list of conditioned GOTO commands. The actual description of the task has been replaced with a task identifier Q_i. Only variables that are relevant to routing are listed in the work process. A task identified with $Q_j(x_i)$ updates variable x_i. We will see how this work process is coded in the SMV language and we see how the SMV model checker can be used to verify properties of the process.

4.1 Formulation in SMV

We formulate the work process as a transition system as described in section 3.3; the properties are formulated like in section 3.5; and the translation to SMV is performed according to the recipe in section 3.7.2; additionally we check for routing problems as described in Section 3.7.3. The resulting program is listed in Appendix A.1.

Task	Routing instructions
$Q_{1.1}(x_1)$	{IF($x_1 = yes$)GOTO $Q_{1.2}$} {IF($x_1 = no$)GOTO $Q_{1.7}$}
$Q_{1.2}(x_2)$	{IF($x_2 = yes$)GOTO $Q_{1.3}$} {IF($x_2 = no$)GOTO $Q_{1.7}$}
$Q_{1.3}$	GOTO $Q_{1.4}$
$Q_{1.4}$	GOTO $Q_{1.5}$
$Q_{1.5}$	GOTO $Q_{1.6}$
$Q_{1.6}$	GOTO $Q_{1.7}$
$Q_{1.7}$	GOTO $Q_{1.8}$
$Q_{1.8}$	GOTO $Q_{1.9}$
$Q_{1.9}$	GOTO $Q_{2.1}$
$Q_{2.1}(x_3)$	GOTO $Q_{2.2}$
$Q_{2.2}(x_4)$	{IF($x_3 = no$ and $x_4 = no$)GOTO $Q_{3.1}$} {IF($x_3 = yes$ or $x_4 = yes$)GOTO $Q_{2.3}$}
$Q_{2.3}$	GOTO $Q_{2.4}$
$Q_{2.4}$	GOTO $Q_{3.1}$
$Q_{3.1}(x_5)$	{IF($x_5 = yes$)GOTO $Q_{3.2}$} {IF($x_5 = no$)GOTO $Q_{3.6}$}
$Q_{3.2}(x_6)$	{IF($x_6 = yes$)GOTO $Q_{3.3}$} {IF($x_6 = no$)GOTO $Q_{3.6}$}
$Q_{3.3}$	GOTO $Q_{3.4}$
$Q_{3.4}$	GOTO $Q_{3.5}$
$Q_{3.5}$	GOTO $Q_{3.6}$
$Q_{3.6}$	GOTO $Q_{4.1}$
$Q_{4.1}(x_7)$	{IF($x_7 = yes$)GOTO $Q_{4.2}$} {IF($x_7 = no$)GOTO $Q_{4.7}$}
$Q_{4.2}(x_8)$	{IF($x_8 = yes$)GOTO $Q_{4.3}$} {IF($x_8 = no$)GOTO $Q_{4.7}$}
$Q_{4.3}$	GOTO $Q_{4.4}$
$Q_{4.4}$	GOTO $Q_{4.5}$
$Q_{4.5}$	GOTO $Q_{4.6}$
$Q_{4.6}$	GOTO $Q_{4.7}$
$Q_{4.7}$	GOTO $Q_{5.1}$
$Q_{5.1}(x_9)$	{IF($x_9 = yes$)GOTO $Q_{5.2}$} {IF($x_9 = no$)GOTO $Q_{6.1}$}
$Q_{5.2}$	GOTO $Q_{5.3}$
$Q_{5.3}$	GOTO $Q_{5.4}$
$Q_{5.4}$	GOTO $Q_{5.5}$
$Q_{5.5}$	GOTO $Q_{5.6}$
$Q_{5.6}$	GOTO $Q_{5.7}$
$Q_{5.7}$	GOTO $Q_{5.8}$
$Q_{5.8}$	GOTO $Q_{5.9}$
$Q_{5.9}$	GOTO $Q_{6.1}$
$Q_{6.1}(x_{10})$	{IF($x_{10} = yes$)GOTO $Q_{6.1.b}$} {IF($x_{10} = no$)GOTO $Q_{6.2}$}
$Q_{6.1.b}$	GOTO $Q_{6.2}$
$Q_{6.2}(x_{11})$	{IF($x_{11} = yes$)GOTO $Q_{6.2.b}$} {IF($x_{11} = no$)GOTO $Q_{6.3}$}
$Q_{6.2.b}$	GOTO $Q_{6.3}$
$Q_{6.3}(x_{12})$	{IF($x_{12} = yes$)GOTO $Q_{6.3.b}$} {IF($x_{12} = no$)GOTO $Q_{6.4}$}
$Q_{6.3.b}$	GOTO $Q_{6.4}$
$Q_{6.4}(x_{13})$	{IF($x_{13} = yes$)GOTO $Q_{6.4.b}$} {IF($x_{13} = no$)GOTO $Q_{6.5}$}
$Q_{6.4.b}$	GOTO $Q_{6.5}$
$Q_{6.5}$	GOTO $Q_{6.6}$
$Q_{6.6}$	GOTO $Q_{6.7}$
$Q_{6.7}(x_{14})$	{IF($x_{14} = yes$)GOTO $Q_{6.7.b}$} {IF($x_{14} = no$)GOTO END}
$Q_{6.7.b}$	GOTO END

Figure 6: Questionnaire used in a asthma research at The University hospital in Iceland

4.2 Checking properties

Having translated the work process, we can check which properties hold on the model of the work process. First of all we check the properties listed in Section 3.5; *is task t performed?* (equation 1), *is task t always performed?* (2), *is there a path of length i?* (3), *can task t_j succeed task t_i?* (6), *does the work process come to an end?* (8). We also check the properties of Section 3.7.3; *is there a routing error* (10), *it the routing ambiguous* (11).

4.3 Runtime results

Feeding the above mentioned program to the SMV model checker, we are able to verify that the work process has no structural errors. The job took the model checker less than 30 seconds. All tasks can be performed; the work

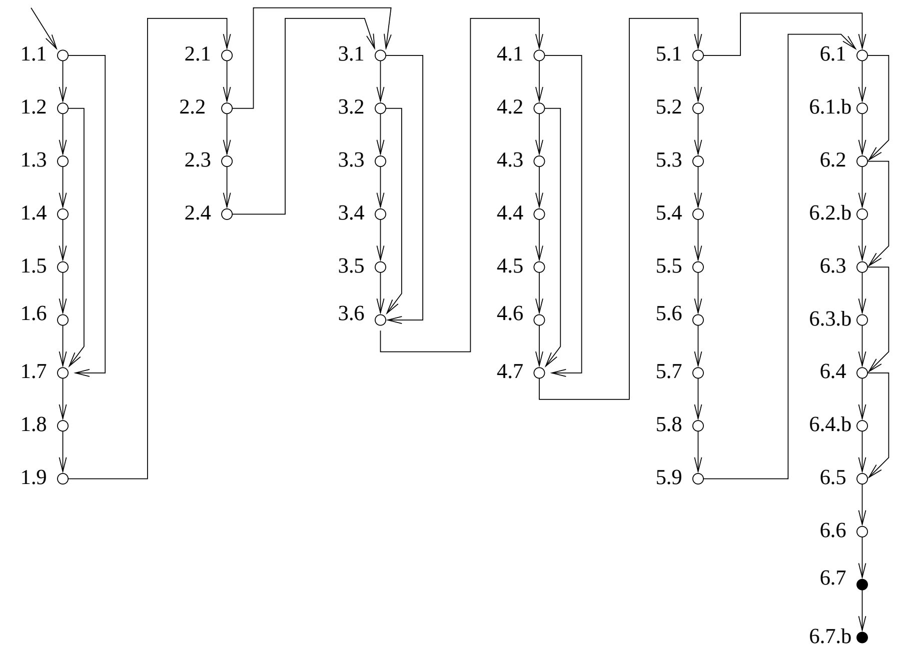

Figure 7: Control flow of the work process

process always comes to an end; the routing is neither erroneous nor ambiguous. Furthermore we get to know that tasks $Q_{1.1}$, $Q_{1.7}$, $Q_{1.8}$, $Q_{1.9}$, $Q_{2.1}$, $Q_{2.2}$, $Q_{3.1}$, $Q_{3.6}$, $Q_{4.1}$, $Q_{4.7}$, $Q_{5.1}$, $Q_{6.1}$, $Q_{6.2}$, $Q_{6.3}$, $Q_{6.4}$, $Q_{6.5}$, $Q_{6.6}$ and $Q_{6.7}$ are exactly the tasks that are always performed. The shortest path is 18 tasks; the longest one is 47 tasks; and furthermore for all integers l, $18 \leq l \leq 47$, there is a path of length l through the work process. From the successor relation tests we can create Figure 7. For each formula the model checker tries to verify, it returns output on the form

```
formula....................................................true
```

