Genetic process mining

Genetic Process Mining

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Alves de Medeiros, Ana Karla

Genetic Process Mining / by Ana Karla Alves de Medeiros. - Eindhoven : Technische Universiteit Eindhoven, 2006. - Proefschrift. -

ISBN 90-386-0785-7

ISBN 978-90-386-0785-6

NUR 983

Keywords: Process mining / Genetic mining / Genetic algorithms / Petri nets / Workflow nets

The work in this thesis has been carried out under the auspices of Beta Research School for Operations Management and Logistics.

Beta Dissertation Series D89

Printed by University Press Facilities, Eindhoven
Cover design: Paul Verspaget & Carin Bruinink, Nuenen, The Netherlands.
The picture was taken by Ana Karla Alves de Medeiros while visiting "Chapada Diamantina" in Bahia, Brazil.

Genetic Process Mining

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op dinsdag 7 november 2006 om 14.00 uur

door

Ana Karla Alves de Medeiros

geboren te Campina Grande, Brazilië

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. W.M.P. van der Aalst

Copromotor:
dr. A.J.M.M. Weijters

To my parents, Salomão and Rosilene.

Contents

Chapter 1

Introduction

Nowadays, most organizations use information systems to support the execution of their business processes [37]. Examples of information systems supporting operational processes are Workflow Management Systems (WMS) [12, 21], Customer Relationship Management (CRM) systems, Enterprise Resource Planning (ERP) systems and so on. These information systems may contain an explicit model of the processes (for instance, workflow systems like Staffware [6], COSA [1], etc), may support the tasks involved in the process without necessarily defining an explicit process model (for instance, ERP systems like SAP R/3 [5]), or may simply keep track (for auditing purposes) of the tasks that have been performed without providing any support for the actual execution of those tasks (for instance, custom-made information systems in hospitals). Either way, these information systems typically support logging capabilities that register what has been executed in the organization. These produced logs usually contain data about cases (i.e. process instances) that have been executed in the organization, the times at which the tasks were executed, the persons or systems that performed these tasks, and other kinds of data. These logs are the starting point for process mining, and are usually called *event logs*. For instance, consider the event log in Table 1.1. This log contains information about four process instances (cases) of a process that handles fines.

Process mining targets the *automatic* discovery of information from an event log. This discovered information can be used to deploy new systems that support the execution of business processes or as a feedback tool that helps in auditing, analyzing and improving already enacted business processes. The main benefit of process mining techniques is that information is *objectively* compiled. In other words, process mining techniques are helpful because they gather information about what is *actually* happening according to an event log of a organization, and not what people *think* that is happen-

Case ID	Task Name	Event Type	Originator	Timestamp	Extra Data
1	File Fine	Completed	Anne	20-07-2004 14:00:00	...
2	File Fine	Completed	Anne	20-07-2004 15:00:00	...
1	Send Bill	Completed	system	20-07-2004 15:05:00	...
2	Send Bill	Completed	system	20-07-2004 15:07:00	...
3	File Fine	Completed	Anne	21-07-2004 10:00:00	...
3	Send Bill	Completed	system	21-07-2004 14:00:00	...
4	File Fine	Completed	Anne	22-07-2004 11:00:00	...
4	Send Bill	Completed	system	22-07-2004 11:10:00	...
1	Process Payment	Completed	system	24-07-2004 15:05:00	...
1	Close Case	Completed	system	24-07-2004 15:06:00	...
2	Send Reminder	Completed	Mary	20-08-2004 10:00:00	...
3	Send Reminder	Completed	John	21-08-2004 10:00:00	...
2	Process Payment	Completed	system	22-08-2004 09:05:00	...
2	Close case	Completed	system	22-08-2004 09:06:00	...
4	Send Reminder	Completed	John	22-08-2004 15:10:00	...
4	Send Reminder	Completed	Mary	22-08-2004 17:10:00	...
4	Process Payment	Completed	system	29-08-2004 14:01:00	...
4	Close Case	Completed	system	29-08-2004 17:30:00	...
3	Send Reminder	Completed	John	21-09-2004 10:00:00	...
3	Send Reminder	Completed	John	21-10-2004 10:00:00	...
3	Process Payment	Completed	system	25-10-2004 14:00:00	...
3	Close Case	Completed	system	25-10-2004 14:01:00	...

Table 1.1: Example of an event log.

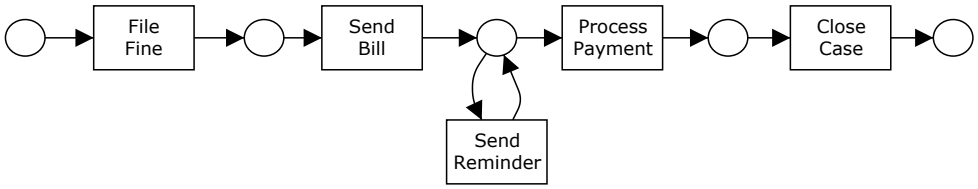


Figure 1.1: Petri net illustrating the control-flow perspective that can be mined from the event log in Table 1.1.

ing in this organization. The starting point of any process mining technique is an event log.

The type of data in an event log determines which *perspectives* of process mining can be discovered. If the log (i) provides the tasks that are executed in the process and (ii) it is possible to infer their order of execution and link these taks to individual cases (or process instances), then the *control-flow perspective* can be mined. The log in Table 1.1 has this data (cf. fields "Case ID", "Task Name" and "Timestamp"). So, for this log, mining algorithms could discover the process in Figure 1.1[1]. Basically, the process describes that after a fine is entered in the system, the bill is sent to the driver. If the driver does not pay the bill within one month, a reminder is sent. When

[1]The reader unfamiliar with Petri nets is referred to Section 3.1.

the bill is paid, the case is archived. If the log provides information about the persons/systems that executed the tasks, the *organizational perspective* can be discovered. The organizational perspective discovers information like the social network in a process, based on transfer of work, or allocation rules linked to organizational entities like roles and units. For instance, the log in Table 1.1 shows that "Anne" transfers work to both "Mary" (case 2) and "John" (cases 3 and 4), and "John" sometimes transfers work to "Mary" (case 4). Besides, by inspecting the log, the mining algorithm could discover that "Mary" never has to send a reminder more than once, while "John" does not seem to perform as good. The managers could talk to "Mary" and check if she has another approach to send reminders that "John" could benefit from. This can help in making good practices a common knowledge in the organization. When the log contains more details about the tasks, like the values of data fields that the execution of a task modifies, the *case perspective* (i.e. the perspective linking data to cases) can be discovered. So, for instance, a forecast for executing cases can be made based on already completed cases, exceptional situations can be discovered etc. In our particular example, logging information about the profiles of drivers (like age, gender, car etc) could help in assessing the probability that they would pay their fines on time. Moreover, logging information about the places where the fines were applied could help in improving the traffic measures in these places. From this explanation, the reader may have already noticed that the control-flow perspective relates to the "How?" question, the organizational perspective to the "Who?" question, and the case perspective to the "What?" question. All these three perspectives are complementary and relevant for process mining. However, *in this thesis we focus on the control-flow perspective of process mining.*

1.1 Control-Flow Mining

The control-flow perspective mines a process model that specifies the relations between tasks in an event log. From event logs, one can find out information about which tasks belong to which process instances, the time at which tasks are executed, the originator of tasks, etc. Therefore, the mined process model is an *objective picture* that depicts possible flows that were followed by the cases in the log (assuming that the events were correctly logged). Because the flow of tasks is to be portrayed, control-flow mining techniques need to support the correct mining of the common control-flow constructs that appear in process models. These constructs are: sequences, parallelism, choices, loops, and non-free-choice, invisible tasks and duplicate

tasks [15]. *Sequences* express situations in which tasks are performed in a predefined order, one after the other. For instance, for the model in Figure 1.2, the tasks "Enter Website" and "Browse Products" are in a sequence. *Parallelism* means that the execution of two or more tasks are independent or concurrent. For instance, the task "Fill in Payment Info" can be executed independently of the tasks "Login" and "Create New Account" in Figure 1.2. *Choices* model situations in which either one task or another is executed. For instance, the tasks "Remove Item from Basket" and "Add Item to Basket" are involved in the same choice in the model in Figure 1.2. The same holds for the tasks "Cancel Purchase" and "Commit Purchase". *Loops* indicate that certain parts of a process can be repeatedly executed. In the model in Figure 1.2, the block formed by the tasks "Browse Products", "Remove Item from Basket", "Add Item to Basket" and "Calculate Total" can be executed multiple times in a row. *Non-free-choice constructs* model a mix of synchronization and choice. For instance, have a look at the non-free-choice construct involving the tasks "Calculate Total" and "Calculate Total with Bonus". Note that the choice between executing one of these two tasks is not done after executing the task "Fill in Delivery Address", but depends on whether the task "Login" or the task "Create New Account" has been executed. In this case, the non-free-choice construct is used to model the constraint that only returning customers are entitled to bonuses. *Invisible tasks* correspond to silent steps that are used for routing purposes only and, therefore, they are not present in event logs. Note that the model in Figure 1.2 has three invisible tasks (the black rectangles). Two of these invisible tasks are used to skip parts of the process and the other one is used to loop back to "Browse Products". *Duplicate tasks* refer to situations in which multiple tasks in the process have the same label. Duplicates are usually embedded in different contexts (surrounding tasks) in a process. The model in Figure 1.2 has two tasks with the same label "Calculate Total". Both duplicates perform the same action of adding up the prices of the products in the shopping basket. However, the first "Calculate Total" (see top part of Figure 1.2) does so while the client is still selecting products and the second one (see bottom part of Figure 1.2) computes the final price of the whole purchase. Control-flow process mining algorithms should be able to tackle these common constructs.

In fact, there has been quite a lot of work on mining the control-flow perspective of process models [14, 17, 35, 43, 23, 52, 73, 64, 81]. For instance, the work in [52] can mine duplicate tasks, [81] can mine non-free-choice, [17] proves to which classes of models their mining algorithm always works, [23] mines common control-flow patterns and [73] captures partial synchronization in block-structured models. However, *none of the current control-flow*

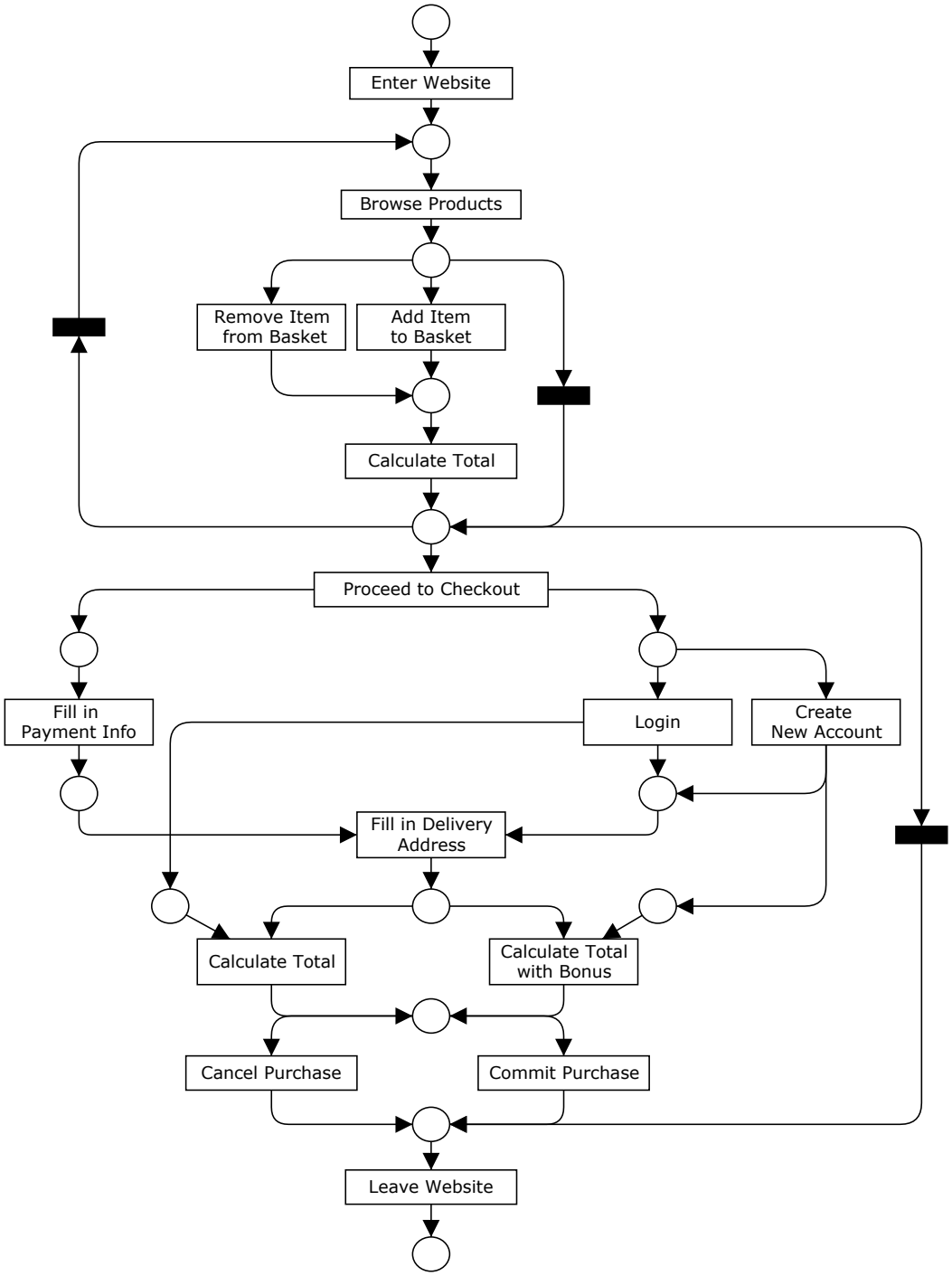


Figure 1.2: Example of a Petri net that contains all the common control-flow constructs that may appear in business processes.

process mining techniques is able to mine all constructs at once. Furthermore, many of them have problems while dealing with another factor that is common in real-life logs: *the presence of noise*. Noise can appear in two situations: event traces were somehow incorrectly logged (for instance, due to temporary system misconfiguration) or event traces reflect exceptional situations. Either way, most of the techniques will try to find a process model that can parse all the traces in the log. However, the presence of noise may hinder the correct mining of the most frequent behavior in the log.

The *first reason* why these techniques have problems to handle all the constructs is that they are based on *local* information in the log. In other words, they use the information about what tasks directly precede or directly follow each other in a log to set the dependencies between these tasks. The problem is that some of these dependencies are not captured by direct succession or precedence. For instance, consider the non-free-choice construct in Figure 1.2. Note that the tasks involved in this non-free-choice constructs never directly follow of precede each other. A *second reason* why some of the techniques cannot mine certain constructs is because the notation they use to model the processes does not support these constructs. For instance, the notation used by the α-algorithm [17] and extensions [81] does not allow for invisible tasks or duplicate tasks. Furthermore, approaches like the ones in [52] and [73] only work over block-structured processes. *Finally*, for many of these approaches, the number of times a relation holds in the log is irrelevant. Thus, these approaches are very vulnerable to noise because they are unable to distinguish between high frequent and low frequent behavior.

Given all these reasons, we decided to investigate the following research question: *Is it possible to develop a control-flow process mining algorithm that can discover all the common control-flow structures and is robust to noisy logs at once?* This thesis attempts to provide an answer to this question. In our case, we have applied genetic algorithms [38, 61] to perform process mining. We call it *genetic process mining*. The choice for using genetic algorithms was mainly motivated by the absence of good heuristics that can tackle all the constructs, and by the fact that genetic algorithms are intrinsically robust to noise.

1.2 Genetic Process Mining

Genetic algorithms are a search technique that mimics the process of evolution in biological systems. The main idea is that there is a search space that contains some solution point(s) to be found by the genetic algorithm. The algorithm starts by randomly distributing a finite number of points into this

search space. Every point in the search space is called an *individual* and the finite set of points at a given moment in time is called a *population*. Every individual has an *internal representation* and the quality of an individual is evaluated by the *fitness* measure. The search continues in an iterative process that creates new individuals in the search space by recombining and/or mutating existing individuals of a given population. That is the reason why genetic algorithms mimic the process of evolution. They always re-use material contained in already existing individuals. Every new iteration of a genetic algorithm is called a *generation*. The parts that constitute the internal representation of individuals constitute the *genetic material* of a population. The recombination and/or modification of the genetic material of individuals is performed by the *genetic operators*. Usually, there are two types of genetic operators: *crossover* and *mutation*. The crossover operator recombines two individuals (or two *parents*) in a population to create two new individuals (or two *offsprings*) for the next population (or generation). The mutation operator randomly modifies parts of individuals in the population. In both cases, there is a *selection criterion* to choose the individuals that may undergo crossover and/or mutation. To guarantee that good genetic material will not be lost, a number of the best individuals in a population (the *elite*) is usually directly copied to the next generation. The search proceeds with the creation of generations (or new populations) until certain *stop criteria* are met. For instance, it is common practice to set the maximum amount of generations (or new populations) that can be created during the search performed by the genetic algorithm, so that the search process ends even when no individual with maximal fitness is found.

In the context of this thesis, individuals are process models, the fitness assesses how well an individual (or process model) reflects the behavior in an event log, and the genetic operators recombine these individuals so that new candidate process models can be created. Therefore, our challenge was to define (i) an internal representation that supports all the common constructs in process models (namely, sequences, parallelism, choices, loops, and non-free-choice, invisible tasks and duplicates tasks), (ii) a fitness measure that correctly assesses the quality of the created process models (or individuals) in every population and (iii) the genetic operators so that the whole search space defined by the internal representation can be explored. All these three points are related and it is common practice in the genetic algorithm community to experiment with different versions of internal representations, fitness measures and genetic operators. However, in this thesis, we have opted for "fixing" a single combination for internal representation and genetic operators, and experimenting more with variations of the fitness measure. The main motivation is to find out the core requirements that the fitness measure

of a genetic algorithm to mine process models should consider, regardless of the internal representation and the genetic operators. In our case, since we look for a process model that objectively captures the behavior expressed in the log, one obvious requirement to assess the quality of any individual (or process model) is that this individual can reproduce (or parse) the traces (i.e. cases or process instances) in the event log. However, although this requirement is necessary, it is not sufficient because usually there is more than one individual that can reproduce the behavior in the log. In particular, there is the risk of finding *over-general* or *over-specific* individuals. An over-general individual can parse any case (or trace) that is formed by the tasks in a log. For instance, Figure 1.3(a) shows an over-general individual (or process model) for the log in Table 1.1. Note that this individual can reproduce all the behavior in Table 1.1, but also allows for extra behavior that cannot be derived from the four cases in this log. As an illustration, note that the model in Figure 1.3(a) allows for the execution of the task "Send Bill" after the task "Close Case". An over-specific solution can only reproduce the exact behavior in the log, without any form of knowledge extraction or abstraction. For instance, Figure 1.3(b) shows an over-specific individual for the log in Table 1.1. Note that this individual does not allow the task "Send Reminder" to be executed more than three times in a row. In fact, over-specific models like this one do not give much more information than a simple look at the unique cases in a log. The fitness measure described in these thesis can distinguish between over-general and over-specific solutions.

The genetic approach described in this thesis has been implemented as plug-ins in the ProM framework [32, 78].

1.3 Methodology

To evaluate our genetic approach, we have used a variety of simulation models [2] and a case study. *Simulation* was used to create the noise-free logs for the experiments. As illustrated in Figure 1.4, based on existing process models (or original process models), logs were generated and given as input to the genetic miner we have developed. The genetic miner discovered a model (the mined model) for every log[3]. Note that the performed discovery is based on data in the logs only. This means that, like it happens in real-life situations, no information about the original models is used during the mining process and only the logs are provided. Once the models were discovered, their qual-

[2]The simulation models were taken both from literature and student projects.

[3]Actually, a population of mined models (individuals) is returned, but only the best individual is selected.