or

```
formula...................................................false
```

depending of whether the formula evaluated true or false. Hence we get lines like

```
(EF (((~route)&(task=0))&(EX (E ((route=1) U ((~route)&(task=1)))))))......true
(EF (((~route)&(task=0))&(EX (E ((route=1) U ((~route)&(task=2))))))).....false
```

saying that task 1 can follow task 0 but task 2 cannot. We collect all the formulas that evaluate to true and use it to manually draw the arrows in the figure. The black nodes in the figure represent nodes that can route to the end. Finding methods to automatically draw these kind of figures is far beyond the scope of this thesis.

Let us now take a look at the resource usage of the model checking. Table 1 displays the resource usage of various properties checks. In the top half of

Property	formulas	BDD nodes allocated	user time	system time	reachable states
Ending (8)	1	32874	0.53s	0.03s	2164
Route error (10)	1	31288	0.38s	0.04s	2802
Amb. routing (11)	1	26452	0.39s	0.01s	2284
Performed? (1)	47	35559	1.20s	0.02s	2545
Always perf. (2)	47	35554	13.25s	0.15s	2545
Path length (3)	48	228127	5.69s	0.09s	2245
Successor (6)	1128	35554	5.66s	0.15s	2545
Combinations					
8, 10, 11	3	37305	0.54s	0.04s	2961
8, 10, 11, 1, 2	97	50710	15.89s	0.02s	2885
8, 10, 11, 3	51	232889	5.89s	0.08s	2319
8, 10, 11, 6	1131	38212	6.25s	0.05s	2885
8, 10, 11, 1, 2, 3, 6	1273	194199	27.83s	0.43s	3010

Table 1: Resource usage

the table we see the checking cost of individual properties. In the bottom half we see the checking costs of several combinations of properties. We group *ending*, *routing error* and *ambiguous routing* together to make the most basic test package. We then look at how the resource usage is affected by adding of the other test to this basic package.

We can see that checking *ending*, *routing error* and *ambiguous routing* are the simplest checks, both with respect to number of BDD nodes and time. Each of those checks only consist of one formula and the formula is simple, only two connectives and one atom.

The checks of *which tasks can be performed*, *which tasks are always performed* and *the successor relations*; require BDD of similar size. Their time usage is however quite different. The first one taking the shortest time and the second one taking the longest time. The first and second check consist on the same number of formulas while the third one contains quite a few more.

Checking *possible path lengths* requires far bigger BDD than any other check. The time required is however similar to checking successor relations.

The running time of a combination of checks is approximately the sum of the time it took to do the checks individually. The size of BDD required for a combination was 2-30% larger than the BDD required to check the most complex of its components, except in the case of all checks. When checking all properties the BDD was 15% smaller than the BDD required to check only *path lengths*.

In the above model checking everything went well and no errors were detected. Unfortunately, this is not always the case. People do occasionally make mistakes, do typos etc. In order to take a look at how the model checker detects errors, let us introduce typos to the original questionnaire. We try to create realistic typos, in the sense that they are quite likely to be man-made. Most of them are such that they are likely to be results of careless use of "copy-paste"-ing.

Routing error Let us make a typo in the routing after task $Q_{1.2}$. We let both routing options have the same constraint $x_2 = yes$, that is, task $Q_{1.2}$ will be

$Q_{1.2}(x_2)$ $\{\text{IF}(x_2 = yes)\text{GOTO } Q_{1.3}\}$ $\{\text{IF}(x_2 = yes)\text{GOTO } Q_{1.7}\}$

We run the model checker again and make the same checks as before. The model checker returns that a routing error has occurred

```
(AG (~route_error))........................................false
```

and provides us with the following counter example.

State	route	task	route_error	var[0](x_1)	var[1](x_2)	Next state
1	1	-1	0	undef	undef	2
2	0	0	0	undef	undef	3
3	1	0	0	y	undef	4
4	0	1	0	y	undef	5
5	1	1	0	y	n	6
6	0	-2	1	y	n	6

The counter example describes a computation that resulted in a violation of the specification `AG(~route_error)`. In state 1, the `task` variable is `-1` indicating that we are at the beginning of a work process execution. In state 2, `route` is `0` and `task` is `0`, thus task `0` is being performed. In state 3, `route` is `1` and `task` is `0`, thus routing after task `0` is being determined. The variable `var[0]` has been set to `y` to indicate how it was updated when task `0` was performed. In state 4, `route` is `0` and `task` is `1`, thus task `1` is being performed. In state 5, `route` is `1` and `task` is `1`, thus routing after task `1` is being determined. The variable `var[1]` has been set to `n` to indicate how it was updated when task `1` was performed. In task 6, `route_error` is `1`, thus routing was not found in the previous state. Hence the counter example reveals that when assigning y to x_1 and n to x_2, no routing is found after performing task $Q_{1.2}$.

Ambiguous routing Let us now introduce a typo in the routing after task $Q_{2.2}$ We write 'or' instead of 'and' in the first constraint, that is, task $Q_{2.2}$ will be

$Q_{2.2}(x_4)$ $\{\text{IF}(x_3 = no \text{ or } x_4 = no)\text{GOTO } Q_{3.1}\}$ $\{\text{IF}(x_3 = yes \text{ or } x_4 = yes)\text{GOTO } Q_{2.3}\}$

We run the model checker again and make the same checks as before. The model checker returns that a routing error has occurred

```
(AG (~amb_route))..........................................false
```

and provides us with the following counter example.

State	route	task	amb_route	x_1	x_3	x_4	Next state
1	1	-1	0	undef	undef	undef	2
2	0	0	0	undef	undef	undef	3
3	1	0	0	n	undef	undef	4
4	0	6	0	n	undef	undef	5
5	1	6	0	n	undef	undef	6
6	0	7	0	n	undef	undef	7
7	1	7	0	n	undef	undef	8
8	0	8	0	n	undef	undef	9
9	1	8	0	n	undef	undef	10
10	0	9	0	n	undef	undef	11
11	1	9	0	n	n	undef	12
12	0	10	0	n	n	undef	13
13	1	10	0	n	n	y	14
14	0	-2	1	n	n	y	14

The counter example reveals that when assigning n to x_3 and y to x_4, the routing after task $Q_{2.2}$ is ambiguous.

Unreachable tasks Let us now introduce a typo in the routing after task $Q_{1.1}$ In the first routing option we make the process route to task $Q_{1.3}$ instead of task $Q_{1.2}$, that is, task $Q_{1.1}$ becomes

$$Q_{1.1}(x_1) \quad \{\text{IF}(x_1 = yes)\text{GOTO } Q_{1.3}\}\ \{\text{IF}(x_1 = no)\text{GOTO } Q_{1.7}\}$$

We run the model checker again and make the same checks as before. The model checker returns that the task $Q_{1.2}$ cannot be performed

```
(EF ((~route)&(task=1)))...................................false
```

but does not provide a counter example. Failing to prove existence of a path does not give rise to a counter example.

Cyclic runs Let us now introduce another typo in the routing after task $Q_{1.2}$. Suppose the questionnaire designer created that task by copying task $Q_{1.1}$, but forgot to change the routing destinations. Then, task $Q_{1.2}$ becomes

$$Q_{1.2}(x_2) \quad \{\text{IF}(x_2 = yes)\text{GOTO } Q_{1.2}\}\ \{\text{IF}(x_2 = no)\text{GOTO } Q_{1.7}\}$$

We run the model checker again and make the same checks as before. The model checker returns that the execution of the work process does not come to an end

```
(AF (task=-2))...............................................false
```

and gives the following counter example

State	route	task	var[0](x_1)	var[1](x_2)	Next state
1	1	-1	undef	undef	2
2	0	0	undef	undef	3
3	1	0	y	undef	4
4	0	1	y	undef	5
5	1	1	y	y	6
6	0	1	y	y	5

revealing that if the value y is assigned to both x_1 and x_2, we end up with a cyclic run after performing task $Q_{1.2}$.