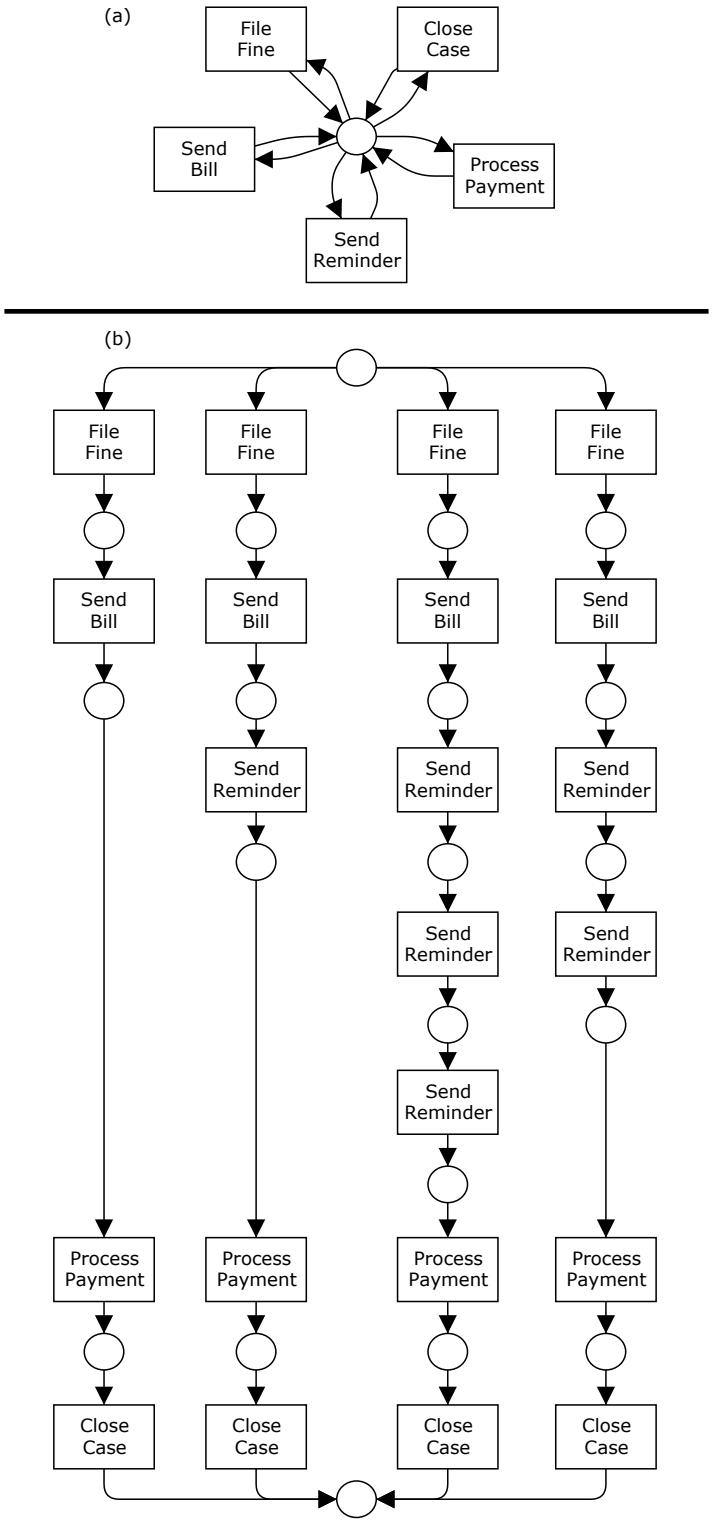


Figure 1.3: Examples of over-general (a) and over-specific (b) process models to the log in Table 1.1.

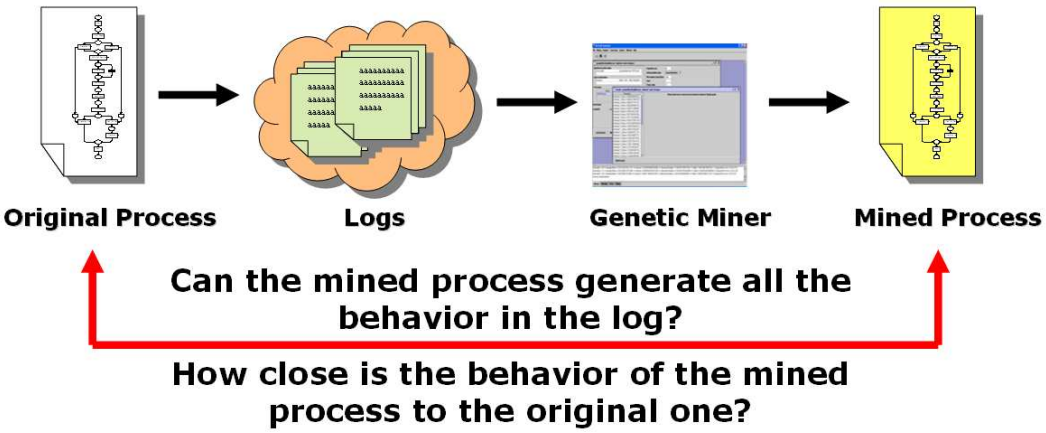


Figure 1.4: Experimental setup for synthetic logs created via simulation.

ity was assessed based on three criteria: (i) how many of the traces in the log can be reproduced (or parsed) by the mined model, (ii) how close is the behavior of the mined model to the behavior of the original one, and (iii) whether the mined model tends to be over-general or over-specific. Because different mined models can portrait the exact same behavior, we had to develop analysis metrics that go beyond the sheer comparison of the structures of the mined and original process models. Still for the experiments with simulated logs, two settings were used: experiments with synthetic logs of *known models* and of *unknown models*. The main difference from the experiments with known models and unknown ones is that, in the first case, the original models were mainly created by us and, therefore, were known before the mined models were selected. In the second case, also called *single-blind experiments*, the original model and the logs were generated by other people. In both cases, models were mined based on the logs only and the best mined model was objectively selected based on its fitness measure. The *case study* was conducted to show the applicability of our genetic mining approach in a real-life setting. The logs for the case study came from a workflow system used in a Dutch municipality.

1.4 Contributions

This thesis has two main contributions:

- The definition of a genetic algorithm that can mine models with *all* common structural constructs that can be found in process models, while correctly capturing the *most frequent* behavior in the log. The

definition of this algorithm consists of (i) an internal representation, (ii) a fitness measure and (iii) two genetic operators: crossover and mutation.

- The definition of analysis metrics that check for the quality of the mined models based on the amount of behavior that they have in common with the original models. These metrics are applicable beyond the scope of genetic algorithms and quantify the similarity of any two process models with respect to a given event log.

Additional contributions are:

- The provision of a common framework to create synthetic logs for process mining algorithms.
- The implementation of all algorithms in an open-source tool (ProM).

1.5 Road Map

In this section we give a brief description of each chapter. The suggested order of reading for the chapters is illustrated in Figure 1.5. The chapters are organized as follows:

Chapter 1 The current chapter. This chapter provides an overview of the approach and motivates the need for better process mining techniques.

Chapter 2 Provides a review of the related work in the area of control-flow process mining.

Chapter 3 Uses a simple but powerful existing mining algorithm to introduce the reader to the process mining field. Additionally, this chapter defines notions that are used in the remaining chapters.

Chapter 4 Explains a genetic algorithm that can handle all common structural constructs, except for duplicate tasks. This chapter is the core of this thesis because the solutions provided in chapters 5 and 6 build upon the material presented in this chapter.

Chapter 5 Extends the approach in Chapter 4 to also mine process models with duplicate tasks.

Chapter 6 Shows how we support the post-pruning of arcs from mined models. The arc post-pruning can be used to manually "filter out" arcs in the mined models that represent infrequent/exceptional behavior.

Chapter 7 Presents all the plug-ins that were developed in the ProM framework to realize our genetic mining approach. The plug-ins range from the mining algorithms themselves to the implementation of the defined analysis metrics.

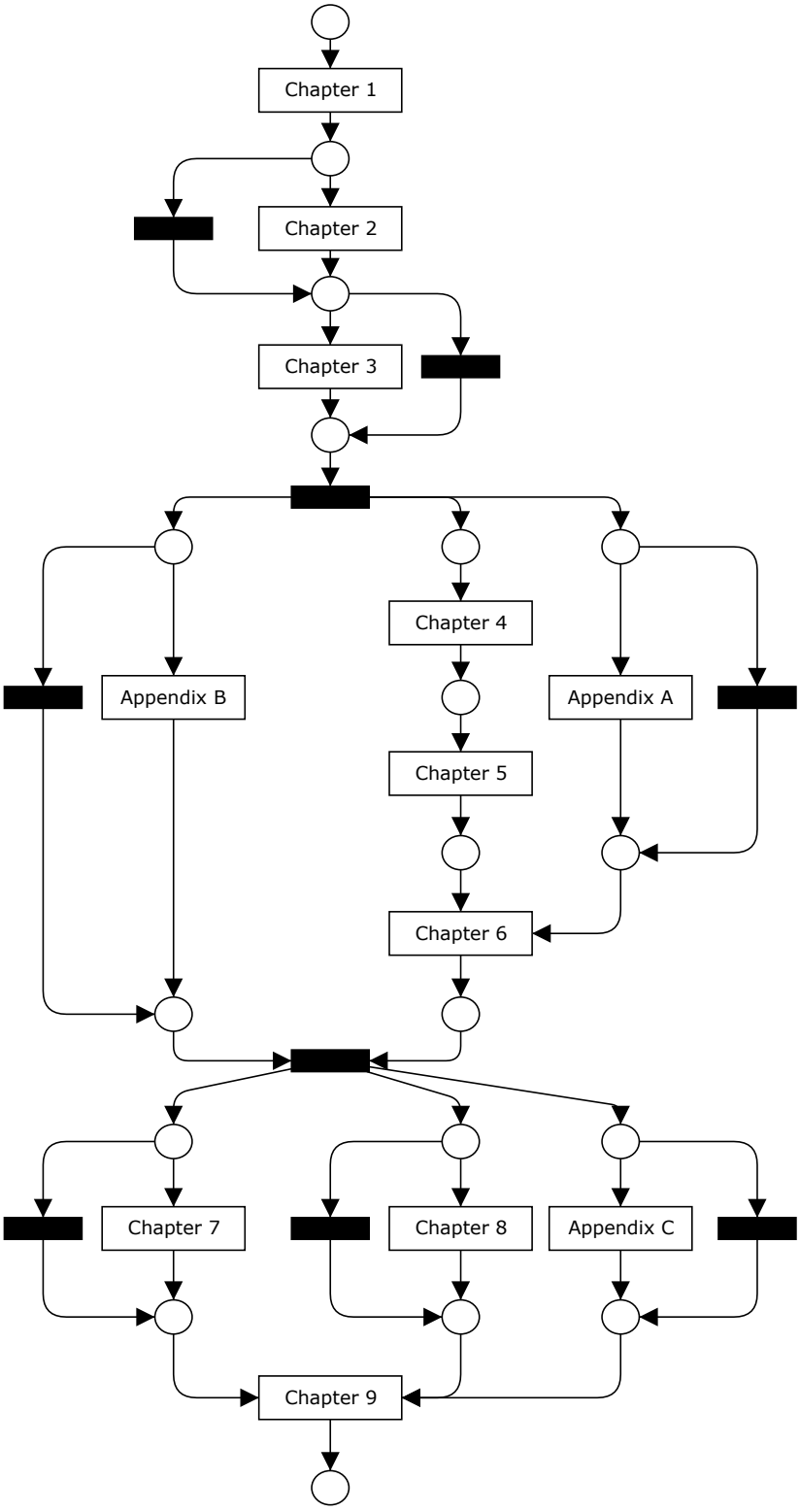


Figure 1.5: Suggested order of reading for the chapters and appendices in this thesis.

Chapter 8 Describes the experiments with known models, the single-blind experiments and the case study.

Chapter 9 Concludes this thesis and points out directions for future work.

In addition, three appendices are provided. **Appendix A** contains the mapping forth-and-back between Petri nets and the internal representation of individuals. **Appendix B** shows all the models that were used during the experiments with known models. **Appendix C** contains all the models that were used during the single-blind experiments.

Chapter 2

Related Work

This thesis presents a genetic approach for discovering the *control-flow perspective* of a process model. Therefore, this chapter focusses on reviewing other approaches that also target the mining of this perspective. The remainder of this chapter is organized as follows. Section 2.1 provides an overview about the related approaches. Section 2.2 describes in more details how every approach (mentioned in Section 2.1) works. Section 2.3 presents a short summary of this chapter.

2.1 Overview of the Related Approaches

The first papers on process mining [1] appeared in 1995, when Cook et al. [22, 23, 24, 25, 26] started to mine process models from event logs in the context of software engineering. They called it *process discovery*. Process mining in the business sense was first introduced 1998 by Agrawal et al. [18]. They called it *workflow mining*. Since then, many groups have focussed on mining process models [14, 17, 35, 43, 52, 73, 64, 81].

This section gives an overview about each one of these groups. The lenses we use to analyse each approach is how well they handle the common structural patterns that can appear in the processes. We do so because our focus is on the mining of the control-flow perspective. The structural patterns are: sequence, parallelism, choice, loops, non-free-choice, invisible tasks and duplicate tasks. These patterns were already explained in Section 1.1. Additionally to this control-flow pattern perspective, we also look at how robust the techniques are with respect to noise. This is important because real-life logs usually contain noise.

[1] In this section, whenever we use the term *process mining*, we actually mean the mining of the *control-flow perspective* of a process model.

Table 2.1 summarizes the current techniques. Before we dig into checking how every related work approach is doing, let us explain what every row in the table means:

- **Event log** indicates if the technique assumes that the log has information about the start and the completion of a task (*non-atomic tasks*) or not (*atomic tasks*). For instance, consider the net in Figure 2.1(a). Techniques that work based on atomic tasks require at least two traces to detect the parallelism between tasks A and B. One trace with the substring "AB" and another with "BA". However, techniques that work with non-atomic tasks can detect this same parallelism with a single process instance in which the execution times of A and B overlap. In other words, the substring "$A_{start}, B_{start}, A_{complete}, B_{complete}$" would be enough to detect the parallel construct.
- **Mined model** refers to the amount of information that is directly shown in the structure of the mined process model. Some techniques mine a process model that only expresses the task *dependencies* (for instance, see Figure 2.1(b)); in other techniques, the mined process model also contains the semantics of the *split/join* points. In other words, other techniques directly show if the split/join points have an *OR*, *XOR* or *AND* semantics (for instance, see Figure 2.1(a)). Moreover, some mining techniques aim at mining a *whole model*, others focus only on identifying the *most common* substructures in the process model.
- **Mining approach** indicates if the technique tries to mine the process model in a *single step* or if the approach has *multiple steps* with intermediary mined process models that are refined in following steps.
- **Sequence**, **choice** and **parallelism** respectively show if the technique can mine tasks that are in a sequential, choice or concurrent control-flow pattern structure.
- **Loops** points out if the technique can mine only *block-structured* loops, or if the technique can handle *arbitrary* types of loops. For instance, Figure 2.1(e) shows a loop that is not block-structured.
- **Non-free-choice** shows if the technique can mine non-free-choice constructs that can be detected by looking at *local* information at the log or *non-local* one. A non-local non-free-choice construct cannot be detected by *only* looking at the direct successors and predecessors (the local context) of a task in a log. As an illustration, consider the model in figures 2.1(c) and 2.1(d). Both figures show a non-free-choice construct involving the tasks A, B, C, D and E. However, the non-free-choice in Figure 2.1(d) is local, while the one in Figure 2.1(c) is not. Note that

the task A never directly precedes the task C and a similar situation holds for the tasks B and D in any trace of a log for the model in Figure 2.1(c).

- **Invisible tasks** refer to the type of invisible tasks that the technique can tackle. For example, invisible tasks can be used to *skip* other tasks in a choice situation (see Figure 2.1(f), where B is skipped). Other invisible tasks are used for more elaborate routing constructs like *split/join* points in the model (see Figure 2.1(g), where the AND-split and AND-join are invisible "routing" tasks.).
- **Duplicate tasks** can be in *sequence* in a process model (see Figure 2.1(h)), or they can be in *parallel* branches of a process model (see Figure 2.1(i)).
- Added to the structural constructs, the **noise** perspective shows how the techniques handle noise in the event log. Most of the techniques handle noisy logs by first inferring the process model and, then, making a *post-pruning* of the dependencies (or arcs) that are below a given threshold.

The columns in Table 2.1 show the first author of the publication to which the analysis summarized in this table refers to [2]. Now that we know what the information in the table means, let us see what we can conclude from it.

In short, Table 2.1 shows the following. Agrawal et al. [18] and Pinter et al. [64] are the only ones that do not explicitly capture the nature of the split/join points in the mined models. The reason is that they target a model for the Flowmark workflow system [53] and every point in this system has an OR-split/join semantics. In fact, every directed arc in the model has a boolean function that evaluates to true or false after a task is executed. The evaluation of the boolean conditions sets how many branches are activated after a task is executed. Cook et al. [23] have the only approach that does not target a whole mined model. Their approach looks for the most frequent patterns in the model. Actually, sometimes they do mine a whole process model, but that is not their main aim. The constructs that cannot be mined by all techniques are loops, non-free-choice, invisible tasks and duplicate tasks. Grecco et al. [44] cannot mine any kind of loops. The reason is that they prove that the models their algorithms mine allow for as little extra behavior (that is not in the event log) as possible. They do so by enumerating all the traces that the mined model can generate and comparing them with the traces in the event log. Models with loops would make this task impractical. Some other techniques cannot mine arbitrary

[2]Some of the authors have worked in multiple approaches.

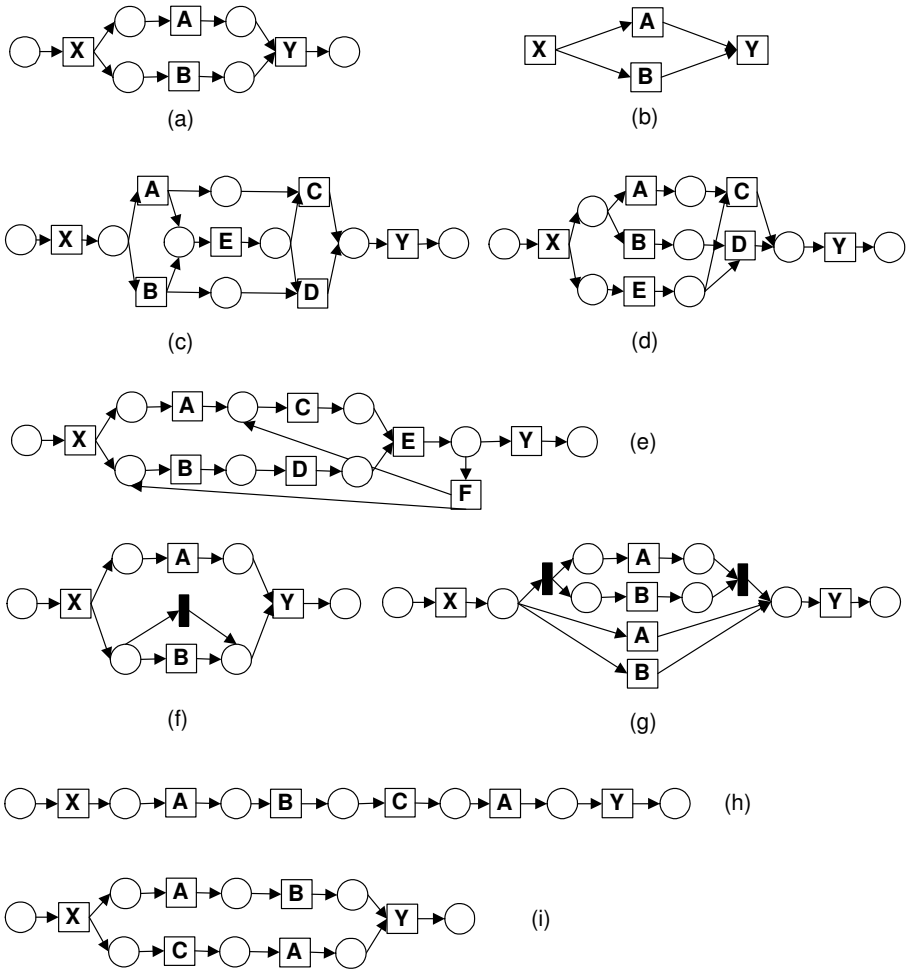


Figure 2.1: Illustration of the main control-flow patterns.

	Cook et al. [23]	Agrawal et al. [18]	Pinter et al. [64]	Herbst et al. [52]	Schimm [73]	Grecco et al. [44]	Van der Aalst et al. [17]	Weijters et al. [79]	Dongen et al. [35]	Wen et al. [81]
Event log:										
- Atomic tasks	✓	✓		✓		✓	✓	✓	✓	✓
- Non-atomic tasks			✓		✓					
Mined model:										
- Dependencies	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
- Nature split/join	✓			✓	✓	✓	✓	✓	✓	✓
- Whole model		✓	✓	✓	✓	✓	✓	✓	✓	✓
Mining approach:										
- Single step	✓	✓	✓				✓	✓		
- Multiple steps				✓	✓	✓			✓	✓
Sequence:	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Choice:	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Parallelism:	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Loops:										
- Structured	✓	✓	✓	✓	✓		✓	✓	✓	✓
- Arbitrary		✓	✓				✓	✓	✓	✓
Non-free-choice:										
- Local	✓	✓	✓			✓	✓	✓	✓	✓
- Non-local										✓
Invisible Tasks:										
- Skip	✓	✓	✓	✓	✓	✓		✓	✓	
- Split/join	✓			✓						
Duplicate Tasks:										
- Sequence				✓						
- Parallel				✓						
Noise:										
- Dependency pruning	✓	✓	✓	✓						
- Other								✓		

Table 2.1: Related work analysis.

loops because their model notation (or representation) does not support this kind of loop. The main reason why most techniques cannot mine *non-local* non-free-choice is that most of their mining algorithms are based on local information in the logs. The techniques that do not mine *local* non-free-choice cannot do so because their representation does not support such a construct. Usually the technique is based on a block-structured notation, like Herbst et al. and Schimm. Skip tasks are not mined due to representation limitations as well. Split/join invisible tasks are not mined by many techniques, except for Schimm and Herbst et al.. Actually, we also do not target at discovering such kind of tasks. However, it is often the case that it is possible to build a model without any split/join invisible tasks that expresses the same behavior as in the model with the split/join invisible tasks. Duplicate tasks are not mined because many techniques assume that the mapping between the tasks and their labels is injective. In other words, the labels are unique per task. The only techniques that mine duplicate tasks are Cook et al. [24, 25] for sequential processes only, and Herbst et al. [49, 51, 52] for both sequential and parallel processes. We do not consider Schimm [73] to mine process models with duplicate tasks because his approach assumes that the detection of the duplicate tasks is done in a pre-processing step. This step identifies all the duplicates and makes sure that they have unique identifiers when the event log is given as input to the mining algorithm. Actually, all the techniques that we review here would tackle duplicate tasks if this same pre-processing step would be done before the log in given as input to them.