4.4 Conclusions

By using model checking the designer of a questionnaire is able to verify that her design is correct from a structural point of view. In this case study we verified that the initial design had no structural flaws, all questions were reachable and the routing had no errors. Useful properties such as shortest and longest run were also retrieved. The model checking process was also able to detect probable typos.

It must be stressed that we are only checking structural properties. Although the questionnaire has an acceptable structure, its questions might be biased or badly phrased. There also is no guarantee that the questionnaire routing is free of typos. The only thing we know is that the typos do not lead to obvious structural errors.

The questionnaire in this case study is very simple and is not very challenging for the model checker. All process variables have a boolean domain, are only updated once in the process and each variable is only used once when routing is determined (x_1 is used only after task $Q_{1.1}$, x_2 is used only after task $Q_{1.2}$ etc.) The routing through the questionnaire is also very simple.

In this particular case the information about successor relations are perhaps not extremely helpful. The work process is already described as a graph and can be drawn from that description. Errors in routes and constraints are more likely to be caught by other checks. The checks could however point at an edge in the graph that is never used. The successor relation check will however prove useful in the next section.

5 Case study - linear work processes

In this case study we will consider the work process in Example 3. That is the following work process:
$Q_0(x_0)$
if (x_0=a or x_0=b or x_0=c) then $Q_1(x_1)$
if (x_0=d or x_0=e or x_0=f or x_0=g) then $Q_2(x_2)$
if (x_0=b or x_0=e or x_0=g) then $Q_3(x_3)$
if (x_0=d or x_0=e or x_0=f or x_0=g) then $Q_4(x_4)$
if (x_0=a or x_0=b or x_0=c) then $Q_5(x_5)$
if (x_0=d or x_0=e or x_0=f or x_0=g) then $Q_6(x_6)$
if (x_6=a) then $Q_7(x_7)$
if (x_6=a) then $Q_8(x_8)$
if (x_8=1) then $Q_9(x_9)$
if (x_6=b or x_6=c) then $Q_{10}(x_{10})$
if (x_0=g) then $Q_{11}(x_{11})$
if (x_0=d or x_0=e or x_0=f or x_0=g) then $Q_{12}(x_{12})$
if (x_{12}=1) then $Q_{13}(x_{13})$
$Q_{14}(x_{14})$
We will see how this work process can be coded in the SMV language and how we can use the SMV model checker to verify properties which hold for the process.

5.1 Formulation in SMV

We formulate the work process as a transition system as described in Section 3.3; the properties are formulated as explained in Section 3.5; and the translation to SMV is performed according to the recipe in Section 3.7. The resulting program can be seen in Appendix A.2.

5.2 Checking properties

Having translated the work process, we can check which properties hold on it. First of all, we check the properties listed in Section 3.5; *is task t performed?*, *is task t always performed?*, *is there a path of length i?*, *can task t_j succeed task t_i?*, *does the work process come to an end?* (Equations 1, 2, 3, 6, 8 respectively). Additionally we check properties that are designed specially for this particular work process.

We can verify that task 5 is performed if and only if task 1 is performed; but task 4 is performed if and only if task 1 is not performed. This is true if the formulas

$$AF((\neg r \wedge t = 1) \leftrightarrow AF(\neg r \wedge t = 5)) \tag{12}$$

and

$$AF((\neg r \wedge t = 1) \leftrightarrow \neg EF(\neg r \wedge t = 4)) \tag{13}$$

hold.

To make the path length question a bit more interesting, we might ask what are the different path length possibilities of paths containing task 11. Here we verify for each i, $0 \leq i \leq number\_of\_tasks$ that the formula

$$EF((\neg r \wedge t = 11) \wedge EF(t = -2 \wedge performed = i)) \tag{14}$$

holds. The first conjunct says that task 11 is performed and the second conjunct says that in the future there exists a point where the process has come to an end and i tasks were performed

5.3 Runtime results

Feeding the above mentioned program to the SMV model checker, we get to know that all tasks can be performed. Tasks 0 and 14 are the only one to be performed always. The shortest path is 4 tasks; the longest one is 12 tasks; indeed for all l, $4 \leq l \leq 12$, there exists a path through the work process whose length is l. Looking at the task successor relations, we can create a diagram showing the flow of control through the work process. A description of how to create this diagram can be seen in the previous case study (page 36). Comparing this diagram to Figure 1 we see that Figure 1 is not correct. An edge from task 8 to task 11 is missing. The figure was drawn manually and an error was made. Hence we can see that model checking can make a difference.

Manually comparing the figure and the results of the model checker is not the ideal solution. Furthermore we do not get any information on the conditions under which this additional arrow was drawn. It would be more sensible to code the figure as a graph work process, and use the model checker to verify that the two models are equivalent. In case of a missing arrow the model checker would

Property	test count	BDD nodes allocated	user time	system time	reachable states
Ending (8)	1	5855	0.11s	0.04s	452
Performed? (1)	15	10007	0.18s	0.03s	452
Always perf. (2)	15	10007	1.03s	0.03s	452
Path length (3)	16	11336	0.26s	0.01s	508
Successor (6)	105	10002	0.51s	0.04s	452
1 iff 5 (12)	1	5532	0.13s	0.01s	452
1 iff 4 (13)	1	5577	0.11s	0.03s	452
Path via 11 (14)	15	13432	0.41s	0.01s	508
Combinations					
8,1,2,3,6	152	21212	1.94s	0.05s	508
8,1,2,3,6,12,13,14	170	22659	2.04s	0.06s	508

Table 2: Resource usage

give a counter example, pointing at the location of the error. We will come back to this problem in Section 5.4.

The formulas 12 and 13 evaluated to true. The verifier said the shortest path going through task 11 was of length 8; the longest path is of length 12; and all path lengths there between can occur. The author of this thesis manually calculated the shortest path to be of length 7. His mistake was not to notice that every path going through task 11 also goes through task 3.

Let us now take a look at the resource usage of the model checking. Table 2 displays the resource usage of various properties checks. As before we split the table into two parts, one for single checks another for combinations of checks. On one hand we combine all the standard checks together and on the other combine the standard checks with the ones, tailor made for this work process

Of the standard checks (8 1, 2, 3, 6) the *check for ending* required the least number of BDD nodes allocated and also the shortest running time.

The checks of *which task can be performed*, *which tasks are always performed* and *successor relations*; all required similarly sized BDDs. As in the previous case study, the first one took the shortest time while the second one took the longest.

The check of *possible path lengths* required the largest BDD, but did not require much more time than checking *which task can be performed.*

Of the checks, tailor made for this work process (12, 13, 14), the checks for *task dependability* (12 and 13) required the least BDD nodes of all checks, standard and custom made. The check for *path length going through task 11* (14) required the largest BDD.

The running time of the combination of checks is approximately the sum of the running time of its components. The size of BDD required for a combination of test is 47-69% larger than the BDD of the most complex of its components.

5.4 Checking work process equivalence

Let us now look at how we can compare two work processes and check if they are equivalent. We continue the work in Section 5.3, where we wanted to check the equality of the work process Example 3 and the work process in Figure 1.