2.2 A More Detailed Analysis of Related Approaches

For the interested reader, the subsections 2.2.1 to 2.2.10 give more details about each of the approaches in Table 2.1 (see Section 2.1). Every subsection contains a short summary about the approach and highlights its main characteristics.

2.2.1 Cook et al.

Cook et al. [22, 23, 24, 25, 26] were the first ones to work on process mining. The main difference of their approach compared to the other approaches is that they do not really aim at retrieving a complete and correct model, but a model that express the most frequent patterns in the log.

In their first papers [22, 24, 25], they extended and developed algorithms to mine sequential patterns from event logs in the software development

domain. These patterns were expressed as Finite State Machines (FSMs). In [22, 24, 25] they show three algorithms: *RNet*, *KTail* and *Markov*. From these three algorithms, only Markov was fully created by Cook et al.. The other algorithms were existing methods extended by the authors. The *RNet algorithm* provides a purely statistical approach that looks at a window of *predecessor events* while setting the next event in the FSM. The approach uses neural networks. The authors extended this approach to work with window sizes bigger than 2 and to more easily retrieve the process model "coded" in the neural network. The *Ktail algorithm* provides a purely algorithmic approach that looks at a window of *successor events* while building the equivalence classes that compose the mined process model. The authors extended this approach to perform more folding in the mined model and to make it robust to noise. The *Markov algorithm* is based on a mix of a statistical and algorithmic approaches and looks at both *predecessors and successors* events while inserting a task into a process model. The approach uses *frequency tables* to set the probability that an event will occur, given that it was preceded and succeeded by a sequence of certain size. The Markov algorithm proved to be superior to the other two algorithms. RNet was the "worst" of the three algorithms. All algorithms are implemented in the DaGama tool that is part of the data analysis framework Balboa.

In [23, 26], Cook et al. extend their Markov approach to mine concurrent process models. The main challenge here was to identify the nature of the split/join points. One notable fact is that the authors now *work with a window size of one*. In other words, they only look at the frequency tables for the direct predecessors and direct successors of an event. Additional to this change, the authors define four statistical metrics that are used to distinguish between XOR/AND-split/join points. The metrics are: entropy, event type counts, causality and periodicity. The *entropy* indicates how related two event types are. This is important to set the direct successors and predecessors of an event type. The *event type counts* distinguish between the AND/XOR-split/join situations. Note that in an AND-split situation, the split point and its direct successors are executed the same amount of times. In an XOR-split situation, the amount of times the direct successors are executed add up to the amount of times the XOR-split point executes (assuming a noise-free log!). The *causality* metric is used to distinguish between concurrent events and length-two-loop events. The *periodicity* metric helps in identifying the synchronization points. Due to their probabilistic nature, Cook et al.'s algorithms are robust to noise.

2.2.2 Agrawal et al.

Agrawal et al. [18] were the first ones to apply process discovery (or process mining, as we name it) in a business setting, specifically in the context of IBM's MQ Series workflow product. Their mining algorithm assumes that the log has atomic tasks. The mined model shows the dependencies between these tasks, but no indication of the semantics of the split/join points. Besides, their approach requires the target model to have a single start task and a single end task. This is not really a limitation since one can always preprocess the log and respectively insert a start task and an end one at the beginning and at the end of every process instance. The algorithm does not handle duplicate tasks and assumes that no task appears more than once in a process instance. So, to tackle loops, a re-labeling process takes place.

The mining algorithm aims at retrieving a complete model. In a nutshell, it works as follows. First the algorithm renames repeated labels in a process instance. This ensures the correct detection of loops. Second, the algorithm builds a dependency-relation graph with all the tasks in the event log. Here it is remarkable how the dependency relations are set. The algorithm does not only consider the appearance of task A next to B in a same process instance. Two tasks may also be dependent based on transitiveness. So, if the task A is followed by the task C and C by B, then B may depend on A even when they never appear in a same process instance. After the dependency-relation graph has been inferred, the algorithm removes the arrows in both directions between two tasks, removes the strongly connected components, and applies a transitive reduction to the subgraph (at the main graph) that represents a process instance. Finally, the re-labelled tasks are merged into a single element.

The mined model has the dependencies and the boolean functions associated to the dependencies. These boolean functions are determined based on other parameters in the log. Although they do not show in [18] how to mine these functions, they indicate that a data mining classifier algorithm can be used to do so. The noise is handled by pruning the arcs that are inferred less times than a certain threshold. The authors have implemented their approach (since they conducted some experiments), but they do not mention a public place where one could get this implementation.

2.2.3 Pinter et al.

Pinter et al.'s work [42, 64] is an extension of the Agrawal et al.'s one (see Subsection 2.2.2). The main difference is that they consider the tasks to have a *start* and a *complete* event. So, the detection of parallelism becomes

more trivial. However, better mined models are also obtained. Another difference is that they only look at the process instances individually to set the dependency relations. Agrawal et al. looked at the log as whole. The rest is pretty much the same. Like Agrawal et al., the authors also do not mention where to get their approach's implementation.

2.2.4 Herbst et al.

The remarkable aspect of Herbst et al.'s [49, 51, 52] approach is its ability to tackle duplicate tasks. Herbst et al. developed three algorithms: *MergeSeq*, *SplitSeq* and *SplitPar*. All the three algorithms can mine models with duplicate tasks. In short, the algorithms mine process models in a two-step approach. In the first step, a beam-search is performed to induce a *Stochastic Activity Graph* (SAG) that captures the dependencies between the tasks in the workflow log. This graph has transition probabilities associated to every dependency but no information about AND/XOR-split/join. The transition probability indicates the probability that a task is followed by another one. The beam-search is mainly guided by the *log-likelihood* (LLH) metric. The LLH metric indicates how well a model express the behavior in the event log. The second step converts the SAG to the Adonis Definition Language (ADL). The ADL is a block-structured language to specify workflow models. The conversion from SAG to ADL aims at creating well-defined workflow models. MergeSeq and SplitSeq [51] are suitable for the mining of sequential process models. SplitPar [49, 52] can also mine concurrent process models. The *MergeSeq algorithm* is a bottom-up approach that starts with a SAG that has a branch for every unique process instance in the log, and apply successive folding to nodes in this SAG. The *SplitSeq algorithm* has a top-down approach that starts with a SAG that models the behavior in the log but does not contain any duplicate tasks. The algorithm applies a set of split operations to nodes *in the SAG*. The *SplitPar algorithm* can be considered as an extension of the SplitSeq one. It is also top-down. However, because it targets at mining concurrent processes, the split operations are done *at the process instance level*, instead of on the SAG directly. The reason for that is that the split of a node may have non-local side effects for the structure of the process model. All algorithms are implemented in the InWoLve mining tool. In [47], Hammori shows how he made the InWoLve tool more user-friendly. His guidelines are useful for people that want to develop a process mining tool.

2.2.5 Schimm

The main difference from Schimm's approach [14, 69, 70, 71, 72, 73] to the others is that he aims at retrieving a complete and *minimal* model. In other words, the model does not generalize beyond what is in the log. For instance, the other approaches consider two events to be parallel if there is an interleaving of their appearance in the log. In Schimm's case, there are two possibilities for these events. If their start and complete times overlap, they are indeed mined in parallel. Otherwise, they are in an interleaving situation. So, they can occur in any order, by they cannot occur at the same time.

Schimm mines process models by defining a set of axioms for applying rewriting rules over a workflow algebra. The models are block-structured (well-formed, safe and sound). Because the models are block-structured, one would expect that Schimm's approach cannot model partial synchronization. However, Schimm smartly extended his algebra to model pointers to the tasks in the models, so that the modelling of partial synchronization becomes viable. The main idea is that a task can be executed whenever all of its pointers in the model are enabled. Note that this is different from duplicate tasks.

Schimm's approach does not really tackle duplicates during the mining. Actually, it assumes that his mining technique is embedded in a Knowledge Discovery Database (KDD) process [39]. This way, the log is pre-processed and all duplicate tasks are detected by their context (predecessors and successors). The author does not elaborate in his papers how to perform such detection. Additionally, this pre-processing phase also makes sure that noise is tackled before the event-log is given to the mining algorithm.

The mining algorithm assumes that tasks have a start and a complete event. So, the overlapping between start and complete states of different tasks is used to detect concurrent behavior, alternative behavior or causal one. The algorithm uses a six-step approach: (1) the algorithm relabels the multiple appearances of a same *task identifier* in a log; (2) the algorithm cluster the process instances based on the *happened-before* relationship and the set of tasks; (3) the clusters are further grouped based on the *precedence* relation; (4) a block-structured model is built for every cluster and they are bundled into a single process model that has a big alternative operator at the beginning; (5) the term-rewriting axioms are used to perform folding in the models; and (6), if the event log has this information, resource allocation is included into the mined model. The last step is optional. The algorithm is implemented in the tool *Process Miner* [69].

Schimm's approach cannot mine non-local non-free-choice constructs be-

cause it only looks at direct neighbors in a process instance. We do not consider the approach to mine duplicate tasks because the duplicates are detected in a pre-processing step.

2.2.6 Greco et al.

Greco et al. [43, 44] aim at mining a hierarchical tree of process models that describe the event log at different levels of abstraction. The root of the tree has the most general model that encompasses the features (patterns of sequences) that are common in all process instances of the log. The leaves represent process models for partitions of the log. The nodes in between the root node and the leaf ones show the common features of the nodes that are one level below. To build this tree, the authors have a two-step approach. The first step is top-down. It starts by mining a model for the whole event log. Given this root model, a feature selection is performed to cluster the process instances in the log in partition sets. This process is done for every level of the tree until stop conditions are met. Once the tree is built, the second step takes place. This second step is bottom-up. It starts at the leaves of the mined tree and goes up until the root of the tree. The main idea is that every parent model (or node in the tree generated in the first step) is an abstract view of all of its child models (or nodes). Therefore, the parent model preserves all the tasks that are common to all of its child models, but replaces the tasks that are particular to some of its child models by new tasks. These new tasks correspond to sub-processes in the parent model. The first step of the approach is implemented as the *Disjunctive Workflow Schema (DWS) mining plug-in* in the ProM framework tool [32].

2.2.7 Van der Aalst et al.

Van der Aalst et al. [11, 17, 28, 29, 32, 33] developed the α-algorithm. The main difference from this approach to the others is that Van der Aalst et al. prove to which class of models their approach is guaranteed to work. The authors assume the log to be noise-free and complete with respect to the *follows* relation. So, if in the original model a task A can be executed just before a task B (i.e., B follows A), at least one process instance in the log shows this behavior. Their approach is proven to work for the class of *Structured Workflow Nets* (SWF-nets) *without short loops and implicit places* (see [17] for details). The α-algorithm works based on binary relations in the log. There are four relations: *follows*, *causal*, *parallel* and *unrelated*. Two tasks A and B have a *follows* relation if they appear next to each other in the log. This relation is the basic relation from which the other relations

derive. Two tasks A and B have a *causal* relation if A follows B, but B does not follow A. If B also follows A, then the tasks have a *parallel* relation. When A and B are not involved in a follows relation, they are said to be *unrelated*. Note that all the dependency relations are inferred based on local information in the log. Therefore, the α-algorithm cannot tackle non-local non-free-choice. Additionally, because the α-algorithm works based on sets, it cannot mine models with duplicate tasks.

In [11, 28, 29] the α-algorithm is extended to mine short loops as well. The work presented in [28, 29] did so by adding more constraints to the notion of log completeness and by redefining the binary relations. The work in [11] works based on non-atomic tasks, so that parallel and short-loop constructs can be more easily captured.

The α-algorithm was originally implemented in the EMiT tool [33]. Afterwards, this algorithm became the *α-algorithm plug-in* in the ProM framework tool [32].

Finally, it is important to highlight that the α-algorithm does not take into account the frequency of a relation. It just checks if the relation holds or not. That is also one of the reasons why the α-algorithm is not robust to noise.

2.2.8 Weijters et al.

The approach by Weijters et al. [16, 79] can be seen as an extension of the α-algorithm. As described in Section 2.2.7, it works based on the follows relation. However, to infer the remaining relations (causal, parallel and unrelated), *it considers the frequency of the follows relation in the log*. For this reason this approach can handle noise. The approach is also a bit similar to Cook et al.'s one (see Subsection 2.2.1). The main reason behind the heuristics is that the more often task A follows task B and the less often B follows A, the higher the probability that A is a cause for B. Because the algorithm mainly works based on binary relations, the non-local non-free-choice constructs cannot be captured. The algorithm was originally implemented in the *Little Thumb* tool [79]. Nowadays this algorithm is implemented as the *Heuristics miner plug-in* in the ProM framework tool [32].

2.2.9 Van Dongen et al.

Van Dongen et al. [34, 35] introduced a multi-step approach to mine Event-driven Process Chains (EPCs). The first step consists of mining a process model for every trace in the event log. To do so, the approach first makes sure that no task appears more than once in every trace. This means that

every *instance* of a task in a trace is assigned a unique identifier. After this step, the algorithm infers the binary relations like the α-algorithm does (see Subsection 2.2.7). These binary relations are inferred *at the log level*. Based on these relations, the approach builds a model *for every trace* in the log. These models show the *partial order* between the instances of tasks in a trace. Note that at the trace level no choice is possible because all tasks (instances) in the trace have been indeed executed. The second step performs the aggregation (or merging) of the models mined, during the first step, for every trace. Basically, the identifiers that refer to instances of a same task are merged. The algorithm distinguishes between three types of split/join points: AND, OR and XOR. The type of the split/join points is set based on counters associated to the edges of the aggregated task. It is a bit like the frequency metric in Cook et al. (see Subsection 2.2.1). If a split point occurs as often as its direct successors, an AND-split is set. If the occurrence of its successors add up to the number of times this split point was executed, an XOR-split is determined. Otherwise, an OR-split is set. The algorithm is implemented as the *Multi-phase mining plug-in* at the ProM framework tool [32].

2.2.10 Wen et al.

Wen et al. [80, 81] have implemented two extensions for the α-algorithm (cf. Subsection 2.2.7). *The first extension* - the β-algorithm [80] - can mine Structured Workflow Nets (SWF-nets) *with short loops*. The extension is based on the assumption that the tasks in the log are *non-atomic*. The approach uses the intersecting execution times of tasks to distinguish between parallelism and short loops. The β-algorithm has been implemented as the *Tsinghua-alpha algorithm plugin* at the ProM framework. The *second extension* - the α^{++}-algorithm [81] - can mine Petri nets with *local or non-local non-free-choice* constructs. This extension follows-up on the extensions in [28, 29]. The main idea is that the approach looks at window sizes bigger than 1 to set the dependencies between the tasks in the log. This second extension is implemented as the *Alpha++ algorithm plug-in* in the ProM framework.

2.3 Summary

This chapter reviewed the main approaches that aim at mining the control-flow perspective of a process model. The approaches were compared based on their capabilities to handle the common constructs in process models, as well as the presence of noise in the log. As it can be seen in Table 2.1,

the constructs that cannot be mined by all approaches are: loops (especially the arbitrary ones), invisible tasks, non-free-choice (especially the non-local ones) and duplicate tasks. Loops and invisible tasks cannot be mined mainly because they are not supported by the representation that is used by the approaches. Non-free-choice is not mined because most of the approaches work based on local information in the event log. As discussed in Section 2.1, non-local non-free-choice requires the techniques to look at more distant relationships between the tasks. Duplicate tasks cannot be mined because many approaches assume an one-to-one relation between the tasks in the log and their labels. Finally, noise cannot be properly tackled because many techniques do not take the frequency of the task dependencies into account when mining the model.

As mentioned in Chapter 1, our aim is to develop an algorithm to mine models that can also contain advanced constructs in processes and is robust to noise. By looking at the reasons why the current approaches have problems to mine such constructs, we can already draw some requirements to our algorithm: (i) the representation should support all constructs, (ii) the algorithm should also consider non-local information in the log, and (iii) the algorithm should take into account the frequency of the traces/tasks in the log. However, before digging deep into how our algorithm actually works, let us get more insight into the mining of the control-flow perspective of process models. The next chapter uses the α-algorithm to do so. We chose this algorithm because some the concepts used in our genetic approach were inspired by the concepts dealt with in the α-algorithm.

Chapter 3

Process Mining in Action: The α-algorithm

This chapter uses the α-algorithm [17] to give the reader more insight into the way *control-flow perspective* of a process can be mined. We chose the α-algorithm for two main reasons: (i) it is simple to understand and provides a basic introduction into the field of process mining, and (ii) some of its concepts are also used in our genetic approach. The α-algorithm receives as input an *event log* that contains the sequences of execution (traces) of a process model. Based on this log, *ordering relations* between tasks are inferred. These ordering relations indicate, for instance, whether a task is a *cause* to another tasks, whether two tasks are in *parallel*, and so on. The α-algorithm uses these ordering relations to (re-)discover a process model that describes the behavior in the log. The mined (or discovered) process model is expressed as a *Workflow net* (WF-net). Workflow nets form a special type of *Petri nets*. This chapter introduces and defines the concepts and notions that are required to understand the α-algorithm. However, some of these definitions and notions are also used in chapters 4 to 6. For example, the notions of *Petri nets* (Definition 1), *bags*, *firing rule* (Definition 2), *proper completion* (Definition 10), *implicit place* (Definition 11) and *event log* (Definition 13) are also used in subsequent chapters. The reader familiar with process mining techniques and the previously mentioned notions can safely skip this chapter.

The remainder of this chapter is organized as follows. Petri nets, workflow nets and some background notations used by the α-algorithm are introduced in Section 3.1. The α-algorithm itself is described in Section 3.2. The limitations of the α-algorithm are discussed in Section 3.3. Section 3.4 explains the relations between the constructs that the α-algorithm cannot correctly mine. Pointers to extensions of the α-algorithm are given in Section 3.5.

Section 3.6 summarizes this chapter.

3.1 Preliminaries

This section contains the main definitions used by the α-algorithm. A more detailed explanation about the α-algorithm and Structured Workflow Nets (SWF-nets) is given in [17]. In Subsection 3.1.1, standard Petri-net notations are introduced. Subsection 3.1.2 defines the class of WF-nets.

3.1.1 Petri Nets

We use a variant of the classic Petri-net model, namely *Place/Transition nets*. For an elaborate introduction to Petri nets, the reader is referred to [31, 62, 66].

Definition 1 (P/T-nets). [1] *A Place/Transition net, or simply P/T-net, is a tuple* (P, T, F) *where:*

1. P *is a finite set of* places*,*
2. T *is a finite set of* transitions *such that* $P \cap T = \emptyset$*, and*
3. $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs, called the* flow relation*.*

A marked *P/T-net is a pair* (N, s)*, where* $N = (P, T, F)$ *is a P/T-net and where* s *is a bag over* P *denoting the* marking *of the net, i.e.* $s \in P \rightarrow \mathbb{N}$*. The set of all marked P/T-nets is denoted* $\mathcal{N}$*.*

A marking is a *bag* over the set of places P, i.e., it is a function from P to the natural numbers. We use square brackets for the enumeration of a bag, e.g., $[a^2, b, c^3]$ denotes the bag with two a-s, one b, and three c-s. The sum of two bags $(X + Y)$, the difference $(X - Y)$, the presence of an element in a bag $(a \in X)$, the intersection of two bags $(X \cap Y)$ and the notion of subbags $(X \leq Y)$ are defined in a straightforward way and they can handle a mixture of sets and bags.

Let $N = (P, T, F)$ be a P/T-net. Elements of $P \cup T$ are called *nodes*. A node x is an *input node* of another node y iff there is a directed arc from x to y (i.e., $(x, y) \in F$ or xFy for short). Node x is an *output node* of y iff yFx. For any $x \in P \cup T$, $\overset{N}{\bullet}x = \{y \mid yFx\}$ and $x\overset{N}{\bullet} = \{y \mid xFy\}$; the superscript N may be omitted if clear from the context.