Until now, we have described the process within the module main. Now we must encode two processes, and we do that in two modules.

```
module graph(){
 ...
}
module linear(){
 ...
}
module main(){
 gra : graph();
 lin : linear();

 equality checking ...
}
```

The details of the modules `graph` and `linear` are skipped here, but can be seen in Appendix A.3. The module `graph` describes the transition system of the execution of the work process in Figure 1 and the module `linear` describes the transition system of the execution of the work process in Example 3. Both modules are implemented according to the recipe in section 3.7. The variables `gra` and `lin` are instances of the modules `graph` and `linear` respectively. When it comes to checking equality, we have several options. One is to check if the processes have identical successor relations

```
for(i=0;i<15;i=i+1){
 for(j=i+1;j<15;j=j+1){
  SPEC (EF((~gra.route & gra.task=i)
        & EX(E[gra.route=1 U (~gra.route & gra.task=j)]))
       <->(EF((~lin.route & lin.task=i)
        & EX(E[lin.route=1 U (~lin.route & lin.task=j)])));
 }
}
```

and if they have the same final tasks

```
for(i=0;i<15;i=i+1){
 SPEC (EF((~gra.route & gra.task=i)
       & EX(E[gra.route=1 U (~gra.route & gra.task=-2)]))
      <->(EF((~lin.route & lin.task=i)
       & EX(E[lin.route=1 U (~lin.route & lin.task=-2)])));
```

The first formula verifies for each $i < j$ that if j can possibly be successor of i in one process then it is also the case in the other, and vice versa. The second formula verifies that if i is a final task in one process, so is it in the other. See further description of formula 6 in Section 3.5.

This is actually enough to answer our question, whether a flow diagram correctly describes a linear work process. Running it on our examples we get that the two processes only disagree on an arrow from task 8 to task 11. The user time was 3.91s; the system time was 0.05s; there were 54491 BDD nodes allocated; and there were 1586 reachable states.

5.5 Conclusions

As in the previous case study we show how model checking can help a questionnaire designer to verify her design. Useful properties were verified, such as that all questions could be posed. We were able to extract information about longest and shortest path.

Although the questionnaire is not very complex, the model checker did on two occasions prove ‘manual reasoning’ to be wrong.

Here, checking successor relations proves to be of more use than for a graph work process. It provides enough information to draw up the flow of control through the work process. Although the drawing procedure is not fully automatic, this is an important step towards full automation and replaces the manual calculation of successor relations which proved to be error prone.

Being able to verify work process equivalence proved helpful in detecting errors in flow diagrams. This problem does however not exist if the flow diagram is drawn semi-automatically by the methods described above. Equivalence checking might however be useful in comparing work processes, created in different work process design tools.

As in the previous case study, the questionnaire that is analysed is quite simple. Variables are only updated once in the process. This questionnaire is however more complex than the previous one, in the way that some process variables are used more than once when routing is being determined (x_0 is used when determining routing for eight tasks).

6 Case study - classification

We will in this case study consider the following questionnaire, used to classify people with respect to their alcohol addiction. The questionnaire is taken from the web site netdoctor.co.uk [13].

1. Has anyone ever expressed concern about your drinking habits or suggested that you cut down?
   (a) No, never
   (b) Yes, but more than a year ago
   (c) Yes, during this year
2. Have you been responsible for accidents where you or others have been hurt, or could have been hurt, because you had been drinking?
   (a) Yes, during this year
   (b) Yes, but more than a year ago
   (c) NO, never
3. Do you ever feel guilty or have a bad conscience after you have been drinking?
   (a) No
   (b) Yes, but not every month
   (c) Yes, every month
   (d) Yes, every week
   (e) Yes, almost every day or daily
4. Do you ever find that you cannot stop drinking once you have started?

(a) Yes, almost every day or daily
(b) Yes, every week
(c) Yes, every month
(d) Yes, but not every month
(e) No

5. Do you ever find that you cannot live up to other people's expectations after you have been drinking?
   (a) No
   (b) Yes, but not every month
   (c) Yes, every month
   (d) Yes, every week
   (e) Yes, almost every day or daily
6. After a night of drinking, do you start the next morning with a drink?
   (a) Yes, almost every day or daily
   (b) Yes, every week
   (c) Yes, every month
   (d) Yes, but not every month
   (e) No
7. Do you ever experience black-outs after drinking?
   (a) No
   (b) Yes, but not every month
   (c) Yes, every month
   (d) Yes, every month
   (e) Yes, almost every day or daily
8. How often do you drink beer, wine or spirits?
   (a) Never
   (b) Once a month at most
   (c) 2-4 times a month
   (d) 2-3 times a week
   (e) 4 times a week or more
9. When you drink, how many drinks do you have in one day?
   (a) 1-2
   (b) 3-4
   (c) 5-6
   (d) 7-9
   (e) 10 or more
10. How often do you drink more than six units of alcohol in one day?
    (a) Never
    (b) Once a month at most

(c) 2-4 times a month

(d) 2-3 times a week

(e) 4 times a week or more

After a user has filled in the questionnaire, he is classified with respect to how likely he is to have alcoholic addiction. The classification is based on the answers given by the user. The classifier uses both the answer choices and a score variable. Each answer option adds a certain amount to the user's score. As an example option (e) in question 3 adds four points to the score, while option (a) adds no points to the score.

The users are classified into six classes. Each class is identified by a constraint. All runs that make a constraint true, are said to belong to its class. In this case study we will verify that the classification works, meaning that the following three conditions hold.

We want to make sure that there are no superfluous classes. I.e. that for each class, there exists at least one run trough the work process that belongs to the class. Hence we check for each class constraint c the formula

$$EF(t = -2 \wedge c) \tag{15}$$

which is true if and only if at the end of some run through the work process the constraint c holds. Hence the run belongs to the class identified by c.

We want to make sure that the classification is total. I.e. that all runs belong to some classification. Hence if $c_1, \ldots, c_6$ are the class constraints, we check the formula

$$AF(t = -2 \wedge (c_1 \vee \ldots \vee c_6)) \tag{16}$$

which is true if and only if at the end of each run, at least one of the class constraints holds. Hence each run belongs to one of the classes.

We also want to make sure that the classification is unique. I.e. that each run only belongs to one class. Hence we check the formula

$$AF\big(t = -2 \wedge \neg\big(\bigvee_{1 \leq i < j \leq 6} (c_i \wedge c_j)\big)\big) \tag{17}$$

which is true if and only if and only if at the end of each run, only one of the class constraints holds. Hence each run only belongs to one class.

6.1 Formulation in SMV

The transition system is coded according to the description in Section 3.7.1. The scoring feature is realized by adding (automatically) performed tasks to the work process. As an example, we add two tasks hidden between task 1 and task 2. One is performed if the answer to question 1 was (b) and it updates the score variable by adding two points to it. The second task is performed if the answer to question 1 was (c) and it updates the score variable by adding four points to it.

Since this is a linear work process, all the checks of structural properties are applicable. In this case study we will however concentrate on verifying classification properties. I.e. properties 15, 16 and 17 above. The resulting SMV program can be found in Appendix A.4.

6.2 Runtime results

Feeding the program to the SMV model checker reveals that the classification works. I.e. it is total, unique and there are no superfluous classes. The user time was 2.16s; the system time was 0.05s; 46925 BDD nodes were allocated; and there were 4915 nodes in the transition system.

6.3 Conclusions

We were able to verify that the work process classification was without errors.I.e. we verified that the classification was total, unique and had no superfluous classes. We do not however say anything about whether the classification is sociologically correct, that is we do not try to verify that the classification correctly classifies users with respect to their drinking behaviour.

For some applications, the requirements of classification to be total and unique, is likely to be dropped. For example in a disease diagnostics system, one wants to classify the patient to belong to several classes, representing different possible diseases. It is also possible that the patient has no disease symptoms at all or even symptoms of an unknown disease, leaving the patient unclassified.

7 Conclusions

Using model checking it is possible to check standard structural properties of work processes, such as "Is an ending always reached?", "Is the routing ambiguous?", "How are the successor relations?". Using a standard language like CTL for coding the properties, gives the designer the opportunity to do non-standard tests for a specific questionnaire, such as, "Do all people who answer question 5 also answer question 12?", "What is the longest path going through question 3 but does not go through question 7?", or any other property expressible in CTL.

We are able to verify if two work processes are equivalent, with respect to the order in which they perform their tasks. We are also able to verify if a given classification is a total, unique classification of the runs and contains no superfluous classes.

The tools introduced in this thesis are very customisable. But to be able to use the full power of the tools one must know the basics of CTL and model checking. This knowledge is not to be expected to be in possession of work process designers. Therefore these tools are more likely to be used to extend existing work process and questionnaire design tools. Those tools could include the standard checks mentioned in this thesis. They could also include some high level specification generator to create customised specifications. Those generators would typically be able to translate a fragment of a natural language into a fragment of CTL. Holt et. al. [6] have created tools for translating into CTL, hardware verification specifications written in English.

None of the work process examples mentioned in this thesis were very complex. The question of scalability hence remains unanswered. The complexity is affected by the size of the model and the size of the formulas being checked. For the structural checks and work process equivalence, when the work processes get more complex their models get larger but the size of the formulas stays the same. It is hence expected that the resource usage increases, but not dramatically. For the classification it is another story. As the work processes get more complex its classification is likely to get more complex. Hence the size of the model increases and the size of the formulas increases as well. We can therefore expect a blowup in resource usage.

One of the most important usage of the tools introduced in this thesis is to draw the control flow of linear work processes. We did not address the problem of turning the output of the model checking, the edges of an execution graph, into a picture. This topic is indeed outside the scope of this thesis but some research has been done on it within graph theory, for example in [14].

While working on this thesis I tried to contact various people that are involved in creating complex questionnaires, in order to get interesting examples for my case studies. Sadly I did not get many responses. I believe that the lack of interest was caused the fact that people did not realise the power of my tools, rather than that my tools are not applicable.

There are a number of problems to address in future work on this topic. First of all the tools need to be tested on a larger set of work processes. Both to test scalability and to realise if there are additional features needed. To make this tool more user friendly, work must be done on a natural language front-end for the input and a better visualisation of output.

References

[1] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: An industrial case study. In *Proceedings of ICSE'2002*, pages 431–441, 2002.

[2] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[3] F.Moller and P. Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://www.dcs.ed.ac.uk/home/cwb/.

[4] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification*, pages 379–394, Ottawa, Canada, 1990.

[5] P. Godefroid. Verisoft:A Tool for the Automatic Analysis of Concurrent Reactive Software. In *Proceedings of the 9th Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 476–479. Springer-Verlag, 1997.

[6] A. Holt, E. Klein, and C. Grover. Natural language for hardware verification: semantic interpretation and model checking. In *ICoS-1: Inference in Computational Semantics*, pages 133–137. Institute for Logic, Language and Computation, University of Amsterdam, 1999.

[7] G. Holzmann. The Spin Model Checker. In *IEEE Transactions on Software Engineering*, volume 23, pages 279–295. May 1997.

[8] M.R. Huth and M.D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2000.

[9] R. Kaivola and K. Kohatsu. Proof Engineering in the Large:Formal Verification of Pentium® 4 Floating-Point Divider. In *Proceedings of CHARME2001*, volume 2144 of *Lecture Notes in Computer Science.* Springer-Verlag, 2001.

[10] K.L.McMillan. *Getting started with SMV.* Cadence Berkeley Labs, Cadence Design Systems, 1998.

[11] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[12] K.L. McMillan. *The SMV language.* Cadence Berkeley Labs, Cadence Design Systems, 1998.

[13] NETDOCTOR.CO.UK. Dou you drink too much? Available at http://www.netdoctor.co.uk/testyourself/alcoholdrinking.asp.

[14] S. Rahman. *Efficient Algorithms for Drawing Planar Graphs.* PhD thesis, Graduate School of Information Sciences, Tohoku University, 1999.

[15] T. Schäfer, A. Knapp, and S. Merz. Model checking uml state machines and collaborations. In Scott D. Stoller and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[16] STEMWIJZER.NL. Tool to help people finding the political party that fits their oppinions. Available online at http://www.stemwijzer.nl.

[17] S. Sudman and N. Bradburn. *Asking Questions: A Practical Guide to Questionnaire Design.* Jossey-Bass, San Francisco, 1982.

[18] Survey design, questionnaire design tips. Available online from: http://www.surveysystem.com/sdesign.htm.

A SMV examples

A.1 Graph work process

The following SMV file is used to model check a graph work process. 4

```
module main(){
task : -2..46;
route : boolean;
vars : array 0..13;
vars[0] : {undef,n,y};
vars[1] : {undef,n,y};
vars[2] : {undef,n,y};
vars[3] : {undef,n,y};
vars[4] : {undef,n,y};
vars[5] : {undef,n,y};
vars[6] : {undef,n,y};
vars[7] : {undef,n,y};
vars[8] : {undef,n,y};
vars[9] : {undef,n,y};
vars[10] : {undef,n,y};
vars[11] : {undef,n,y};
vars[12] : {undef,n,y};
vars[13] : {undef,n,y};
performed : 0..47;
route_error : boolean;
amb_route : boolean;
init(task) := -1;
init(route) :=1;
for(i=0;i<14;i=i+1){
init(vars[i]) := undef;
}
init(performed) :=0;
init(route_error) := 0;
init(amb_route) := 0;
if(task~=-2){
 next(route) := ~route;
 if(~route){
  next(performed) := performed + 1;
  case{
   task=0 : next(vars[0]) := {n,y};
   task=1 : next(vars[1]) := {n,y};
   task=9 : next(vars[2]) := {n,y};
   task=10 : next(vars[3]) := {n,y};
   task=13 : next(vars[4]) := {n,y};
   task=14 : next(vars[5]) := {n,y};
   task=19 : next(vars[6]) := {n,y};
   task=20 : next(vars[7]) := {n,y};
   task=26 : next(vars[8]) := {n,y};
   task=35 : next(vars[9]) := {n,y};
   task=37 : next(vars[10]) := {n,y};
   task=39 : next(vars[11]) := {n,y};
   task=41 : next(vars[12]) := {n,y};
   task=45 : next(vars[13]) := {n,y};
  }
```

```
}else{
 case{
  task=-1 : next(task) := 0;
  task=0 :
   if(vars[0]=y){
    if(vars[0]=n){
     next(amb_route) := 1;
     next(task) := -2;
    }else next(task) := 1;
   }
   else if(vars[0]=n)
    next(task) := 6;
   else{
    next(route_error) := 1;
    next(task) := -2;
   }
  task=1 :
   if(vars[1]=y){
    if(vars[1]=n){
     next(amb_route) := 1;
     next(task) := -2;
    }else next(task) := 2;
   }
   else if(vars[1]=n)
    next(task) := 6;
   else{
    next(route_error) := 1;
    next(task) := -2;
   }
  task=2 : next(task) := 3;
  task=3 : next(task) := 4;
  task=4 : next(task) := 5;
  task=5 : next(task) := 6;
  task=6 : next(task) := 7;
  task=7 : next(task) := 8;
  task=8 : next(task) := 9;
  task=9 : next(task) := 10;
  task=10 :
   if(vars[2]=n & vars[3]=n){
    if(vars[2]=y | vars[3]=y){
     next(amb_route) := 1;
     next(task) := -2;
    }else next(task) := 13;
   }
   else if(vars[2]=y | vars[3]=y)
    next(task) := 11;
   else{
    next(route_error) := 1;
    next(task) := -2;
   }
  task=11 : next(task) := 12;
  task=12 : next(task) := 13;
  task=13 :
   if(vars[4]=y){
```

```
  if(vars[4]=n){
   next(amb_route) := 1;
   next(task) := -2;
  }else next(task) := 14;
 }
 else if(vars[4]=n)
  next(task) := 18;
 else{
  next(route_error) := 1;
  next(task) := -2;
 }
task=14 :
 if(vars[5]=y){
  if(vars[5]=n){
   next(amb_route) := 1;
   next(task) := -2;
  }else next(task) := 15;
 }
 else if(vars[5]=n)
  next(task) := 18;
 else{
  next(route_error) := 1;
  next(task) := -2;
 }
task=15 : next(task) := 16;
task=16 : next(task) := 17;
task=17 : next(task) := 18;
task=18 : next(task) := 19;
task=19 :
 if(vars[6]=y){
  if(vars[6]=n){
   next(amb_route) := 1;
   next(task) := -2;
  }else next(task) := 20;
 }
 else if(vars[6]=n)
  next(task) := 25;
 else{
  next(route_error) := 1;
  next(task) := -2;
 }
task=20 :
 if(vars[7]=y){
  if(vars[7]=n){
   next(amb_route) := 1;
   next(task) := -2;
  }else next(task) := 21;
 }
 else if(vars[7]=n)
  next(task) := 25;
 else{
  next(route_error) := 1;
  next(task) := -2;
 }
```

```
task=21 : next(task) := 22;
task=22 : next(task) := 23;
task=23 : next(task) := 24;
task=24 : next(task) := 25;
task=25 : next(task) := 26;
task=26 :
 if(vars[8]=y){
  if(vars[8]=n){
   next(amb_route) := 1;
   next(task) := -2;
  }else next(task) := 27;
 }
 else if(vars[8]=n)
  next(task) := 35;
 else{
  next(route_error) := 1;
  next(task) := -2;
 }
task=27 : next(task) := 28;
task=28 : next(task) := 29;
task=29 : next(task) := 30;
task=30 : next(task) := 31;
task=31 : next(task) := 32;
task=32 : next(task) := 33;
task=33 : next(task) := 34;
task=34 : next(task) := 35;
task=35 :
 if(vars[9]=y){
  if(vars[9]=n){
   next(amb_route) := 1;
   next(task) := -2;
  }else next(task) := 36;
 }
 else if(vars[9]=n)
  next(task) := 37;
 else{
  next(route_error) := 1;
  next(task) := -2;
 }
task=36 : next(task) := 37;
task=37 :
 if(vars[10]=y){
  if(vars[10]=n){
   next(amb_route) := 1;
   next(task) := -2;
  }else next(task) := 38;
 }
 else if(vars[10]=n)
  next(task) := 39;
 else{
  next(route_error) := 1;
  next(task) := -2;
 }
task=38 : next(task) := 39;
```

```
   task=39 :
    if(vars[11]=y){
     if(vars[11]=n){
      next(amb_route) := 1;
      next(task) := -2;
     }else next(task) := 40;
    }
    else if(vars[11]=n)
     next(task) := 41;
    else{
     next(route_error) := 1;
     next(task) := -2;
    }
   task=40 : next(task) := 41;
   task=41 :
    if(vars[12]=y){
     if(vars[12]=n){
      next(amb_route) := 1;
      next(task) := -2;
     }else next(task) := 42;
    }
    else if(vars[12]=n)
     next(task) := 43;
    else{
     next(route_error) := 1;
     next(task) := -2;
    }
   task=42 : next(task) := 43;
   task=43 : next(task) := 44;
   task=44 : next(task) := 45;
   task=45 :
    if(vars[13]=y){
     if(vars[13]=n){
      next(amb_route) := 1;
      next(task) := -2;
     }else next(task) := 46;
    }
    else if(vars[13]=n)
     next(task) := -2;
    else{
     next(route_error) := 1;
     next(task) := -2;
    }
   task=46 : next(task) := -2;
  }
 }
}
SPEC AF task=-2;
SPEC AG ~route_error;
for(i=0;i<47;i=i+1){
 SPEC EF (~route & task=i);
}
for(i=0;i<47;i=i+1){
 SPEC AF (~route & task=i);
```

```
}
for(i=0;i<47;i=i+1){
 for(j=i+1;j<47;j=j+1){
  SPEC EF((~route & task=i) & EX(E[route U (~route & task=j)]));
 }
}
for(i=0;i<47;i=i+1){
 SPEC EF((~route & task=i) & EX(E[route U (~route & task=-2)]));
}
for(i=0;i<48;i=i+1){
 SPEC EF (performed=i & task=-2);
}
SPEC AG ~amb_route;
}
```