[1] In the literature, the class of Petri nets introduced in Definition 1 is sometimes referred to as the class of (unlabeled) *ordinary* P/T-nets to distinguish it from the class of Petri nets that allows more than one arc between a place and a transition, and the class of Petri nets that allows for transition labels.

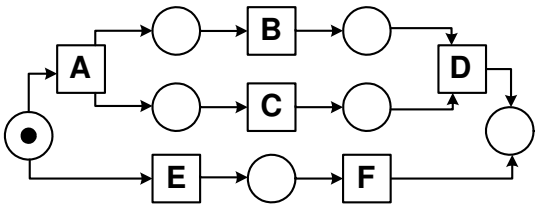


Figure 3.1: An example of a Place/Transition net.

Figure 3.1 shows a P/T-net consisting of 7 places and 6 transitions. Transition A has one input place and two output places. Transition A is an *AND-split*. Transition D has two input places and one output place. Transition D is an *AND-join*. The black dot in the input place of A and E represents a token. This token denotes the initial marking. The dynamic behavior of such a marked P/T-net is defined by a *firing rule*.

Definition 2 (Firing rule). *Let $N = ((P, T, F), s)$ be a marked P/T-net. Transition $t \in T$ is* enabled*, denoted $(N, s)[t\rangle$, iff $\bullet t \leq s$. The* firing rule *$\_[\_\rangle\_ \subseteq \mathcal{N} \times T \times \mathcal{N}$ is the smallest relation satisfying for any $(N = (P, T, F), s) \in \mathcal{N}$ and any $t \in T$, $(N, s)[t\rangle \Rightarrow (N, s)\ [t\rangle\ (N, s - \bullet t + t\bullet)$.*

In the marking shown in Figure 3.1 (i.e., one token in the source place), transitions A and E are enabled. Although both are enabled only one can fire. If transition A fires, a token is removed from its input place and tokens are put in its output places. In the resulting marking, two transitions are enabled: B and C. Note that B and C can be fired concurrently and we assume *interleaving semantics*. In other words, parallel tasks are assumed to be executed in some order.

Definition 3 (Reachable markings). *Let (N, s_0) be a marked P/T-net in $\mathcal{N}$. A marking s is* reachable *from the initial marking s_0 iff there exists a sequence of enabled transitions whose firing leads from s_0 to s. The set of reachable markings of (N, s_0) is denoted $[N, s_0\rangle$.*

The marked P/T-net shown in Figure 3.1 has 6 reachable markings. Sometimes it is convenient to know the sequence of transitions that are fired in order to reach some given marking. This thesis uses the following notations for sequences. Let A be some alphabet of identifiers. A *sequence of length n*, for some natural number $n \in \mathbb{N}$, over alphabet A is a function $\sigma : \{0, \ldots, n-1\} \rightarrow A$. The sequence of length zero is called the empty sequence and written ε. For the sake of readability, a sequence of positive length is usually written by juxtaposing the function values. For example, a sequence $\sigma = \{(0, a), (1, a), (2, b)\}$, for $a, b \in A$, is written aab. The set of all sequences of arbitrary length over alphabet A is written A^*.

Definition 4 ($\in$, *first*, *last*). *Let A be a set, $a_i \in A$ ($i \in \mathbb{N}$), and $\sigma = a_0 a_1 ... a_{n-1} \in A^*$ a sequence over A of length n. Functions $\in$, first, last are defined as follows:*

1. *$a \in \sigma$ iff $a \in \{a_0, a_1, ...a_{n-1}\}$,*
2. *$first(\sigma) = a_0$, if $n \geq 1$, and*
3. *$last(\sigma) = a_{n-1}$, if $n \geq 1$.*

Definition 5 (Firing sequence). *Let (N, s_0) with $N = (P, T, F)$ be a marked P/T net. A sequence $\sigma \in T^*$ is called a* firing sequence *of (N, s_0) if and only if, for some natural number $n \in \mathbb{N}$, there exist markings $s_1, \ldots, s_n$ and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = t_1 \ldots t_n$ and, for all i with $0 \leq i < n$, $(N, s_i)[t_{i+1}\rangle$ and $s_{i+1} = s_i - \bullet t_{i+1} + t_{i+1}\bullet$. (Note that $n = 0$ implies that $\sigma = \varepsilon$ and that ε is a firing sequence of (N, s_0).) Sequence σ is said to be* enabled *in marking s_0, denoted $(N, s_0)[\sigma\rangle$. Firing the sequence σ results in a marking s_n, denoted $(N, s_0)\ [\sigma\rangle\ (N, s_n)$.*

Definition 6 (Connectedness). *A net $N = (P, T, F)$ is* weakly connected*, or simply* connected*, iff, for every two nodes x and y in $P \cup T$, $x(F \cup F^{-1})^* y$, where R^{-1} is the inverse and R^* the reflexive and transitive closure of a relation R. Net N is* strongly connected *iff, for every two nodes x and y, xF^*y.*

We assume that all nets are weakly connected and have at least two nodes. The P/T-net shown in Figure 3.1 is connected but not strongly connected.

Definition 7 (Boundedness, safeness). *A marked net (N, s), with $N = (P, T, F)$, is* bounded *iff the set of reachable markings $[N, s\rangle$ is finite. It is* safe *iff, for any $s' \in [N, s\rangle$ and any $p \in P$, $s'(p) \leq 1$. Note that safeness implies boundedness.*

The marked P/T-net shown in Figure 3.1 is safe (and therefore also bounded) because none of the 6 reachable states puts more than one token in a place.

Definition 8 (Dead transitions, liveness). *Let (N, s), with $N = (P, T, F)$, be a marked P/T-net. A transition $t \in T$ is* dead *in (N, s) iff there is no reachable marking $s' \in [N, s\rangle$ such that $(N, s')[t\rangle$. (N, s) is* live *iff, for every reachable marking $s' \in [N, s\rangle$ and $t \in T$, there is a reachable marking $s'' \in [N, s'\rangle$ such that $(N, s'')[t\rangle$. Note that liveness implies the absence of dead transitions.*

None of the transitions in the marked P/T-net shown in Figure 3.1 is dead. However, the marked P/T-net is not live since it is not possible to enable each transition repeatedly.

3.1.2 Workflow Nets

Most workflow systems offer standard building blocks such as the AND-split, AND-join, XOR-split, and XOR-join [12, 40, 54, 57]. These are used to model sequential, conditional, parallel and iterative routing (WFMC [40]). Clearly, a Petri net can be used to specify the routing of cases. *Tasks* are modeled by transitions and causal dependencies are modeled by places and arcs. In fact, a place corresponds to a *condition* which can be used as pre- and/or post-condition for tasks. An AND-split corresponds to a transition with two or more output places, and an AND-join corresponds to a transition with two or more input places. XOR-splits/joins correspond to places with multiple outgoing/ingoing arcs. Given the close relation between tasks and transitions we use the terms interchangeably.

A Petri net which models the control-flow dimension of a workflow, is called a *WorkFlow net* (WF-net). It should be noted that a WF-net specifies the dynamic behavior of a single case in isolation.

Definition 9 (Workflow nets). *Let $N = (P, T, F)$ be a P/T-net and $\bar{t}$ a fresh identifier not in $P \cup T$. N is a* workflow net *(WF-net) iff:*

1. *object creation: P contains an input place i such that $\bullet i = \emptyset$,*
2. *object completion: P contains an output place o such that $o\bullet = \emptyset$,*
3. *connectedness: $\bar{N} = (P, T \cup \{\bar{t}\}, F \cup \{(o, \bar{t}), (\bar{t}, i)\})$ is strongly connected,*

The P/T-net shown in Figure 3.1 is a WF-net. Note that although the net is not strongly connected, the *short-circuited* net with transition $\bar{t}$ is strongly connected. Even if a net meets all the syntactical requirements stated in Definition 9, the corresponding process may exhibit errors such as deadlocks, tasks which can never become active, livelocks, garbage being left in the process after termination, etc. Therefore, we define the following correctness criterion.

Definition 10 (Sound). *Let $N = (P, T, F)$ be a WF-net with input place i and output place o. N is* sound *iff:*

1. *safeness: $(N, [i])$ is safe,*
2. *proper completion: for any marking $s \in [N, [i]\rangle$, $o \in s$ implies $s = [o]$,*
3. *option to complete: for any marking $s \in [N, [i]\rangle$, $[o] \in [N, s\rangle$, and*
4. *absence of dead tasks: $(N, [i])$ contains no dead transitions.*

The set of all sound WF-nets is denoted $\mathcal{W}$.

The WF-net shown in Figure 3.1 is sound. Soundness can be verified using standard Petri-net-based analysis techniques. In fact soundness corresponds to liveness and safeness of the corresponding short-circuited net [7, 8, 12].

This way efficient algorithms and tools can be applied. An example of a tool tailored towards the analysis of WF-nets is Woflan [76, 77].

The α-algorithm aims at rediscovering WF-nets from event logs. However, not all places in sound WF-nets can be detected. For example places may be implicit which means that they do not affect the behavior of the process. These places remain undetected by the α-algorithm. Therefore, we have defined the notion of implicit places.

Definition 11 (Implicit place). *Let $N = (P, T, F)$ be a P/T-net with initial marking s. A place $p \in P$ is called implicit in (N, s) if and only if, for all reachable markings $s' \in [N, s\rangle$ and transitions $t \in p\bullet$, $s' \geq \bullet t \setminus \{p\} \Rightarrow s' \geq \bullet t$.*

Figure 3.1 contains no implicit places. However, adding a place p connecting transition A and D yields an implicit place. The α-algorithm is unable to detect p because the addition of the place does not change the behavior of the net and therefore is not visible in the log.

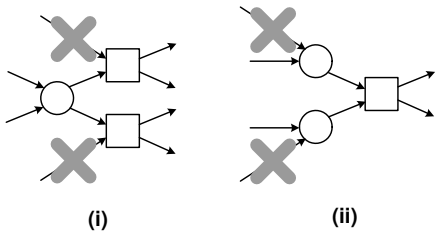


Figure 3.2: Constructs not allowed in SWF-nets.

The α-algorithm is proven to correctly mine *structured workflow nets* (SWF-nets). This implies that the constructs shown in Figure 3.2 may not always be correctly derived by the α-algorithm. The left construct illustrates the constraint that choice and synchronization should never meet. If two transitions share an input place, and therefore "fight" for the same token, they should not require synchronization. This means that choices (places with multiple output transitions) should not be controlled by synchronizations. The right-hand construct in Figure 3.2 illustrates the constraint that if there is a synchronization all directly preceding transitions should have fired, i.e., it is not allowed to have synchronizations directly preceded by an XOR-join. SWF-nets are defined as:

Definition 12 (SWF-net). *A WF-net $N = (P, T, F)$ is an* SWF-net *(Structured workflow net) if and only if:*

1. *For all $p \in P$ and $t \in T$ with $(p, t) \in F$: $|p\bullet| > 1$ implies $|\bullet t| = 1$.*
2. *For all $p \in P$ and $t \in T$ with $(p, t) \in F$: $|\bullet t| > 1$ implies $|\bullet p| = 1$.*

3. There are no implicit places.

3.2 The α-Algorithm

The starting point of any process mining algorithm is an event log. An event log is a bag of event traces. Each trace corresponds to an execution of a process, i.e. a *case* or *process instance*. Note that the same traces of events may occur multiple times in a log. In this situation, different cases followed the same "path" in the process.

Definition 13 (Event Trace, Event Log). *Let T be a set of tasks. $\sigma \in T^*$ is an* event trace *and $L : T^* \to \mathbb{N}$ is an* event log*. For any $\sigma \in dom(L)$, $L(\sigma)$ is the number of occurrences of σ. The set of all event logs is denotes by $\mathcal{L}$.*

Note that we use $dom(f)$ and $rng(f)$ to respectively denote the *domain* and *range* of a function f. Furthermore, we use the notation $\sigma \in L$ to denote $\sigma \in dom(L) \wedge L(\sigma) \geq 1$. For example, assume a log $L = [abcd, acbd, abcd]$ for the net in Figure 3.1. Then, we have that $L(abcd) = 2$, $L(acbd) = 1$ and $L(ab) = 0$.

From an event log, relations between tasks can be inferred. In the case of the α-algorithm, any two tasks in the event log must have one of the following four *ordering relations*: $>_L$ (follows), $\rightarrow_L$ (causal), $\|_L$ (parallel) and $\#_L$ (unrelated). These ordering relations are extracted based on *local information* in the event log. The ordering relations are defined as:

Definition 14 (Log-based ordering relations). *Let L be a event log over T, i.e., $L : T^* \to \mathbb{N}$. Let $a, b \in T$:*

- *$a >_L b$ if and only if there is a trace $\sigma = t_1 t_2 t_3 \ldots t_n$ and $i \in \{1, \ldots, n-1\}$ such that $\sigma \in L$ and $t_i = a$ and $t_{i+1} = b$,*
- *$a \rightarrow_L b$ if and only if $a >_L b$ and $b \not>_L a$,*
- *$a \#_L b$ if and only if $a \not>_L b$ and $b \not>_L a$, and*
- *$a \|_L b$ if and only if $a >_L b$ and $b >_L a$.*

To ensure the event log contains the minimal amount of information necessary to mine the workflow, the notion of log completeness is defined as follows.

Definition 15 (Complete event log). *Let $N = (P, T, F)$ be a sound WF-net, i.e., $N \in \mathcal{W}$. L is an* event log *of N if and only if $dom(L) \in T^*$ and every trace $\sigma \in L$ is a firing sequence of N starting in state $[i]$ and ending in state $[o]$, i.e., $(N, [i])[\sigma\rangle(N, [o])$. L is a* complete *event log of N if and only if for any event log L' of N: $>_{L'} \subseteq >_L$.*

For Figure 3.1, a possible *complete* event log L is: $\{abcd, acbd, ef\}$. From this complete log, the following ordering relations are inferred:

- (follows) $a >_L b$, $a >_L c$, $b >_L c$, $b >_L d$, $c >_L b$, $c >_L d$ and $e >_L f$.
- (causal) $a \rightarrow_L b$, $a \rightarrow_L c$, $b \rightarrow_L d$, $c \rightarrow_L d$ and $e \rightarrow_L f$.
- (parallel) $b \|_L c$ and $c \|_L b$.
- (unrelated) $x \#_L y$ and $y \#_L x$ for $x \in \{a, b, c, d\}$ and $y \in \{e, f\}$.

Now we can give the formal definition of the α-algorithm followed by a more intuitive explanation.

Definition 16 (Mining algorithm α). *Let L be a event log over T. $\alpha(L)$ is defined as follows.*

1. $T_L = \{t \in T \mid \exists_{\sigma \in L} t \in \sigma\}$,
2. $T_I = \{t \in T \mid \exists_{\sigma \in L} t = \mathit{first}(\sigma)\}$,
3. $T_O = \{t \in T \mid \exists_{\sigma \in L} t = \mathit{last}(\sigma)\}$,
4. $X_L = \{(A, B) \mid A \subseteq T_L \wedge B \subseteq T_L \wedge \forall_{a \in A} \forall_{b \in B}\ a \rightarrow_L b \wedge \forall_{a_1, a_2 \in A}\ a_1 \#_L a_2 \wedge \forall_{b_1, b_2 \in B}\ b_1 \#_L b_2\}$,
5. $Y_L = \{(A, B) \in X_L \mid \forall_{(A', B') \in X_L} A \subseteq A' \wedge B \subseteq B' \Longrightarrow (A, B) = (A', B')\}$,
6. $P_L = \{p_{(A,B)} \mid (A, B) \in Y_L\} \cup \{i_L, o_L\}$,
7. $F_L = \{(a, p_{(A,B)}) \mid (A, B) \in Y_L \wedge a \in A\} \cup \{(p_{(A,B)}, b) \mid (A, B) \in Y_L \wedge b \in B\} \cup \{(i_L, t) \mid t \in T_I\} \cup \{(t, o_L) \mid t \in T_O\}$, *and*
8. $\alpha(L) = (P_L, T_L, F_L)$.

The α-algorithm works as follows. First, it examines the log traces and (Step 1) creates the set of transitions (T_L) in the workflow net, (Step 2) the set of output transitions (T_I) of the source place, and (Step 3) the set of the input transitions (T_O) of the sink place. In steps 4 and 5, the α-algorithm creates sets (X_L and Y_L, respectively) used to define the places of the discovered workflow net. In Step 4, the α-algorithm discovers which transitions are causally related. Thus, for each tuple (A, B) in X_L, each transition in set A causally relates to *all* transitions in set B, and no transitions within A (or B) follow each other in some firing sequence. These constraints to the elements in sets A and B allow for the correct mining of AND-split/join and XOR-split/join constructs. Note that the XOR-split/join requires the fusion of places. In Step 5, the α-algorithm refines set X_L by taking only the largest elements with respect to set inclusion. In fact, Step 5 establishes the exact amount of places the discovered net has (excluding the source place i_L and the sink place o_L). The places are created in Step 6 and connected to their respective input/output transitions in Step 7. The discovered workflow net is returned in Step 8.

Finally, we define what it means for a WF-net to be rediscovered.

Definition 17 (Ability to rediscover). *Let $N = (P, T, F)$ be a sound WF-net, i.e., $N \in \mathcal{W}$, and let α be a mining algorithm which maps event logs of N onto sound WF-nets, i.e., $\alpha : \mathcal{L} \rightarrow \mathcal{W}$. If for any complete event log L of N the mining algorithm returns N (modulo renaming of places), then α is able to rediscover N.*

Note that no mining algorithm is able to find names of places. Therefore, we ignore place names, i.e., α is able to rediscover N if and only if $\alpha(L) = N$ modulo renaming of places.

Theorem 1 (Class of nets α-algorithm can always discover). *Let $N = (P, T, F)$ be a sound SWF-net and let L be a complete workflow log of N. If for all $a, b \in T$ $a \bullet \cap \bullet b = \emptyset$ or $b \bullet \cap \bullet a = \emptyset$, then $\alpha(L) = N$ modulo renaming of places.*

As stated in Theorem 1, the α-algorithm is proven to always rediscover the class of *SWF-nets without short loops*. For a proof of Theorem 1, the reader is referred to [17].

3.3 Limitations of the α-algorithm: Loops, Invisible Tasks, Non-Free-Choice and Duplicate Tasks

As motivated in the previous section the α-algorithm can successfully mine SWF-nets that do not contain short-length loops (cf. Theorem 1). But, the α-algorithm has also limitations. In this section we present a classification of possible common constructs the α-algorithm cannot mine correctly, and relations between these constructs. Some of the constructs are within the scope of SWF-nets (like *short loops*), but others are beyond the scope of SWF-nets (like *duplicate tasks*). Moreover, there are WF-nets that can be mined correctly, but that are not SWF-nets (cf. Figure 3.3).

There are problems in the resulting net the α-algorithm produces when its *input* is incomplete and/or has noise (because different relations may be inferred). But even if the log is noise free and complete, there are a number of workflow constructs that cannot be correctly rediscovered by the α-algorithm. Below we will discuss some of the problems. The discussion is illustrated with the WF-nets in Figure 3.4 to 3.8. For each WF-net, the resulting net generated by the α-algorithm is shown in these figures. Note that the resulting nets are based on complete logs of the original models.

Length-one loop In a length-one loop, the same task can be executed multiple times in sequence. Thus, all ingoing places of this task are also

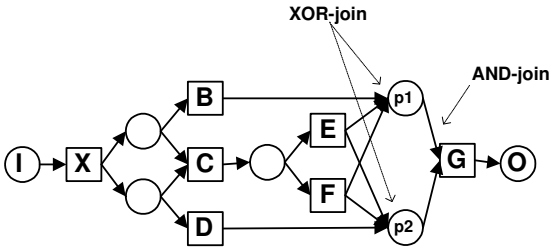


Figure 3.3: A WF-net that can be rediscovered by the α-algorithm, although it is not an SWF-net (see Definition 12).

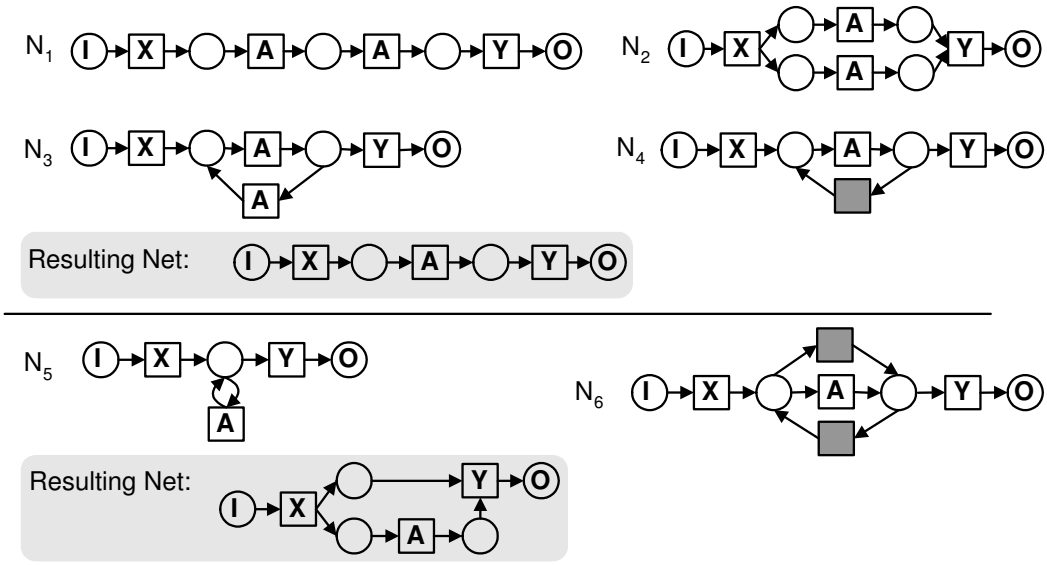


Figure 3.4: Example of the existing relations between duplicate tasks, invisible tasks and length-one loops.

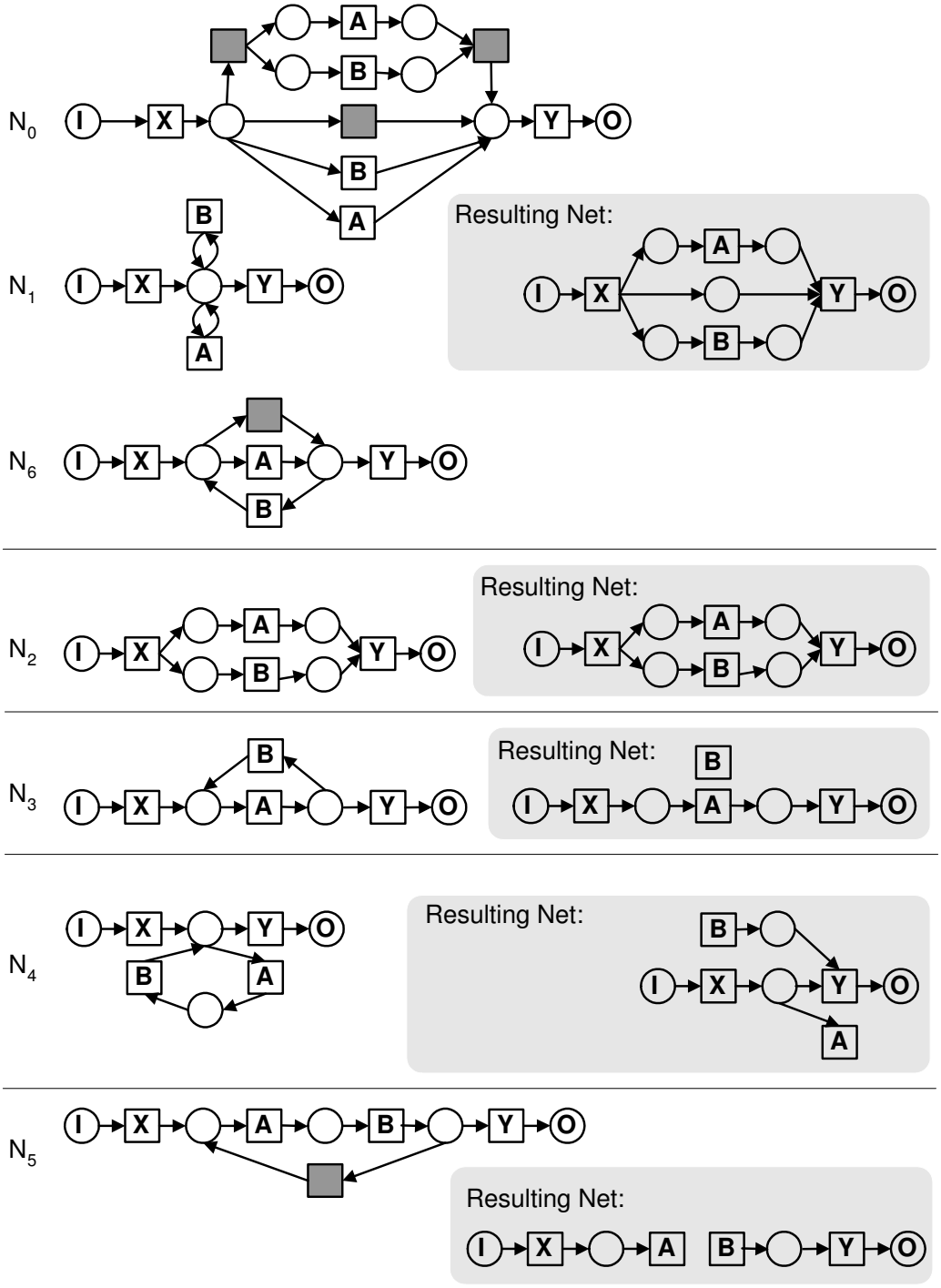


Figure 3.5: Example of the existing relations between duplicate tasks, invisible tasks and length-one/two loops.

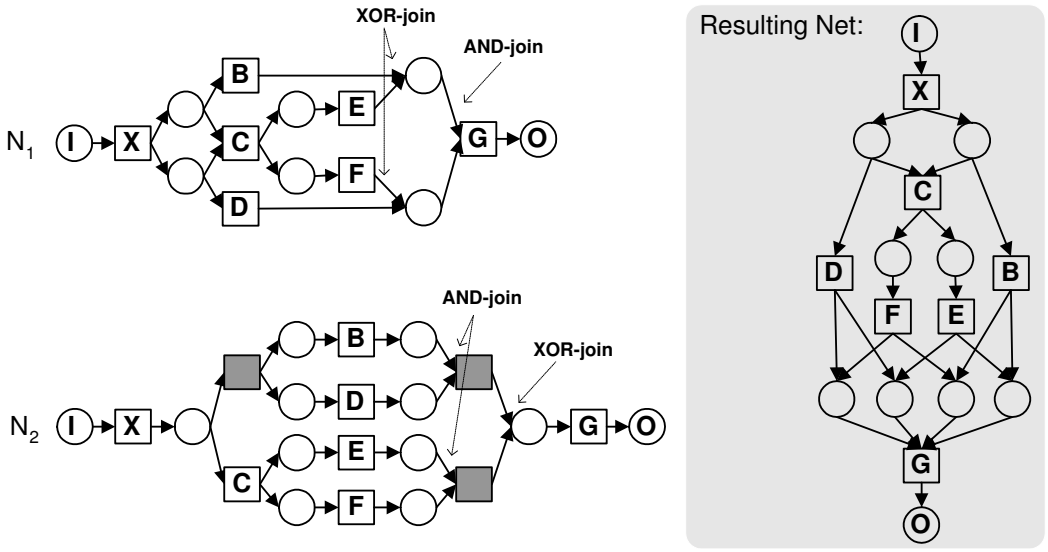


Figure 3.6: Mined and original nets have different number of places.

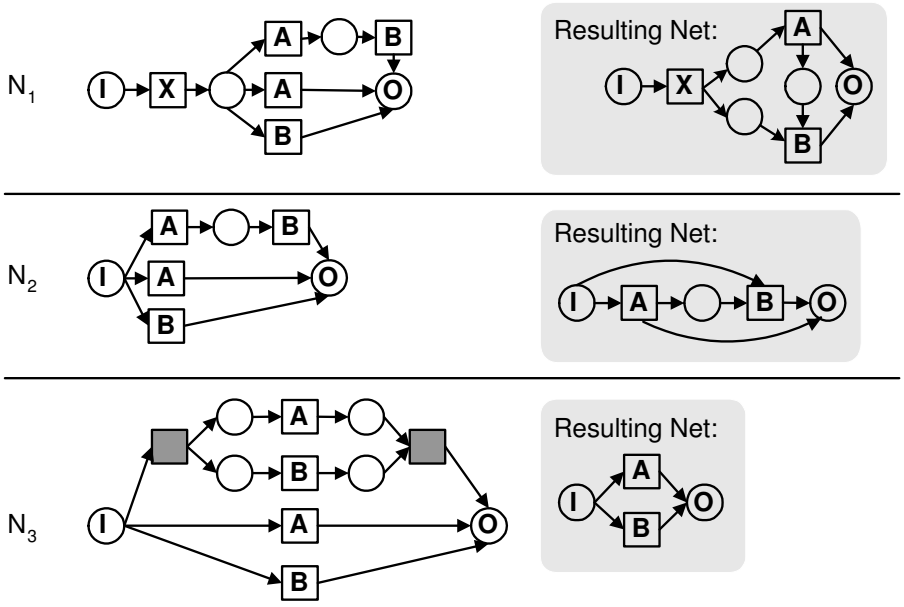


Figure 3.7: Nets with duplicate tasks.

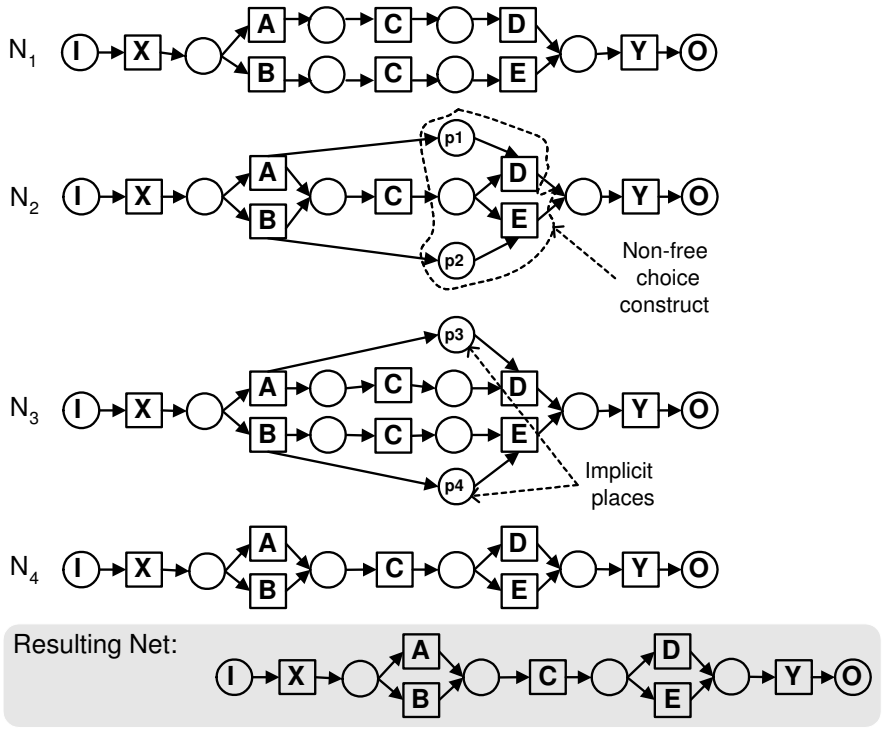


Figure 3.8: Example of existing relations between duplicate tasks, non-free choice nets, implicit places and SWF-nets.

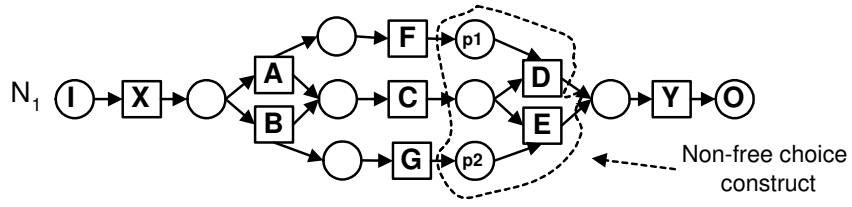


Figure 3.9: Example of a non-free choice net which the α-algorithm can mine correctly.

its outgoing places in the WF-net. In fact, for SWF-nets, a length-one-loop task can only have one single place connected to it. As an example, see net N_5 in Figure 3.4, and also net N_1 in Figure 3.5. Note that in the resulting nets, the length-one-loop transitions do not have the same place as its ingoing and outgoing place. This happens because, to generate a place with a common ingoing and outgoing task, the α-algorithm requires the causal relation $task \rightarrow_L task$. But it is impossible to have $task >_L task$ and $task \not>_L task$ at the same time.

Length-two loop In this case, the α-algorithm infers that the two involved tasks are in parallel and, therefore, no place is created between them. For instance, see nets N_3 and N_4 in Figure 3.5. Note that there are no arcs between tasks A and B in the resulting net. However, the α-algorithm would correctly mine both N_3 and N_4 if the relations $A \rightarrow_L B$ and $B \rightarrow_L A$ were inferred, instead of the relation $A||_L B$.

Invisible Tasks Invisible tasks do not appear in any log trace. Consequently, they do not belong to T_L (set of transitions in the mined net) and cannot be present in the net the α-algorithm generates. Two situations lead to invisible tasks: (i) a task is not registered in the log, for instance, because is only there for routing purposes (e.g., see tasks without label in net N_2, Figure 3.6), or (ii) there is noise in the log generation and some of the task occurrences are missing in the log traces.

Duplicate Tasks Sometimes a task appears more than once in the same workflow. In this case, the same label is given (and thus registered in the log) to more than one task. This can happen, for instance, when modelling the booking process in a travel agency. Clients can go there to *book a flight* only, *book a hotel* only, or *both*. Thus, a workflow model describing this booking situation could be like net N_3, in Figure 3.7 (assume $A =$"book flight" and $B =$"book hotel"). Note that the resulting net for net N_3 contains only one task with label A and one with B. The α-algorithm will never capture task duplication because it cannot distinguish different tasks with the same label (see also the other nets in Figure 3.7). In fact, in an SWF-net it is assumed that tasks are uniquely identifiable. Thus, a heuristic to capture duplicate tasks will have to generate WF-nets in which tasks can have identical labels.

Implicit Places SWF-nets do not have implicit places. Places are implicit if their presence or absence does not affect the possible log traces of a workflow. For example, places $p3$ and $p4$ are implicit in net N_3 (see Figure 3.8). Note that the same causal relations are inferred when these implicit places are present or absent. However, the α-algorithm creates

places according to the existing causal relations. Thus, implicit places cannot be captured because they do not influence causal relations between tasks. Note also that this same reason prevents the α-algorithm from generating *explicit* places between tasks that do not have a causal relation. As an example, see places $p1$ and $p2$ in net N_2 (also in Figure 3.8). Both places constrain the execution of tasks D and E because the choice between the execution of these tasks is made after the execution of A or B, respectively, and not after the execution of C. In fact, if the places $p1$ and $p2$ are removed from N_2, net N_4 is obtained (see Figure 3.8). However, in N_4, the choice between the execution of tasks D and E is made after the execution of task C. Consequently, a log trace like $XACEY$ can be generated by N_4, but is not possible according to N_2.

Non-free choice The non-free choice construct combines synchronization and choice. Thus, it is not allowed in SWF-nets because it corresponds to construct (i) in Figure 3.2. Nets containing non-free choice constructs are not always correctly mined by the α-algorithm. For instance, consider the non-free-choice net N_2, Figure 3.8. The α-algorithm does not correctly mine N_2 because this net cannot generate any log trace with the substring AD and/or BE. Consequently, there is no causal relation $A \rightarrow_L D$ and $B \rightarrow_L E$, and no creation of the respective places $p1$ and $p2$ in the resulting net. However, there are non-free choice constructs which the α-algorithm can mine correctly. As an example, consider net N_1 in Figure 3.9. This net is similar to net N_2 in Figure 3.8, but N_1 has two additional tasks F and G. The α-algorithm can correctly mine N_1 because there is a causal relation $F \rightarrow_L D$ (enabling the creation of place $p1$) and $G \rightarrow_L E$ (enabling the creation of $p2$). Thus, the α-algorithm can correctly mine non-free-choice constructs as far as the causal relations can be inferred.

Synchronization of XOR-join places As shown in Figure 3.2(ii), the *synchronization of XOR-join places* is a non-SWF-net construct. However, although this is a non-SWF-net construct, sometimes the α-algorithm can correctly mine it. For instance, see the WF-net in Figure 3.3. Places $p1$ and $p2$ are XOR-join places. $p1$ is an XOR-join place because it contains a token if task B *or* E *or* F is executed. Similarly, $p2$ if task D *or* E *or* F is executed. Besides, both $p1$ and $p2$ are synchronized at task G, since this task can happen only when there is a token in both $p1$ *and* $p2$. Note that this construct corresponds to a non-SWF-net because task G can be executed whenever *some* of the tasks that precede it have been executed. If the net in Figure 3.3

were an SWF-net, task G could be executed only after the execution of tasks B, D, E *and* F. However, although the net in Figure 3.3 is a non-SWF-net, the α-algorithm can correctly mine it because the necessary and sufficient *causal* ($\rightarrow_L$) and *unrelated*($\#_L$) relations are inferred. However, in other cases, the inferred causal and unrelated relations may not be enough to correctly mine the net. For instance, consider net N_1 in Figure 3.6. The resulting net the α-algorithm mines is not equal to N_1 because it contains two additional places between tasks B, D, E, F and task G. This net structure with extra places derives from the inferred relations. Note that because $B \parallel_L D$ and $E \parallel_L F$ in net N_1, but $B\#_L E$, $B\#_L F$,$D\#_L E$ and $D\#_L F$, the places $p_{(\{B,E\},\{G\})}$, $p_{(\{B,F\},\{G\})}$, $p_{(\{D,E\},\{G\})}$ and $p_{(\{D,F\},\{G\})}$ are created by the α-algorithm, while only places $p_{(\{B,E\},\{G\})}$ and $p_{(\{D,F\},\{G\})}$ would do. Thus, in this case, the inferred relations do not allow the α-algorithm to correctly mine the net. However, the resulting net is *behaviorally* equivalent to the original net, even if their structures are different because both nets generate exactly the same set of traces. Note that the two additional places in the resulting net are implicit. However, in this case the problem is not the absence of implicit places; the problem is the addition of implicit places by the α-algorithm.

3.4 Relations among Constructs

There are relations among the problematic constructs identified in Section 3.3. The problematic constructs are related because (i) the same set of log traces can satisfy the current notion of log completeness (cf. Definition 15), and/or (ii) the same set of ordering relations (cf. Definition 14) can be inferred when the original net contains one of the constructs. Therefore, no mining algorithm can detect which of the constructs are in the original net. Some examples demonstrating that the problematic constructs are related:

Duplicate Tasks (Sequence vs Parallel vs Choice) Duplicate tasks may appear in *sequential*, *parallel*, or *choice* structures in the WF-net. These duplicate task structures are related because given a log there may be different WF-nets containing duplicate tasks. As an example, see the respective nets N_1, N_2 and N_3 in Figure 3.4. Note that a log containing *only* the trace $XAAY$ would be complete for the three nets N_1, N_2, and N_3. Thus, given this input trace, it is impossible for a mining algorithm to determine which duplicate task structure really exists in the original net.

Invisible Tasks vs Duplicate Tasks WF-nets with the same ordering relations can be created either using invisible tasks or using duplicate tasks. For instance, consider nets N_3 and N_4 in Figure 3.4. These two nets may share an identical complete log. In fact, a log containing *only* the trace $XAAY$ would be complete for all four nets (N_{1-4}) in Figure 3.4.

Invisible Tasks vs Loops Behaviorally equivalent WF-nets can be created either using invisible tasks or using loops. For instance, consider nets N_5 and N_6 in Figure 3.4. These nets generate exactly the same set of log traces.

Invisible Tasks vs Synchronization of XOR-join places See nets N_1 and N_2 in Figure 3.6. The α-algorithm generates the same resulting net for both N_1 and N_2 because these nets are behaviorally equivalent.

Non-Free Choice vs Duplicate Tasks Nets N_1 and N_2 in Figure 3.8 are behaviorally equivalent and, as a result, may have identical complete logs.

Loops vs Invisible Tasks together with Duplicate Tasks Nets with equal sets of ordering relations can be created if loops or invisible tasks in combination with duplicate tasks are used. For instance, see nets N_0 and N_1 in Figure 3.5. Net N_0 has duplicate tasks and invisible tasks in its structure. Net N_1 has two length-one loops, involving tasks A and B. These two nets lead to the same set of ordering relations because, whatever the complete event log, the inferred causal and parallel ordering relations will always be $X \rightarrow_L A$, $X \rightarrow_L B$, $X \rightarrow_L Y$, $B \rightarrow_L Y$, $A \rightarrow_L Y$, and $A||_L B$.

3.5 Extensions to the α-algorithm

The α-algorithm has been extended to remove some of the constraints mentioned here. The works in [11, 28, 29, 80] show how to mine SWF-nets with short loops. In this thesis, we do not elaborate on our extension in [28, 29]. The work in [81] shows how to extend the α^+-algorithm in [28] to also mine a wide range of non-free-choice constructs. In the related work sections (cf. subsections 2.2.7 and 2.2.10), we briefly explain how these extensions work.