A.2 Linear work process

This is the smv file of the work process analysed in section 5

```
module main(){
task : -2..15;
vars : array 0..3;
vars[0] : {b,a,undef,g,f,e,d,c};
vars[1] : {b,a,undef,f,e,d,c};
vars[2] : {undef,n,y};
vars[3] : {undef,n,y};
route : boolean;
performed : 0..15;
init(task) := -1;
init(route) :=1;
for(i=0;i<4;i=i+1){
 init(vars[i]) := undef;
}
init(performed) :=0;
if(task~=-2){
 if(~route){
  next(performed) := performed + 1;
  next(task) := task +1;
  next(route) := 1;
  case{
   task=0 : next(vars[0]) := {b,a,g,f,e,d,c};
   task=6 : next(vars[1]) := {b,a,f,e,d,c};
   task=8 : next(vars[2]) := {n,y};
   task=12 : next(vars[3]) := {n,y};
  } }else{
  case{
   task=-1 : next(task) := 0;
   task=0 : next(route) := 0;
   task=1 :
    if(vars[0]=a | vars[0]=b | vars[0]=c)next(route) := 0;
    else next(task) := task+1;
   task=2 :
    if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)next(route) := 0;
    else next(task) := task+1;
```

```
   task=3 :
    if(vars[0]=b | vars[0]=e | vars[0]=g)next(route) := 0;
    else next(task) := task+1;
   task=4 :
    if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)next(route) := 0;
    else next(task) := task+1;
   task=5 :
    if(vars[0]=a | vars[0]=b | vars[0]=c)next(route) := 0;
    else next(task) := task+1;
   task=6 :
    if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)next(route) := 0;
    else next(task) := task+1;
   task=7 :
    if(vars[1]=a)next(route) := 0;
    else next(task) := task+1;
   task=8 :
    if(vars[1]=a)next(route) := 0;
    else next(task) := task+1;
   task=9 :
    if(vars[2]=y)next(route) := 0;
    else next(task) := task+1;
   task=10 :
    if(vars[1]=b | vars[1]=c)next(route) := 0;
    else next(task) := task+1;
   task=11 :
    if(vars[0]=g)next(route) := 0;
    else next(task) := task+1;
   task=12 :
    if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)next(route) := 0;
    else next(task) := task+1;
   task=13 :
    if(vars[3]=y)next(route) := 0;
    else next(task) := task+1;
   task=14 : next(route) := 0;
   default : next(task) := -2;
  }
 }
}
SPEC AF task=-2;
for(i=0;i<15;i=i+1){
 SPEC EF (~route & task=i);
}
for(i=0;i<15;i=i+1){
 SPEC AF (~route & task=i);
}
for(i=0;i<15;i=i+1){
 for(j=i+1;j<15;j=j+1){
  SPEC EF((~route & task=i) & EX(E[route=1 U (~route & task=j)]));
 }
}
for(i=0;i<16;i=i+1){
 SPEC EF (performed=i & task=-2);
}
SPEC AF((~route & task=1)<->~(EF(~route & task=4)));
```

```
SPEC AF((~route & task=1)<->AF(~route & task=5));
for(i=1;i<16;i=i+1){
 SPEC EF((~route & task=11) & EF(performed=i & task=-2));
}
}
```