3.6 Summary

This chapter explained the basic concepts of control-flow mining: *by extracting ordering relations from an event log, a mining algorithm can (re-)discover*

process models that reflect the behavior in this log. The concepts were explained using the α-algorithm. The α-algorithm is a very simple but powerful mining algorithm. It is proven to correctly mine *SWF-nets without short loops* from *noise-free* complete logs. The α-algorithm is based on four ordering relations: $\rightarrow_L$, $_L$, $\#_L$ and $\|_L$ (cf. Definition 14). These ordering relations are also used by the heuristics of our genetic approach.

Among the limitations of the α-algorithm is its inability to capture short loops, invisible tasks, duplicate tasks, non-local non-free-choice constructs and noise. Invisible tasks are not supported by SWF-nets. Short loops are confused with parallel constructs when inferring the ordering relations. The non-local non-free-choice constructs are also not captured by the log ordering relations. There have been extensions to the α-algorithm, but these extensions still have problems while handling invisible tasks and duplicate tasks. Besides, they are also very sensitive to noise.

The next chapters present genetic algorithms that do not have these limitations and are more robust to noise. The *basic genetic algorithm* (GA) explained in Chapter 4 tackles all problematic constructs, except for the presence of duplicate tasks. As an illustration, a model like the one in Figure 3.10 can be mined by the GA. Figure 3.11 shows the mined model returned by the α-algorithm for a complete log of the model in Figure 3.10. Note that the α-algorithm captures neither the invisible task to skip "Receive License" nor the non-local non-free-choice construct involving the tasks "Attend Classes..." and "Do Practical Exam...". The extension α^{++} [81] correctly mines the non-local non-free-choice in Figure 3.10, but it has the same problems as the α to mine the invisible task. The *duplicate tasks genetic algorithm* (DGA) in Chapter 5 can mine models that also contain duplicate tasks. For instance, the DGA mines models like the one in Figure 3.12, while the α-algorithm (cf. mined model in Figure 3.13) and its extensions are unable to capture the duplication of the tasks "Travel Car" and "Travel Train". The examples in Figure 3.10 and 3.12 are respectively used as running examples in chapters 4 and 5.

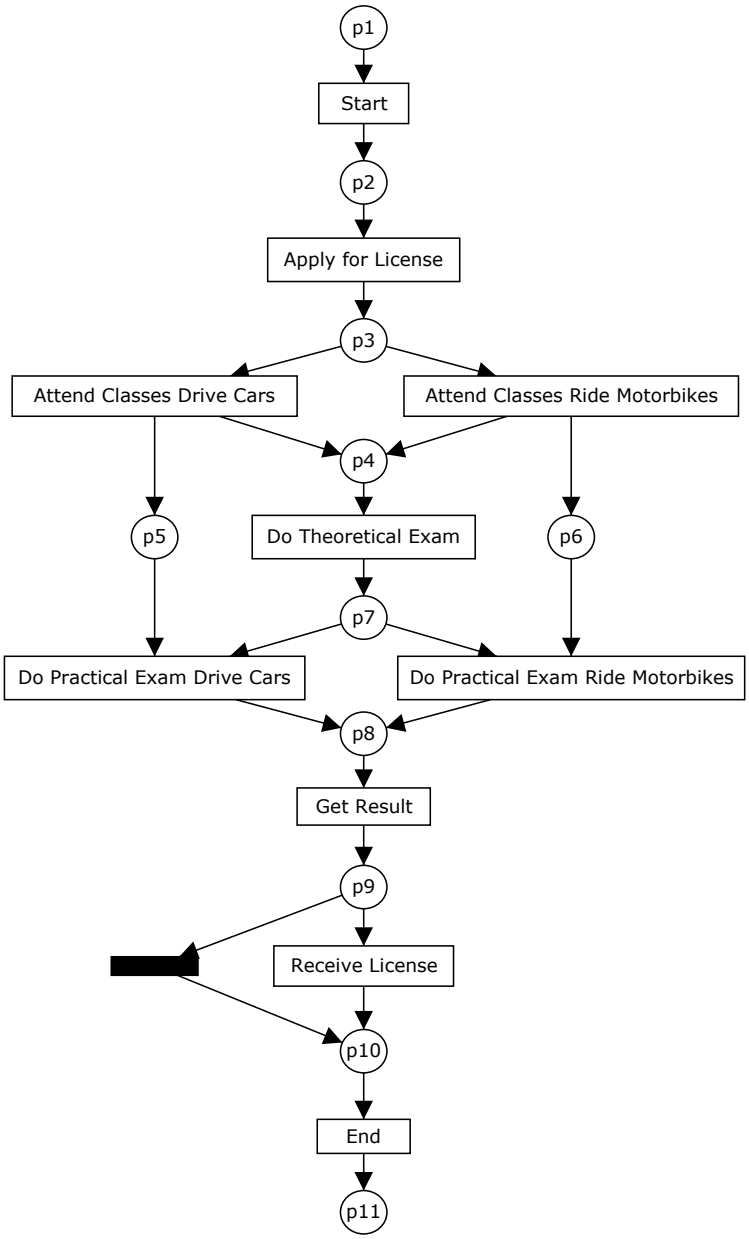


Figure 3.10: Model with non-local non-free-choice and invisible task.

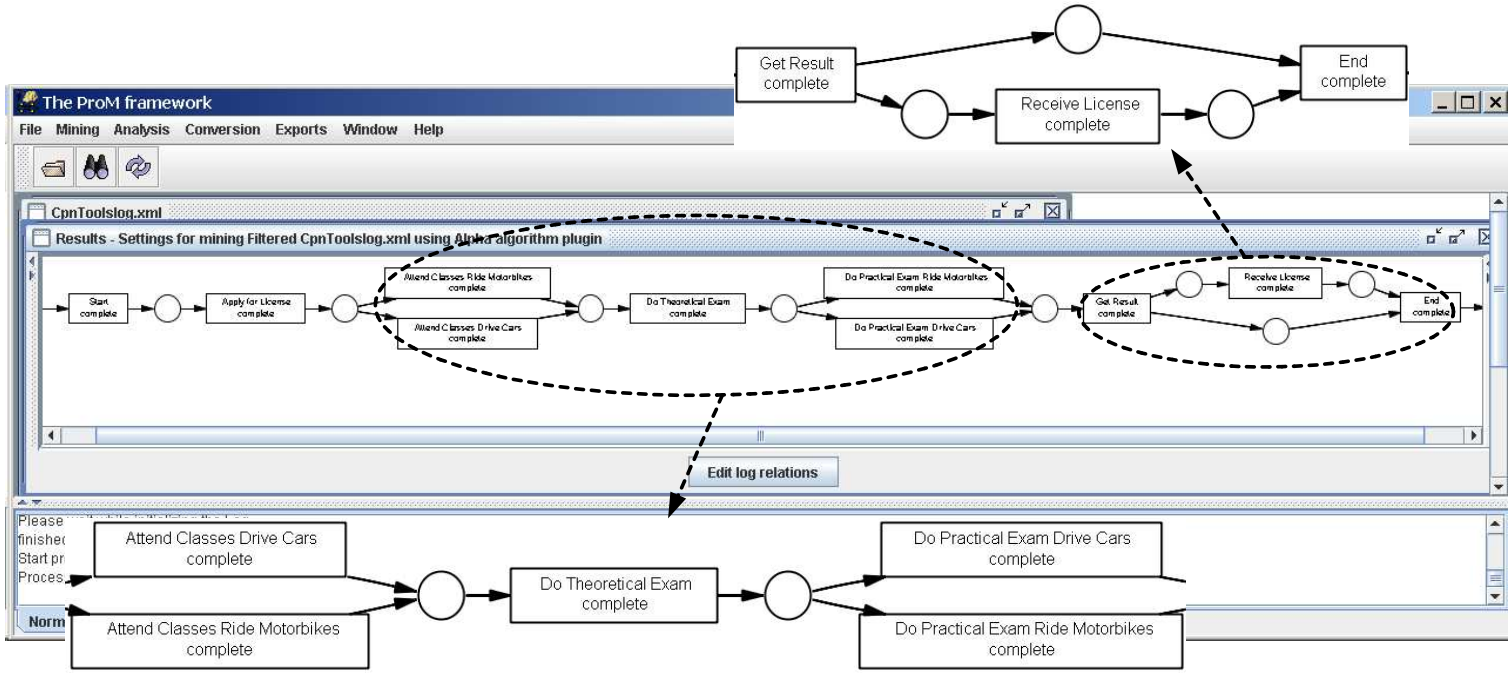


Figure 3.11: The result produced by the α-algorithm for the example in Figure 3.10. Note that the non-local non-free-choice construct involving the tasks "Attend Classes..." and "Do Practical Exam..." as well as the invisible task to skip "Receive License" are not correctly mined.

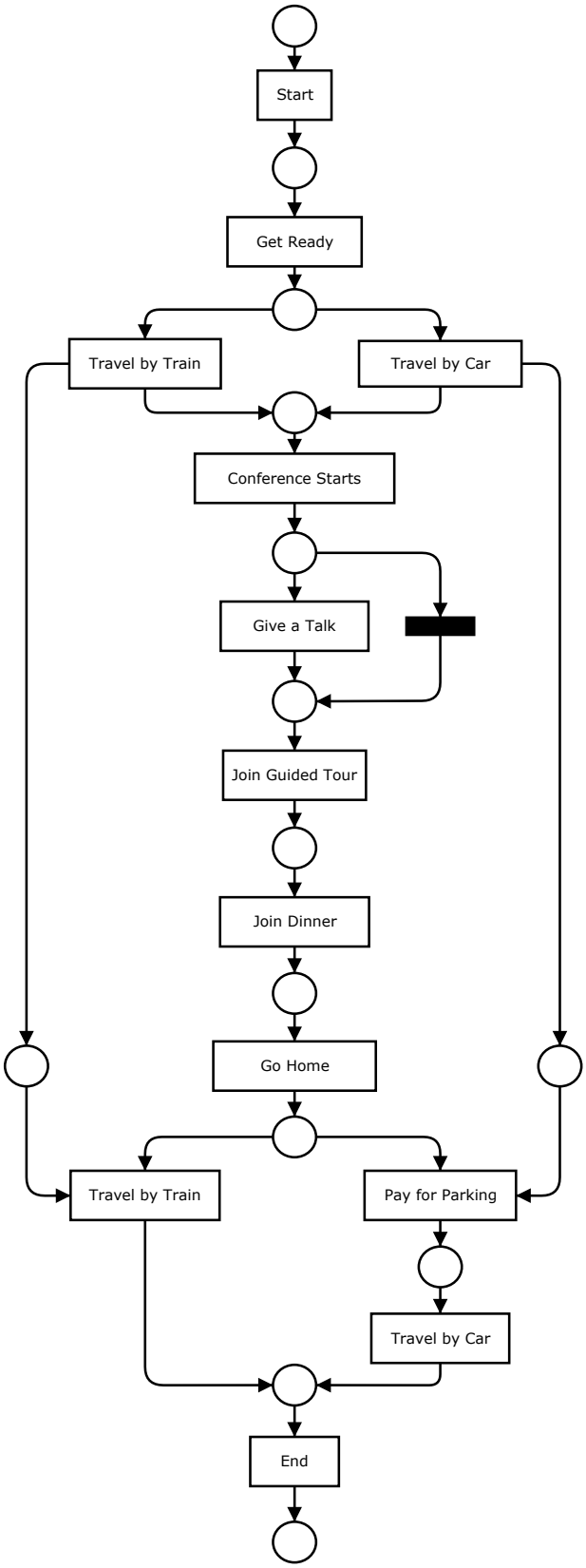


Figure 3.12: Example of a model containing duplicate tasks, non-local non-free choice and invisible task.

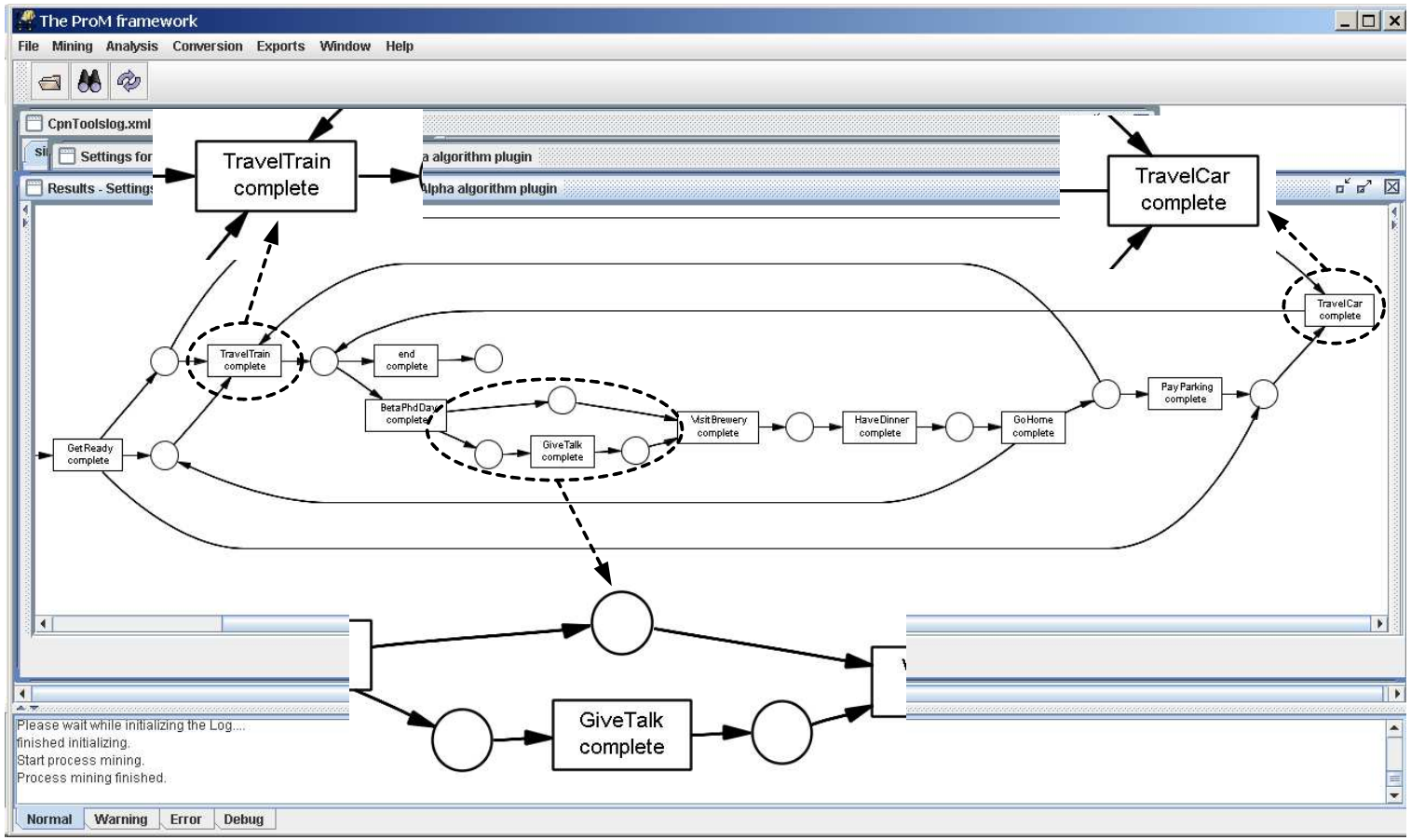


Figure 3.13: The result produced by the α-algorithm for the example in Figure 3.12. There are no duplicates for the tasks "TravelCar" and "TravelTrain", and it is not possible to skip the task "GiveTalk".

Chapter 4

A Genetic Algorithm to Tackle Non-Free-Choice and Invisible Tasks

As indicated in Chapter 1, our aim is to develop a Genetic Algorithm (GA) to mine process models that may contain all the common structural constructs (sequence, choice, parallelism, loops, non-free-choice, invisible tasks and duplicate tasks), while being robust to noisy logs. However, we think that the whole approach can be more easily understood if we explain it in a step-wise fashion. Therefore, we have distributed the different aspects of the GA over chapters 4, 5 and 6. Chapter 4 (the current chapter) shows a GA to tackle *all constructs but duplicate tasks*. This chapter introduces the main concepts behind our GA and the experimental evaluation. Chapter 4 contains material that is used as a basis for the other chapters. Chapter 5 shows how we extend this basic GA to mine event logs from process models with duplicate tasks. Chapter 6 explains the post-processing step we have implemented to clean mined models. But first things first. In the remainder of this chapter we will present the basic genetic algorithm.

Genetic algorithms are adaptive search methods that try to mimic the process of evolution [38, 61]. These algorithms start with an initial *population* of *individuals*. Every individual is assigned a *fitness* measure to indicate its quality. In our case, an individual is a possible process model and the fitness is a function that evaluates how well an individual is able to reproduce the behavior in the log. Populations evolve by selecting the fittest individuals and generating new individuals using *genetic operators* such as crossover (combining parts of two or more individuals) and mutation (random modification of an individual). Additionally, it is common practice to directly copy a number of the best individuals in a current population (the *elite*) to

the next population. This way, one ensures that the best found individuals are kept in future populations. Every individual in a population has an *internal representation*. Besides, like with human beings, the way the internal representation of an individuals is coded is called its *genotype*, but how this representation is shown to the outside world is called *phenotype*. This allows, for instance, that individuals with different genotypes can have the same phenotype.

When using genetic algorithms to mine process models, there are three main issues that need to be addressed. The first is to define the *internal representation*. The internal representation defines the search space of a genetic algorithm. The internal representation that we define and explain in this chapter supports all the problematic constructs, except for duplicate tasks. The second issue is to define the *fitness measure*. In our case, the fitness measure evaluates the quality of a point (individual or process model) in the search space against the event log. A genetic algorithm searches for individuals whose fitness is maximal. Thus, our fitness measure makes sure that individuals with a maximal fitness can parse all the process instances (traces) in the log and, ideally, not more than the behavior that can be derived from those traces. The reason for this is that we aim at mining a model that reflects as close as possible the behavior expressed in the event log. If the mined model allows for lots of extra behavior that cannot be derived from the log, it does not give a precise description of what is actually happening. The third issue relates to the *genetic operators* (crossover and mutation) because they should ensure that all points in the search space defined by the internal representation may be reached when the genetic algorithm runs. This chapter presents a genetic algorithm that addresses these three issues, while assuming the absence of noise and duplicate tasks.

The illustrative example used throughout this chapter is the event log in Table 4.1 and the process model in Figure 4.1. The log shows the event traces (process instances) for four different applications to get a license to ride motorbikes or drive cars. Note that applicants for different types of licenses do the same theoretical exam (task "Do Theoretical Exam") but different practical ones (tasks "Do Practical Exam Drive Cars" or "Do Practical Exam Ride Motorbikes"). In other words, whenever the task "Attend classes Drive Cars" is executed, the task "Do practical Exam Drive Cars" is the only one that can be executed after the applicant has done the theoretical exam. This shows that there is a *non-local* dependency between the tasks "Attend Classes Drive Cars" and "Do Practical Exam Drive Cars", and also between the tasks "Attend Classes Ride Motorbikes" and "Do Practical Exam Ride Motorbikes". The dependency is non-local because it cannot be detected by simply looking at the direct predecessor and successor of those tasks in the

Identifier	Process instance
1	Start, Apply for License, Attend Classes Drive Cars, Do Theoretical Exam, Do Practical Exam Drive Cars, Get Result, End
2	Start, Apply for License, Attend Classes Ride Motorbikes, Do Theoretical Exam, Do Practical Exam Ride Motorbikes, Get Result, Receive License, End
3	Start, Apply for License, Attend Classes Drive Cars, Do Theoretical Exam, Do Practical Exam Drive Cars, Get Result, Receive License, End
4	Start, Apply for License, Attend Classes Ride Motorbikes, Do Theoretical Exam, Do Practical Exam Ride Motorbikes, Get Result, End

Table 4.1: Example of an event log with 4 process instances.

log in Table 4.1, e.g. "Attend Classes Drive Cars" is never followed directly by "Do Practical Exam Drive Cars". Moreover, note that only in some process instances (2 and 3) the task "Receive License" was executed. These process instances point to the cases in which the candidate passed the exams. Based on this log and these observations, process mining tools could be used to retrieve the model in Figure 4.1 [1]. In this case, we are using Petri nets (cf. Subsection 3.1.1) to depict this model. We do so because Petri nets [31, 62] will be used to explain the semantics of our internal representation. Moreover, we use Petri-net-based analysis techniques to analyse the resulting models.

The remainder of the chapter is organized as follows. Section 4.1 explains the internal representation that we use and defines its semantics by mapping it onto Petri nets. Section 4.2 explains the fitness measurement. The fitness measurement benefits the individuals that are complete (can reproduce all the behavior in the log) and precise (do not allow for too much extra behavior that cannot be derived from the event log). Section 4.3 explains the genetic operators crossover and mutation. These operators work based on the task dependencies. Section 4.4 shows how the genetic algorithm works. Section 4.5 discusses the experiments and results. The experiments include synthetic logs. This section also introduces the metrics we used to analyse the results. Section 4.6 has a short summary of this chapter.

[1] Note that simple algorithms, like the α-algorithm, are unable to capture the non-local dependencies and the skipping of task "Receive License" (cf. Figure 4.1)

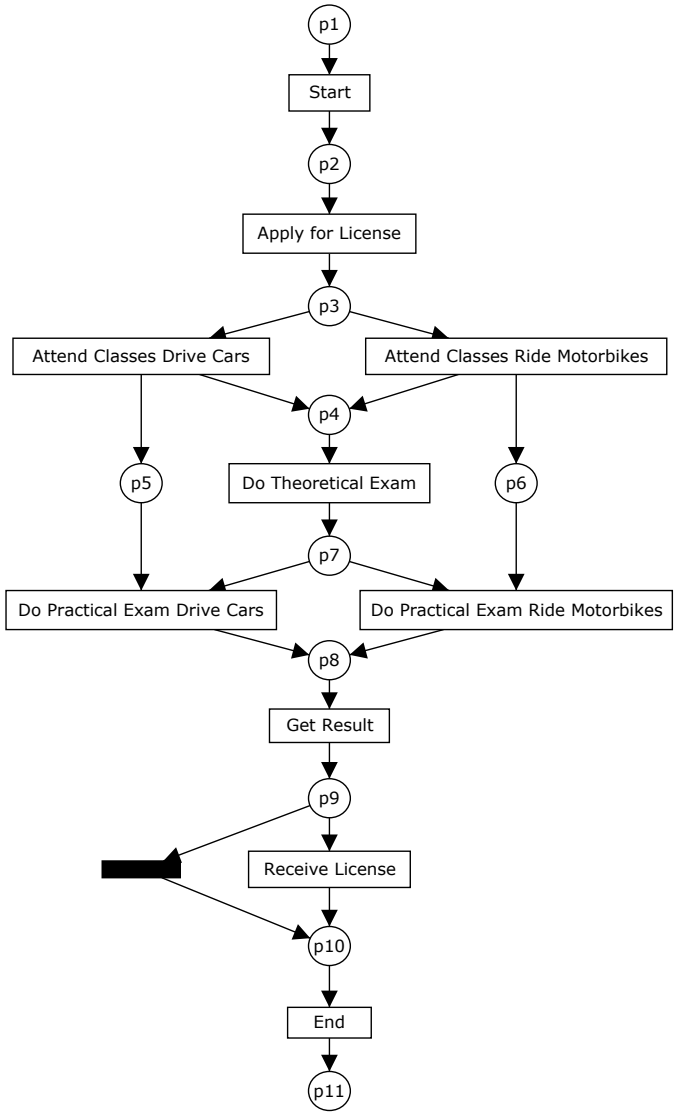


Figure 4.1: Mined net for the log in Table 4.1

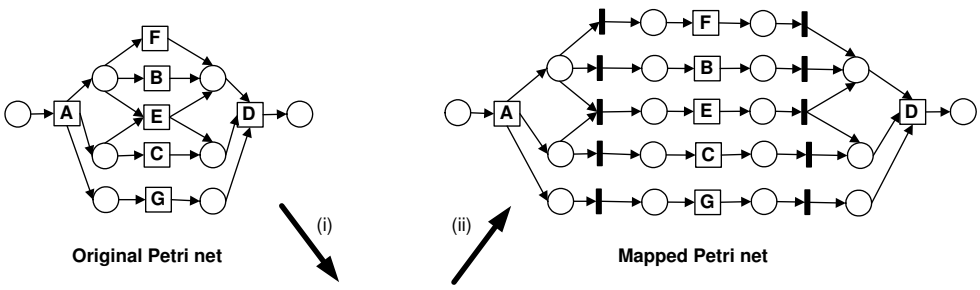
4.1 Internal Representation and Semantics

When defining the internal representation to be used by our genetic algorithm, the main requirement was that this representation should express the dependencies between the tasks in the log. In other words, the model should clearly express which tasks enable the execution of other tasks. Additionally, it is desirable to have an internal representation that is compatible with a formalism to which analysis techniques and tools exist. This way, these techniques could also be applied to the discovered models. Thus, one option would be to directly represent the individual (or process model) as a Petri net [31, 62]. However, such a representation would require determining the number of places in every individual and this is not the core concern. It is more important to show the dependencies between the tasks and the semantics of the split/join tasks. Therefore, we defined an internal representation that is as expressive as Petri nets (from the task dependency perspective) but that only focusses on the tasks. This representation is called *causal matrix*. Figure 4.2 illustrates in (i) the causal matrix that expresses the same task dependencies that are in the "original Petri net". The causal matrix shows which activities [2] enable the execution of other activities via the matching of *input* (I) and *output* (O) condition functions. The sets returned by the condition functions I and O have *subsets* that contain the activities in the model. Activities in a same subset have an XOR-split/join relation, while the different subsets have an AND-split/join relation. Thus, every I and O set expresses a *conjunction of exclusive disjunctions*. Additionally, an activity may appear in more than one subset. For example, consider activity D in the original Petri net in Figure 4.2. The causal matrix states that $I(D) = \{\{F, B, E\}, \{E, C\}, \{G\}\}$ because D is enabled by an AND-join construct that has 3 places. From top to bottom, the first place has a token whenever F *or* B *or* E fires. The second place, whenever E *or* C fires. The third place, whenever G fires. Similarly, the causal matrix has $O(D) = \{\}$ because D is executed last in the model, i.e., no other activity follows D. The following definition formally defines these notions.

Definition 18 (Causal Matrix). *Let LS be a set of labels. A Causal Matrix is a tuple* $CM = (A, C, I, O, Label)$*, where*

- *A is a finite set of activities,*
- *$C \subseteq A \times A$ is the causality relation,*

[2] In the remainder of this document, we use the term "task" when referring to events in a log or transitions in a Petri net, but we use the term "activity" when referring to tasks in a internal representation of an individual.



ACTIVITY	I(ACTIVITY)	O(ACTIVITY)
A	{}	{{F,B,E},{E,C},{G}}
B	{{A}}	{{D}}
C	{{A}}	{{D}}
D	{{F,B,E},{E,C},{G}}	{}
E	{{A}}	{{D}}
F	{{A}}	{{D}}
G	{{A}}	{{D}}

Compact representation of the causal matrix

Figure 4.2: Mapping of a PN with more than one place between two tasks (or transitions).

- $I : A \rightarrow \mathcal{P}(\mathcal{P}(A))$ *is the input condition function,*[3]
- $O : A \rightarrow \mathcal{P}(\mathcal{P}(A))$ *is the output condition function,*
- $Label \in A \rightarrow LS$ *is a labeling function that maps each activity in* A *to a label in* LS*,*

such that

- $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$,[4]
- $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$,
- $C \cup \{(a_o, a_i) \in A \times A \mid a_o \overset{C}{\bullet} = \emptyset \wedge \overset{C}{\bullet} a_i = \emptyset\}$ *is a strongly connected graph,*
- $Label \in A \rightarrow LS$ *is injective* [5]*, i.e.,* $a_1, a_2 \in A$ *and* $Label(a_1) = Label(a_2)$ *implies* $a_1 = a_2$*.*

The set of all causal matrices is denoted by $\mathcal{CM}$*, and a bag of causal matrices is denoted by* $\mathcal{CM}[]$*.*

Any Petri net without duplicate tasks and without more than one place with the same input tasks and the same output tasks can be mapped to a causal

[3] $\mathcal{P}(A)$ denotes the powerset of some set A.

[4] $\bigcup I(a_2)$ is the union of the sets in set $I(a_2)$.

[5] We assume that the function *Label* is injective because we do not allow for duplicate tasks in this chapter. However, since our approach will be extended in Chapter 5 to also mine duplicate tasks, we have added a labelling function.

matrix. Definition 19 formalizes such a mapping. The main idea is that there is a causal relation C between any two tasks t and t' whenever at least one of the *output* places of t is an *input* place of t'. Additionally, the I and O condition functions are based on the input and output places of the tasks. This is a natural way of mapping because the input and output places of Petri nets actually reflect the conjunction of disjunctions that these sets express.

Definition 19 ($\Pi_{PN \to CM}$). *Let $PN = (P, T, F, Label_{PN})$ be a Petri net. The mapping of PN is a tuple $\Pi_{PN \to CM}(PN) = (A, C, I, O, Label)$, where*

- $A = T$,
- $C = \{(t_1, t_2) \in T \times T \mid t_1 \bullet \cap \bullet t_2 \neq \emptyset\}$,
- $I : T \to \mathcal{P}(\mathcal{P}(T))$ *such that* $\forall_{t \in T}\ I(t) = \{\bullet p \mid p \in \bullet t\}$,
- $O : T \to \mathcal{P}(\mathcal{P}(T))$ *such that* $\forall_{t \in T}\ O(t) = \{p \bullet \mid p \in t \bullet\}$,
- $Label = Label_{PN}$.

The semantics of the causal matrix can be easily understood by mapping them back to Petri nets. This mapping is formalized in Definition 20. Conceptually, the causal matrix behaves as a Petri net that contains visible and invisible tasks. For instance, see Figure 4.2. This figure shows (i) the mapping of a Petri net to a causal matrix and (ii) the mapping from the causal matrix to a Petri net. The firing rule for the mapped Petri net is very similar to the firing rule of Petri nets in general (cf. Definition 2). The only difference concerns the invisible tasks. *Enabled invisible tasks can only fire if their firing enables a visible task.* Similarly, a visible task is enabled if all of its input places have tokens or if there exists *a set of* invisible tasks that are enabled and whose firing will lead to the enabling of the visible task. Conceptually, the causal matrix keeps track of the distribution of tokens at a marking in the output places of the visible tasks of the mapped Petri net. The invisible tasks can be seen as "channels" or "pipes" that are only used when a visible task needs to fire. Every causal matrix starts with a token at the start place. Finally, we point out that, in Figure 4.2, although the mapped Petri net does not have the same *structure* of the original Petri net, these two nets are *behaviorally* equivalent (assuming the extended firing rule just mentioned). In other words, given that these two nets initially have a single token and this token is at the start place (i.e., the input place of A), the two nets can generate identical sets of traces. A detailed explanation about the mappings between the different models in this section can be found in Appendix A.

Definition 20 ($\Pi^N_{CM \to PN}$). *Let $CM = (A, C, I, O, Label)$ be a causal matrix. The Petri net mapping of CM is a tuple $\Pi^N_{CM \to PN} = (P, T, F, Label_{PN})$,*

where

- $P = \{i, o\} \cup \{i_{t,s} \mid t \in A \ \wedge \ s \in I(t)\} \cup \{o_{t,s} \mid t \in A \ \wedge \ s \in O(t)\}$,
- $T = A \cup \{m_{t_1,t_2} \mid (t_1, t_2) \in C\}$,
- $F = \{(i, t) \mid t \in A \wedge \ {}^{C}\!\bullet t = \emptyset\} \cup \{(t, o) \mid t \in A \wedge t \stackrel{C}{\bullet} = \emptyset\} \cup \{(i_{t,s}, t) \mid t \in A \wedge s \in I(t)\} \cup \{(t, o_{t,s}) \mid t \in A \wedge s \in O(t)\} \cup \{(o_{t_1,s}, m_{t_1,t_2}) \mid (t_1, t_2) \in C \wedge s \in O(t_1) \wedge t_2 \in s\} \cup \{(m_{t_1,t_2}, i_{t_2,s}) \mid (t_1, t_2) \in C \wedge s \in I(t_2) \wedge t_1 \in s\}$,
- $\forall_{t \in T}, \ Label_{PN}(t) = \begin{cases} Label(t) & \text{if } t \in A, \\ \emptyset & \text{otherwise.} \end{cases}$

In the resulting Petri net, A is the set of *visible* transitions. A transition $m_{t_1,t_2} \in T$ is an *invisible* transition that connects an output place of t_1 to an input place of t_2. As explained before, these invisible transitions are "lazy" because they only fire to enable the visible ones.

4.2 Fitness Measurement

Process mining aims at discovering a process model from an event log. This mined process model should give a good insight into the behavior recorded in the log. In other words, the mined process model should be *complete and precise from a behavioral perspective*. A process model is complete when it can parse (or reproduce) all the event traces in the log. A process model is precise when it cannot parse more than the behavior that can be derived from the traces in the log. The requirement that the mined model should also be precise is important because different models are able to parse all event traces and these models may allow for much more extra behavior that cannot be abstracted from the behavior registered in the log. To illustrate this we consider the nets shown in Figure 4.3. These models can also parse the traces in Table 4.1, but they allow for extra behavior. For instance, both models allow for the applicant to take the exam before attending to classes. The fitness function guides the search process of the genetic algorithm. Thus, the fitness of an individual is assessed by benefiting the individuals that can parse more event traces in the log (the "completeness" requirement) and by punishing the individuals that allow for more extra behavior than the one expressed in the log (the "preciseness" requirement).

To facilitate the explanation of our fitness measure, we divide it into three parts. First, we discuss in Subsection 4.2.1 how we defined the part of the fitness measure that guides the genetic algorithm towards individuals that are complete. Second, we show in Subsection 4.2.2 how we defined the part of the fitness measure that benefits individuals that are precise. Finally, we

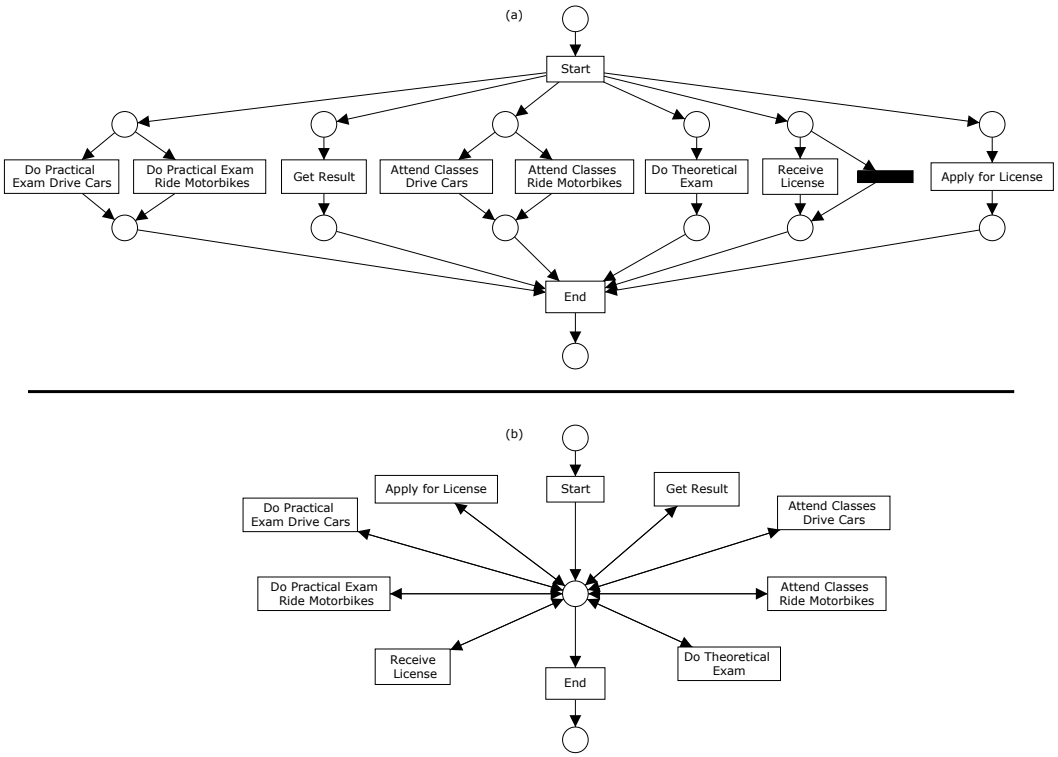


Figure 4.3: Example of nets that can also reproduce the behavior for the log in Table 4.1. The problem here is that these nets allow for extra behavior that is not in the log.

show in Subsection 4.2.3 the fitness measure that our genetic algorithm is using. This fitness measure combines the partial fitness measures that are presented in the subsections 4.2.1 and 4.2.2.

4.2.1 The "Completeness" Requirement

The "completeness" requirement of our fitness measure is based on the parsing of event traces by individuals. For a noise-free log, the perfect individual should have fitness 1. This means that this individual could parse all the traces in the log. Therefore, a natural fitness for an individual to a given log seems to be the number of properly parsed event traces[6] di-

[6]An event trace is properly parsed by an individual if, for an initial marking that contains a single token and this token is at the start place of the mapped Petri net for this individual, after firing the visible tasks in the order in which they appear in the event trace, the end place is the only one to be marked and it has a single token. The notion of proper completion is formalized in Definition 10.

vided by the total number of event traces. However, this fitness measure is too coarse because it does not give an indication about (i) how many parts of an individual are correct when the individual does not properly parse an event trace and (ii) the semantics of the split/join tasks. So, we defined a more refined fitness function: when the activity to be parsed is not enabled, the problems (e.g. number of missing tokens to enable this activity) are registered and the parsing proceeds as if this activity would be enabled. This *continuous* parsing semantics is more robust because it gives a better indication of how many activities do or do not have problems during the parsing of a trace. The partial fitness function that tackles the "completeness" requirement is in Definition 21. The notation used in this definition is as follows. $allParsedActivities(L, CM)$ gives the total number of tasks in the event log L that could be parsed without problems by the causal matrix (or individual) CM. $numActivitiesLog(L)$ gives the number of tasks in L. $allMissingTokens(L, CM)$ indicates the number of missing tokens in all event traces. $allExtraTokensLeftBehind(L, CM)$ indicates the number of tokens that were not consumed after the parsing has stopped plus the number of tokens of the end place minus 1 (because of proper completion). $numTracesLog(L)$ indicates the number of traces in L. The functions $numTracesMissingTokens(L, CM)$ and $numTracesExtraTokensLeftBehind(L, CM)$ respectively indicate the number of traces in which tokens were missing and tokens were left behind during the parsing.

Definition 21 (Partial Fitness - $PF_{complete}$). *Let L be a non-empty event log. Let CM be a causal matrix. Then the partial fitness $PF_{complete} : \mathcal{L} \times \mathcal{CM} \rightarrow (-\infty, 1]$ is a function defined as:*

$$PF_{complete}(L, CM) = \frac{allParsedActivities(L, CM) - punishment}{numActivitiesLog(L)}$$

where

$$punishment = \frac{allMissingTokens(L, CM)}{numTracesLog(L) - numTracesMissingTokens(L, CM) + 1} + \frac{allExtraTokensLeftBehind(L, CM)}{numTracesLog(L) - numTracesExtraTokensLeftBehind(L, CM) + 1}$$

The partial fitness $PF_{complete}$ quantifies how complete an individual is to a given log. The function *allMissingTokens* penalizes (i) nets with XOR-split where it should be an AND-split and (ii) nets with an AND-join where it should be an XOR-join. Similarly, the function *allExtraTokensLeftBehind* penalizes (i) nets with AND-split where it should be an XOR-split and (ii)

nets with an XOR-join where it should be an AND-join. Note that we weigh the impact of the *allMissingTokens* (*allExtraTokensLeftBehind*) function by dividing it by the number of event traces minus the number of event traces with missing tokens (left-behind tokens). The main idea is to promote individuals that correctly parse the most frequent behavior in the log. Additionally, if two individuals have the same *punishment* value, the one that can parse more tasks has a better fitness because its missing and left-behind tokens impact fewer tasks. This may indicate that this individual has more correct I and O condition functions than incorrect ones. In other words, this individual is a better candidate to produce offsprings for the next population (see Subsection 4.3).

Definition 21 may seem rather arbitrary. However, based on many experiments we concluded that this measure has an acceptable performance (in terms of the quality of the mined models). In our tool (see Chapter 7, Section 7.2), we allow for multiple fitness measures.

4.2.2 The "Preciseness" Requirement

The "preciseness" requirement is based on discovering how much extra behavior an individual allows for. To define a fitness measure to punish models that express more than it is in the log is especially difficult because we do not have negative examples to guide our search. Note that the event logs show the allowed (positive) behavior, but they do not express the forbidden (negative) one.

One possible solution to punish an individual that allows for undesirable behavior could be to build the *coverability graph* [62] of the mapped Petri net for this individual and check the fraction of event traces this individual can generate that are not in the log. The main idea in this approach is to punish the individual for every extra event trace it generates. Unfortunately, building the coverability graph for every individual is not very practical (because it is too time consuming) and it is unrealistic to assume that all possible behavior is present in the log.

Because proving that a certain individual is precise is not practical, we use a simpler solution to guide our genetic algorithm towards solutions that have "less extra behavior". We check, for every marking, the *number of visible tasks that are enabled*. Individuals that allow for extra behavior tend to have more enabled activities than individuals that do not. For instance, the nets in Figure 4.3 have more enabled tasks in most reachable markings than the net in Figure 4.1. The main idea in this approach is to benefit individuals that have a smaller amount of enabled activities during the parsing of the log. This is the measure we use to define our second partial fitness function $PF_{precise}$

that is presented in Definition 22. The notation used in this definition is as follows. $allEnabledActivities(L, CM)$ indicates the number of activities that were enabled during the parsing of the log L by the causal matrix (or individual) CM. $allEnabledActivities(L, CM[])$ applies the just explained function $allEnabledActivities(L, CM)$ to every element in the bag of causal matrices (or population) $CM[]$. The function $max(allEnabledActivities(L, CM[]))$ returns the *maximum* value of the amount of enabled activities that individuals in the given population ($CM[]$) had while parsing the log (L).