A.3 Equivalence checking

This is the smv file of the work process equivalence analysed in section 5.4

```
module graph(){
task : -2..14;
route : boolean;
vars : array 0..3;
vars[0] : {b,a,undef,g,f,e,d,c};
vars[1] : {b,a,undef,f,e,d,c};
vars[2] : {undef,n,y};
vars[3] : {undef,n,y};

init(task) := -1;
init(route) := 1;
for(i=0;i<4;i=i+1){
 init(vars[i]) := undef;
}

if(task~=-2){
 next(route) := ~route;
 if(~route){
  case{
   task=0 : next(vars[0]) := {b,a,g,f,e,d,c};
   task=6 : next(vars[1]) := {b,a,f,e,d,c};
   task=8 : next(vars[2]) := {n,y};
   task=12 : next(vars[3]) := {n,y};
  }
 }else{
  case{
   task=-1 : next(task) := 0;
   task=0 :
    if(vars[0]=a | vars[0]=b | vars[0]=c)
     next(task) := 1;
    else if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)
     next(task) := 2;
   task=1 :
    if(vars[0]=b | vars[0]=e | vars[0]=g)
     next(task) := 3;
    else if(~(vars[0]=b | vars[0]=e |vars[0]=g))
     next(task) := 5;
   task=2 :
    if(vars[0]=b | vars[0]=e | vars[0]=g)
     next(task) := 3;
    else if(~(vars[0]=b | vars[0]=e | vars[0]=g))
     next(task) := 4;
   task=3 :
    if(vars[0]=a | vars[0]=b | vars[0]=c)
```

```
     next(task) := 5;
    else if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)
     next(task) := 4;
   task=4 : next(task) := 6;
   task=5 : next(task) := 14;
   task=6 :
    if(vars[1]=a)
     next(task) := 7;
    else if(vars[1]=b | vars[1]=c)
     next(task) := 10;
    else if((vars[1]=d | vars[1]=e | vars[1]=f) & vars[0]=g)
     next(task) := 11;
else if((vars[1]=d | vars[1]=e | vars[1]=f)
& (vars[0]=d | vars[0]=e | vars[0]=f))
     next(task) := 12;
   task=7 : next(task) := 8;
   task=8 :
    if(vars[2]=y)
     next(task) := 9;
    else if(vars[2]=n)
     next(task) := 12;
   task=9 :
    if(vars[0]=g)
     next(task) := 11;
    else if(~(vars[0]=g))
     next(task) := 12;
   task=10 :
    if(vars[0]=g)
     next(task) := 11;
    else if(~(vars[0]=g))
     next(task) := 12;
   task=11 : next(task) := 12;
   task=12 :
    if(vars[3]=y)
     next(task) := 13;
    else if(vars[3]=n)
     next(task) := 14;
   task=13 : next(task) := 14;
   task=14 : next(task) := -2;
  }
 }
}
}

module linear(){

task : -2..15;
route : boolean;
vars : array 0..3;
vars[0] : {b,a,undef,g,f,e,d,c};
vars[1] : {b,a,undef,f,e,d,c};
vars[2] : {undef,n,y};
vars[3] : {undef,n,y};
```

```
init(task) := -1;
init(route) :=1;
for(i=0;i<4;i=i+1){
 init(vars[i]) := undef;
}

if(task~=-2){
 if(~route){
  next(task) := task +1;
  next(route) := 1;
  case{
   task=0 : next(vars[0]) := {b,a,g,f,e,d,c};
   task=6 : next(vars[1]) := {b,a,f,e,d,c};
   task=8 : next(vars[2]) := {n,y};
   task=12 : next(vars[3]) := {n,y};
  } }else{
  case{
   task=-1 : next(task) := 0;
   task=0 : next(route) := 0;
   task=1 :
    if(vars[0]=a | vars[0]=b | vars[0]=c)
     next(route) := 0;
    else
     next(task) :=task+1;
   task=2 :
    if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)
     next(route) := 0;
    else
     next(task) := task+1;
   task=3 :
    if(vars[0]=b | vars[0]=e | vars[0]=g)
     next(route) := 0;
    else
     next(task) := task+1;
   task=4 :
    if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)
     next(route) := 0;
    else
     next(task) := task+1;
   task=5 :
    if(vars[0]=a | vars[0]=b | vars[0]=c)
     next(route) := 0;
    else
     next(task) := task+1;
   task=6 :
    if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)
     next(route) := 0;
    else
     next(task) := task+1;
   task=7 :
    if(vars[1]=a)
     next(route) := 0;
    else
     next(task) := task+1;
```

```
   task=8 :
    if(vars[1]=a)
     next(route) := 0;
    else
     next(task) := task+1;
   task=9 :
    if(vars[2]=y)
     next(route) := 0;
    else
     next(task) := task+1;
   task=10 :
    if(vars[1]=b | vars[1]=c)
     next(route) := 0;
    else
     next(task) := task+1;
   task=11 :
    if(vars[0]=g)
     next(route) := 0;
    else
     next(task) := task+1;
   task=12 :
    if(vars[0]=d | vars[0]=e | vars[0]=f | vars[0]=g)
     next(route) := 0;
    else
     next(task) := task+1;
   task=13 :
    if(vars[3]=y)
     next(route) := 0;
    else
     next(task) := task+1;
   task=14 : next(route) := 0;
   default : next(task) := -2;
  }
 }
}
}
module main(){
gra : process graph();
lin : process linear();

for(i=0;i<15;i=i+1){
 for(j=i+1;j<15;j=j+1){
  SPEC (EF((~gra.route & gra.task=i)
         & EX(E[gra.route=1 U (~gra.route & gra.task=j)])))
        <->(EF((~lin.route & lin.task=i)
         & EX(E[lin.route=1 U (~lin.route & lin.task=j)])));
 }
}
for(i=0;i<15;i=i+1){
  SPEC (EF((~gra.route & gra.task=i)
         & EX(E[gra.route=1 U (~gra.route & gra.task=-2)])))
        <->(EF((~lin.route & lin.task=i)
         & EX(E[lin.route=1 U (~lin.route & lin.task=-2)])));
}
```

```
}
```

A.4 Classification

This is the SMV file of the work process classification analysed in Section 6

```
/*var_translation=
{ var0=0 var1=1 var2=2 var3=3 var4=4 var5=5 var6=6 var7=7 var8=8 var9=9
points=10 }*/
/*task_translation=
{ 0=0 101=1 102=2
1=3 110=4 111=5
2=6 121=7 122=8 123=9 124=10
3=11 131=12 132=13 133=14 134=15
4=16 141=17 142=18 143=19 144=20
5=21 151=22 152=23 153=24 154=25
6=26 161=27 162=28 163=29 164=30
7=31 171=32 172=33 173=34 174=35
8=36 181=37 182=38 183=39 184=40
9=41 191=42 192=43 193=44 194=45 }*/
module main(){
task : -2..46;
vars : array 0..10;
vars[0] : {b,a,c};
vars[1] : {b,a,c};
vars[2] : {b,a,e,d,c};
vars[3] : {b,a,e,d,c};
vars[4] : {b,a,e,d,c};
vars[5] : {b,a,e,d,c};
vars[6] : {b,a,e,d,c};
vars[7] : {b,a,e,d,c};
vars[8] : {b,a,e,d,c};
vars[9] : {b,a,e,d,c};
vars[10] : 0..40;
route : boolean;
performed : 0..46;
init(task) := -1;
init(route) :=1;
init(performed) :=0;
if(task~=-2){
 if(~route){
  next(performed) := performed + 1;
  next(task) := task +1;
  next(route) := 1;
  case{
   task=0 : next(vars[0]) := {b,a,c};
   task=1 : next(vars[10]) := vars[10] + 2;
   task=2 : next(vars[10]) := vars[10] + 4;
   task=3 : next(vars[1]) := {b,a,c};
   task=4 : next(vars[10]) := vars[10] + 4;
   task=5 : next(vars[10]) := vars[10] + 2;
   task=6 : next(vars[2]) := {b,a,e,d,c};
   task=7 : next(vars[10]) := vars[10] + 1;
```

```
 task=8 : next(vars[10]) := vars[10] + 2;
 task=9 : next(vars[10]) := vars[10] + 3;
 task=10 : next(vars[10]) := vars[10] + 4;
 task=11 : next(vars[3]) := {b,a,e,d,c};
 task=12 : next(vars[10]) := vars[10] + 4;
 task=13 : next(vars[10]) := vars[10] + 3;
 task=14 : next(vars[10]) := vars[10] + 2;
 task=15 : next(vars[10]) := vars[10] + 1;
 task=16 : next(vars[4]) := {b,a,e,d,c};
 task=17 : next(vars[10]) := vars[10] + 1;
 task=18 : next(vars[10]) := vars[10] + 2;
 task=19 : next(vars[10]) := vars[10] + 3;
 task=20 : next(vars[10]) := vars[10] + 4;
 task=21 : next(vars[5]) := {b,a,e,d,c};
 task=22 : next(vars[10]) := vars[10] + 4;
 task=23 : next(vars[10]) := vars[10] + 3;
 task=24 : next(vars[10]) := vars[10] + 2;
 task=25 : next(vars[10]) := vars[10] + 1;
 task=26 : next(vars[6]) := {b,a,e,d,c};
 task=27 : next(vars[10]) := vars[10] + 1;
 task=28 : next(vars[10]) := vars[10] + 2;
 task=29 : next(vars[10]) := vars[10] + 3;
 task=30 : next(vars[10]) := vars[10] + 4;
 task=31 : next(vars[7]) := {b,a,e,d,c};
 task=32 : next(vars[10]) := vars[10] + 1;
 task=33 : next(vars[10]) := vars[10] + 2;
 task=34 : next(vars[10]) := vars[10] + 3;
 task=35 : next(vars[10]) := vars[10] + 4;
 task=36 : next(vars[8]) := {b,a,e,d,c};
 task=37 : next(vars[10]) := vars[10] + 1;
 task=38 : next(vars[10]) := vars[10] + 2;
 task=39 : next(vars[10]) := vars[10] + 3;
 task=40 : next(vars[10]) := vars[10] + 4;
 task=41 : next(vars[9]) := {b,a,e,d,c};
 task=42 : next(vars[10]) := vars[10] + 1;
 task=43 : next(vars[10]) := vars[10] + 2;
 task=44 : next(vars[10]) := vars[10] + 3;
 task=45 : next(vars[10]) := vars[10] + 4;
} }else{
case{
 task=-1 : next(task) := 0;
 task=0 : next(route) := 0;
 task=1 :
  if(vars[0]=b)next(route) := 0;
  else next(task) := task+1;
 task=2 :
  if(vars[0]=c)next(route) := 0;
  else next(task) := task+1;
 task=3 : next(route) := 0;
 task=4 :
  if(vars[1]=a)next(route) := 0;
  else next(task) := task+1;
 task=5 :
  if(vars[1]=b)next(route) := 0;
```

```
 else next(task) := task+1;
task=6 : next(route) := 0;
task=7 :
 if(vars[2]=b)next(route) := 0;
 else next(task) := task+1;
task=8 :
 if(vars[2]=c)next(route) := 0;
 else next(task) := task+1;
task=9 :
 if(vars[2]=d)next(route) := 0;
 else next(task) := task+1;
task=10 :
 if(vars[2]=e)next(route) := 0;
 else next(task) := task+1;
task=11 : next(route) := 0;
task=12 :
 if(vars[3]=a)next(route) := 0;
 else next(task) := task+1;
task=13 :
 if(vars[3]=b)next(route) := 0;
 else next(task) := task+1;
task=14 :
 if(vars[3]=c)next(route) := 0;
 else next(task) := task+1;
task=15 :
 if(vars[3]=d)next(route) := 0;
 else next(task) := task+1;
task=16 : next(route) := 0;
task=17 :
 if(vars[4]=b)next(route) := 0;
 else next(task) := task+1;
task=18 :
 if(vars[4]=c)next(route) := 0;
 else next(task) := task+1;
task=19 :
 if(vars[4]=d)next(route) := 0;
 else next(task) := task+1;
task=20 :
 if(vars[4]=e)next(route) := 0;
 else next(task) := task+1;
task=21 : next(route) := 0;
task=22 :
 if(vars[5]=a)next(route) := 0;
 else next(task) := task+1;
task=23 :
 if(vars[5]=b)next(route) := 0;
 else next(task) := task+1;
task=24 :
 if(vars[5]=c)next(route) := 0;
 else next(task) := task+1;
task=25 :
 if(vars[5]=d)next(route) := 0;
 else next(task) := task+1;
task=26 : next(route) := 0;
```

```
  task=27 :
   if(vars[6]=b)next(route) := 0;
   else next(task) := task+1;
  task=28 :
   if(vars[6]=c)next(route) := 0;
   else next(task) := task+1;
  task=29 :
   if(vars[6]=d)next(route) := 0;
   else next(task) := task+1;
  task=30 :
   if(vars[6]=e)next(route) := 0;
   else next(task) := task+1;
  task=31 : next(route) := 0;
  task=32 :
   if(vars[7]=b)next(route) := 0;
   else next(task) := task+1;
  task=33 :
   if(vars[7]=c)next(route) := 0;
   else next(task) := task+1;
  task=34 :
   if(vars[7]=d)next(route) := 0;
   else next(task) := task+1;
  task=35 :
   if(vars[7]=e)next(route) := 0;
   else next(task) := task+1;
  task=36 : next(route) := 0;
  task=37 :
   if(vars[8]=b)next(route) := 0;
   else next(task) := task+1;
  task=38 :
   if(vars[8]=c)next(route) := 0;
   else next(task) := task+1;
  task=39 :
   if(vars[8]=d)next(route) := 0;
   else next(task) := task+1;
  task=40 :
   if(vars[8]=e)next(route) := 0;
   else next(task) := task+1;
  task=41 : next(route) := 0;
  task=42 :
   if(vars[9]=b)next(route) := 0;
   else next(task) := task+1;
  task=43 :
   if(vars[9]=c)next(route) := 0;
   else next(task) := task+1;
  task=44 :
   if(vars[9]=d)next(route) := 0;
   else next(task) := task+1;
  task=45 :
   if(vars[9]=e)next(route) := 0;
   else next(task) := task+1;
  default : next(task) := -2;
 }
}
```

```
}
/* Superfluous tasks */
SPEC EF(task=-2 & 0<=vars[10] & vars[10]<=7 & vars[7]~=e);
SPEC EF(task=-2 & 0<=vars[10] & vars[10]<=7 & vars[7]=e
& (vars[8]=c | vars[8]=d));
SPEC EF(task=-2 & 0<=vars[10] & vars[10]<=7 & vars[7]=e
& (vars[8]~=c | vars[8]~=d));
SPEC EF(task=-2 & 8<=vars[10] & vars[10]<=12 & vars[7]=e & vars[8]=e);
SPEC EF(task=-2 & 8<=vars[10] & vars[10]<=12 & (vars[7]~=e | vars[8]~=e));
SPEC EF(task=-2 & vars[10]>12);
/* total */
SPEC AF(task=-2 &
( (0<=vars[10] & vars[10]<=7 & vars[7]~=e)
| (0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]=c | vars[8]=d))
| (0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]~=c & vars[8]~=d))
| (8<=vars[10] & vars[10]<=12 & vars[7]=e & vars[8]=e)
| (8<=vars[10] & vars[10]<=12 & (vars[7]~=e | vars[8]~=e))
| (vars[10]>12)));
/* Unique */
SPEC AF(task=-2 &
~(((0<=vars[10] & vars[10]<=7 & vars[7]~=e)
& (0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]=c | vars[8]=d)))
| ((0<=vars[10] & vars[10]<=7 & vars[7]~=e)
& (0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]~=c & vars[8]~=d)))
| ((0<=vars[10] & vars[10]<=7 & vars[7]~=e)
& (8<=vars[10] & vars[10]<=12 & vars[7]=e & vars[8]=e))
| ((0<=vars[10] & vars[10]<=7 & vars[7]~=e)
& (8<=vars[10] & vars[10]<=12 & (vars[7]~=e | vars[8]~=e)))
| ((0<=vars[10] & vars[10]<=7 & vars[7]~=e)
& (vars[10]>12))
| ((0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]=c | vars[8]=d))
& (0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]~=c & vars[8]~=d)))
| ((0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]=c | vars[8]=d))
& (8<=vars[10] & vars[10]<=12 & vars[7]=e & vars[8]=e))
| ((0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]=c | vars[8]=d))
& (8<=vars[10] & vars[10]<=12 & (vars[7]~=e & vars[8]~=e)))
| ((0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]=c | vars[8]=d))
& (vars[10]>12))
| ((0<=vars[10] & vars[10]<=7 & vars[7]=e& (vars[8]~=c & vars[8]~=d))
& (8<=vars[10] & vars[10]<=12 & vars[7]=e & vars[8]=e))
| ((0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]~=c & vars[8]~=d))
& (8<=vars[10] & vars[10]<=12 & (vars[7]~=e | vars[8]~=e)))
| ((0<=vars[10] & vars[10]<=7 & vars[7]=e & (vars[8]~=c & vars[8]~=d))
& (vars[10]>12))
| ((8<=vars[10] & vars[10]<=12 & vars[7]=e & vars[8]=e)
& (8<=vars[10] & vars[10]<=12 & (vars[7]~=e | vars[8]~=e)))
| ((8<=vars[10] & vars[10]<=12 & vars[7]=e & vars[8]=e)
& (vars[10]>12))
| ((8<=vars[10] & vars[10]<=12 & (vars[7]~=e | vars[8]~=e))
& (vars[10]>12))));
}
```