Definition 22 (Partial Fitness - $PF_{precise}$). *Let L be a non-empty event log. Let CM be a causal matrix. Let $CM[]$ be a bag of causal matrices that contains CM. The partial fitness $PF_{precise} : \mathcal{L} \times \mathcal{CM} \times \mathcal{CM}[] \rightarrow [0, 1]$ is a function defined as*

$$PF_{precise}(L, CM, CM[]) = \frac{allEnabledActivities(L, CM)}{max(allEnabledActivities(L, CM[]))}$$

The partial fitness $PF_{precise}$ gives an indication of how much extra behavior an individual allows for *in comparison to* other individuals in the same population. The smaller the $PF_{precise}$ of an individual is, the better. This way we avoid over-generalizations.

4.2.3 Fitness - Combining the "Completeness" and "Preciseness" Requirements

While defining the fitness measure, we decided that the "completeness" requirement should be more relevant than the "preciseness" one. The reason is that we are mainly interested in precise models that are also complete. The resulting fitness is defined as follows.

Definition 23 (Fitness - F). *Let L be a non-empty event log. Let CM be a causal matrix. Let $CM[]$ be a bag of causal matrices that contains CM. Let $PF_{complete}$ and $PF_{precise}$ be the respective partial fitness functions given in definitions 21 and 22. Let κ be a real number greater than 0 and smaller or equal to 1 (i.e., $\kappa \in (0, 1]$). Then the fitness $F : \mathcal{L} \times \mathcal{CM} \times \mathcal{CM}[] \rightarrow (-\infty, 1)$ is a function defined as*

$$F(L, CM, CM[]) = PF_{complete}(L, CM) - \kappa * PF_{precise}(L, CM, CM[])$$

The fitness F weighs (by κ) the punishment for extra behavior. Thus, if a set of individuals can parse all the traces in the log, the one that allows for less extra behavior will have a higher fitness value. For instance, assume a population with the corresponding individual for the net in Figure 4.1 and

the corresponding individuals for the nets in Figure 4.3. If we calculate the fitness F of these three individuals with respect to the log in Table 4.1, the individual in Figure 4.1 will have the highest fitness value among the three and the individual in Figure 4.3(b), the lowest fitness value.

4.3 Genetic Operators

The genetic operators make sure that all points in the search space defined by the internal representation can potentially be reached while the genetic algorithm executes. Our genetic algorithm has two genetic operators: crossover and mutation. The *crossover* operator is explained in Subsection 4.3.1. The *mutation* operator is explained in Subsection 4.3.2.

4.3.1 Crossover

Crossover is a genetic operator that aims at *recombining existing material* in the current population. In our case, this material is the current causality relations (cf. Definition 18) in the population. Thus, the crossover operator used by our genetic algorithm should allow for the complete search of the space defined by the existing causality relation in a population. Given a set of causality relations, the search space contains all the individuals that can be created by any combination of a subset of the causality relations in the population. Therefore, our crossover operator allows an individual to: lose activities from the subsets in its I/O condition functions (but not necessarily causality relations because a same activity may be in more than one subset of an I/O condition function), add activities to the subsets in its I/O condition functions (again, not necessarily causality relations), exchange causality relations with other individuals, incorporate causality relations that are in the population but are not in the individual, lose causality relations, decrease the number of subsets in its I/O condition functions, *and/or* increase the number of subsets in its I/O condition functions. The *crossover rate* determines the probability that two parents undergo crossover. The crossover point of two parents is a randomly chosen activity. The pseudo-code for the crossover operator is as follows:

Pseudo-code:
input: Two parents ($parent_1$ and $parent_2$), crossover rate.
output: Two possibly recombined offsprings ($offspring_1$ and $offspring_2$).

1. $offspring_1 \longleftarrow parent_1$ and $offspring_2 \longleftarrow parent_2$.
2. With probability "crossover rate" do:

(a) Randomly select an activity a to be the crossover point of the offsprings.

(b) Randomly select a swap point sp_1 for $I_1(a)$ [7]. The swap point goes from position 0 to $n-1$, where n is the number of subsets in the condition function $I_1(a)$.

(c) Randomly select a swap point sp_2 for $I_2(a)$.

(d) $remainingSet_1(a)$ equals subsets in $I_1(a)$ that are between position 0 and sp_1 (exclusive).

(e) $swapSet_1(a)$ equals subsets in $I_1(a)$ whose position equals or bigger than sp_1.

(f) Repeat steps 2d and 2e but respectively use $remainingSet_2(a)$, $I_2(a)$, sp_2 and $swapSet_2(a)$ instead of $remainingSet_1(a)$, $I_1(a)$, sp_1 and $swapSet_1(a)$.

(g) FOR every subset S_2 in $swapSet_2(a)$ do:

i. With equal probability perform *one of the following* steps:

A. Add S_2 as a new subset in $remainingSet_1(a)$.

B. Join S_2 with an existing subset X_1 in $remainingSet_1(a)$.

C. Select a subset X_1 in $remainingSet_1(a)$, remove the elements of X_1 that are also in S_2 and add S_2 to $remainingSet_1(a)$.

(h) Repeat Step 2g but respectively use S_1, $swapSet_1(a)$, X_2 and $remainingSet_2(a)$ instead of S_2, $swapSet_2(a)$, X_1 and $remainingSet_1(a)$.

(i) $I_1(a) \longleftarrow remainingSet_1(a)$ and $I_2(a) \longleftarrow remainingSet_2(a)$.

(j) Repeat steps 2b to 2h but use $O(a)$ instead of the $I(a)$.

(k) Update the related activities to a.

3. Return $offspring_1$ and $offspring_2$.

Note that, after crossover, the number of causality relations for the whole population remains constant, but how these relations appear in the offsprings may be different from the parents. Moreover, the offsprings may be different even when both parents are equal. For instance, consider the situation in which the crossover operator receives as input two parents that are equal to the causal matrix in Figure 4.2. Assume that (i) the crossover

[7] We use the notation $I_n(a)$ to get the subset returned by the input condition function of activity a in individual n. When there is a single individual, n will be omitted. In this pseudo-code, the individuals are the offsprings.

point is the activity D, (ii) we are doing crossover over the input condition function $I(D) = \{\{F, B, E\}, \{E, C\}, \{G\}\}$, and (iii) the swap points are $sp_1 = 1$ and $sp_2 = 2$. Then, we have that the $remainingSet_1(D) = \{\{F, B, E\}\}$, the $swapSet_1(D) = \{\{E, C\}, \{G\}\}$, the $remainingSet_2(D) = \{\{F, B, E\}, \{E, C\}\}$, the $swapSet_2(D) = \{\{G\}\}$. Let us first crossover the subsets in the $swapSet_2(D)$ with the $remainingSet_1(D)$. During the crossover, the genetic algorithm randomly chooses to merge the subset $S_2 = \{G\}$ in the $swapSet_2(D)$ with the existing subset $X_1 = \{F, B, E\}$. In a similar way, while swapping the subsets in $swapSet_1(D)$ with the $remainingSet_2(D)$, the algorithm randomly chooses (i) to insert the subset $S_1 = \{E, C\}$ and remove activity E from the subset $X_2 = \{F, B, E\}$, and (ii) to insert the subset $S_1 = \{G\}$ as a new subset in the $remainingSet_2(D)$. The result is that $I_1(D) = \{\{F, B, E, G\}\}$ and $I_2(D) = \{\{F, B\}, \{E, C\}, \{G\}\}$. The output condition functions $O_1(D)$ and $O_2(D)$ do not change after the crossover operator because the activity D does not have any output activity. Note that, in this example, the two resulting offsprings are different from each other and from the parents, even though the parents are equal.

After the crossover, the mutation operator takes place as described in the next subsection.

4.3.2 Mutation

The mutation operator aims at *inserting new material* in the current population. In our case, this means that the mutation operator may change the existing causality relations of a population. Thus, our mutation operator performs one of the following actions to the I/O condition functions of an activity in an individual: (i) randomly choose a subset and add an activity (in A) to this subset, (ii) randomly choose a subset and remove an activity out of this subset, *or* (iii) randomly redistribute the elements in the subsets of I/O into new subsets. For example, consider the input condition function of activity D in Figure 4.2. $I(D) = \{\{F, B, E\}, \{E, C\}, \{G\}\}$ can be mutated to (i) $\{\{F, B, E\}, \{E, C\}, \{G, D\}\}$ if activity D is added to the subset $\{G\}$, (ii) $\{\{F, B, E\}, \{C\}, \{G\}\}$ if activity E is removed from the subset $\{E, C\}$, or (iii) $\{\{F\}, \{E, C, B\}, \{G\}, \{E\}\}$ if the elements in the subsets of the original $I(D)$ (i.e., the elements in the list "F, B, E, E, C, G") are randomly redistributed in *four* randomly chosen new subsets. Every activity in an offspring may undergo mutation with the probability determined by the *mutation rate*. The pseudo-code for the mutation operator is as follows:

Pseudo-code:
input: An individual, mutation rate.

output: A possibly mutated individual.

1. For every activity a in the individual do:
   (a) With probability *mutation rate* do *one* of the following operations for the condition function $I(a)$:
      i. Select a subset X in $I(a)$ and add an activity a' to X, where a' belongs to the set of activities in the individual.
      ii. Select a subset X in $I(a)$ and remove an activity a' from X, where a' belongs to X. If X is empty after a' removal, exclude X from $I(a)$.
      iii. Redistribute the elements in $I(a)$. [8]
   (b) Repeat Step 1a, but use the condition function $O(a)$ instead of $I(a)$.
   (c) Update the related activities to a.

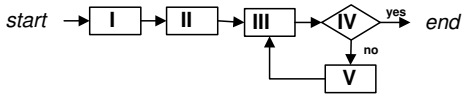
As the reader may already have noticed, both the crossover and the mutation operators perform a repairing operation at the end of their executions. The "update the related activities" operation makes sure that an individual is still compliant with the Definition 18 after undergoing crossover and/or mutation.

4.4 Algorithm

After defining the representation, fitness and genetic operators, we define the Genetic Algorithm (GA). The GA has five main steps (see Figure 4.4). *Step I* simply reads the input event log. Based on the set of tasks in this event log, *Step II* builds a number m of individuals as the initial population. The initial population can be built via a completely random process or by using heuristics. More details about Step II are in Subsection 4.4.1. Once the initial population is built, *Step III* of the GA calculates the fitness for every individual in the population. The fitness is the one given in Definition 23 (see Section 4.2.3, page 62). The *Step IV* checks if the algorithm should stop or not. The mining algorithm stops when (i) it computes n generations, where n is the maximum number of generations; or (ii) the fittest individual has not changed for $n/2$ generations in a row. If the GA stops, it returns the current population. If it does not stop, then *Step V* takes place. This step uses *elitism*, *crossover* and *mutation* to build the individuals of the next

[8] Details about this step: (i) Get a list l with all the elements in the subsets of $I(a)$; (ii) Remove all subsets of $I(a)$ (i.e., make $I(a) = \emptyset$); (iii) Create n sets ($1 \leq n \leq |l|$) and randomly distribute the elements in the list l into these n sets; and (iv) Insert these n sets into $I(a)$.

generation. A percentage of the best individuals (the *elite*) is directly copied to the next population. The other individuals in the population are generated via crossover (see Subsection 4.3.1) and mutation (see Subsection 4.3.2). Two parents produce two offsprings. To select one parent, a *tournament* is played in which five individuals in the population are randomly drawn and the fittest one always wins. Again, Step III is executed to assign a fitness measure to every individual in this new population. The iteration over steps III, IV and V continues until one of the stopping conditions holds and the current population is returned. Here it is worth to make a comment regarding the returned population. During the GA steps executions, the individuals may contain arcs that are never used. Thus, before returning the current population, the GA removes the unused arcs from the individuals. These arcs can be seen as the invisible tasks, in the mapped Petri nets, that never fired. The following subsection provides details about the initial population (Step II).



Step	Description
I	*Read event log*
II	*Build the initial population*
III	*Calculate fitness of the individuals in the population*
IV	*Stop and return the fittest individuals?*
V	*Create next population – use elitism and genetic operators*

Figure 4.4: Main steps of our genetic algorithm.

4.4.1 Initial Population

The initial population is randomly built by the genetic algorithm. As explained in Section 4.1, individuals are causal matrices. When building the initial population, we ensure that the individuals comply with Definition 18. Given a log, all individuals in any population of the genetic algorithm have the same set of activities A. This set contains one activity for every task that appear in the log. The setting of the causality relation C can be done via a completely random approach or a heuristic one. The random approach uses a 50% probability for establishing (or not) a causality relation between two activities in A. The heuristic approach uses the information in the log to determine the probability that two activities are going to have a causality relation set. In a nutshell, the heuristics works as follows: the more often

an activity a_1 is directly followed by an activity a_2 (i.e. the subtrace "a_1a_2" appears in traces in the log), the higher the probability that individuals are built with a causality relation from a_1 to a_2 (i.e., $(a_1, a_2) \in C$). This heuristic way of building an individual is based on the work presented in [79]. Subsection "Heuristics to Build the Causality Relation of a Causal Matrix", on page 68, provides more details about the heuristic approach. Once the causality relations of an individual are determined, the condition functions I and O are randomly built. This is done by setting a maximum size n for any input or output condition function set of an activity a in the initial population[9]. Every activity a_1 that causally precedes an activity a_2, i.e. $(a_1, a_2) \in C$, is randomly inserted in *one or more* subsets of the input condition function of a_2. A similar process is done to set the output condition function of an activity[10]. In our case, we set the number of distinct activities in the log as the maximum size for any input/output condition function set in the initial population[11]. As a result, the initial population can have any individual in the search space defined by a set of activities A that satisfies the constraints for the size of the input/output condition function sets. Note that the more distinct tasks a log contains, the larger this search space. Finally, we emphasize that no further limitations to the input/output condition functions sets are made in the other GA's steps. Therefore, during the "Step V" in the Figure 4.4, these sets can increase or shrink as the population evolves.

Heuristics to Build the Causality Relation of a Causal Matrix

When applying a genetic algorithm to a domain, it is common practice to "give a hand" to the genetic algorithm by using well-known heuristics (in this domain) to build the initial population [38]. Studies show that the use of heuristics often does not alter the end result (if the genetic algorithm runs for infinite amount of time), but it may speed the early stages of the evolution. The GAs that use heuristics are called *hybrid genetic algorithms*.

In our specific domain - process mining - some heuristics have proven to give reasonable solutions when used to mine event logs. These heuristics are mostly based on *local information in the event-log*. Thus, these heuristics can be used by the GA to build an initial population that has many of the correct *local* causality relations. It makes sense to use heuristics because most of the causality relations in process models can be detected based on local information. However, note that the *non-local* causality dependencies

[9]Formally: $\forall_{a \in A}[|I(a)| \leq n \ \wedge \ |O(a)| \leq n]$.

[10]Formally: $\forall_{a_1,a_2 \in A,(a_1,a_2) \in C}[\exists i \in I(a_2) : a_1 \in i]$ and $\forall_{a_1,a_2 \in A,(a_1,a_2) \in C}[\exists o \in O(a_1) : a_2 \in o]$.

[11]Formally: $\forall_{a \in A}[|I(a)| \leq |A| \wedge |O(a)| \leq |A|]$.

will not be in the initial population. In other words, the mutation operator will be responsible for creating these non-local causality dependencies. Due to its similarities to other related work (cf. Chapter 2), we use the heuristics in [79] to guide the setting of the causality relations in the individuals of the initial population. These heuristics are based on the *dependency measure*.

The dependency measure basically indicates how strongly a task depends (or is caused) by another task. The more often a task t_1 *directly* precedes another task t_2 in the log, and the less often t_2 *directly* precedes t_1, the stronger is the dependency between t_1 and t_2. In other words, the more likely it is that t_1 is a *cause* to t_2. The dependency measure is given in Definition 24. The notation used in this definition is as follows. $l2l : T \times T \times \mathcal{L} \rightarrow \mathbb{N}$ is a function that detects length-two loops. $l2l(t_1, t_2, t_1)$ gives the number of times that the substring "$t_1t_2t_1$" occurs in a log. $follows : T \times T \times \mathcal{L} \rightarrow \mathbb{N}$ is a function that returns the number of times that a task is directly followed by another one. That is, how often the substring "t_1t_2" occurs in a log.

Definition 24 (Dependency Measure - D). *Let L be an event log. Let T be the set of tasks in L. Let t_1 and t_2 be two tasks in T. The dependency measure $D : T \times T \times \mathcal{L} \rightarrow \mathbb{R}$ is a function defined as:*

$$D(t_1, t_2, L) = \begin{cases} \frac{l2l(t_1,t_2,L)+l2l(t_2,t_1,L)}{l2l(t_1,t_2,L)+l2l(t_2,t_1,L)+1} & \text{if } t_1 \neq t_2 \text{ and } l2l(t_1, t_2, L) > 0, \\ \frac{follows(t_1,t_2,L)-follows(t_2,t_1,L)}{follows(t_1,t_2,L)+follows(t_2,t_1,L)+1} & \text{if } t_1 \neq t_2 \text{ and } l2l(t_1, t_2, L) = 0, \\ \frac{follows(t_1,t_2,L)}{follows(t_1,t_2,L)+1} & \text{if } t_1 = t_2. \end{cases}$$

Observe that the dependency relation distinguishes between tasks in short loops (length-one and length-two loops) and tasks in parallel. Moreover, the "+1" in the denominator is used to benefit more frequent observations over less frequent ones. For instance, if a length-one-loop "tt" happens only once in the log L, the dependency measure $D(t, t, L) = 0.5$. However, if this same length-one-loop would occur a hundred times in the log, $D(t, t, L) = 0.99$. Thus, the more often a substring (or pattern) happens in the log, the stronger the dependency measure.

Once the dependency relations are set for the input event log, the genetic algorithm uses it to randomly build the causality relations for every individual in the initial population. The pseudo-code for this procedure is the following:

Pseudo-code:
input: An event-log L, an odd power value p and a dependency function D.
output: A causality relation C.

1. $T \longleftarrow$ set of tasks in L.
2. $C \longleftarrow \emptyset$.
3. FOR every tuple (t_1, t_2) in $T \times T$ do:
   (a) Randomly select a number r between 0 (inclusive) and 1.0 (exclusive).
   (b) IF $r < D(t_1, t_2, L)^p$ then:
      i. $C \longleftarrow C \cup \{(t_1, t_2)\}$.
4. Return the causality relation C.

Note that we use an odd power value p to control the "influence" of the dependency measure in the probability of setting a causality relation. Higher values for p lead to the inference of fewer causality relations among the tasks in the event log, and vice-versa.

4.5 Experiments and Results

This section explains how we conducted the experiments and, most important of all, how we analyzed the quality of the models that the genetic algorithm mined. To conduct the experiments we needed (i) to implement our genetic algorithm and (ii) a set of event logs. The *genetic algorithm* described in this chapter *is implemented as the "Genetic algorithm plug-in"* in the ProM framework (see Figure 7.1 for a screenshot). Chapter 7 provides more details about ProM and the genetic algorithm plug-in. The *logs* used in our experiments are *synthetic*. In brief, we built the model (or copied it from some related work) and simulated it to create a synthetic event log [12]. We then run the genetic algorithm over these sets of logs. Once the genetic algorithm finished the mining process, we analyzed the results. A genetic algorithm run is successful whenever it finds an individual that is *behaviorally* complete and precise (see Section 4.2 for details). Thus, at first sight, the natural way to check for this seemed to be a direct comparison of the causal matrix of the original model (the one that was simulated to create the synthetic event logs) and the causal matrix of the individual that was mined by the genetic algorithm. However, this is not a good evaluation criterion because there are different ways to model the exact behavior expressed in a log. For instance, consider the net in Figure 4.6. This net produces exactly the same behavior as the one in Figure 4.1. However, their causal matrices are different. For this reason, we defined evaluation metrics that check equality both from a

[12] Chapter 8 provides more details about how the synthetic logs used in this thesis were created.

behavioral and from a structural perspective. The remainder of this section is divided in three parts: (i) Subsection 4.5.1 motivates and explains the metrics we have used to assess the quality of the mined models, (ii) Subsection 4.5.2 describes the experiments setup, and (iii) Subsection 4.5.3 contains the results.

4.5.1 Evaluation

The genetic algorithm searches for models that are *complete* and *precise* (see Section 4.2). Therefore, when evaluating our results, we should check if the mined models are indeed complete and precise. Furthermore, even when the mined models are not complete and/or precise, we should be able to assess how much correct material they contain. This is important because we do not let the genetic algorithm run for an "infinite" amount of time. Thus, even when the mined model is not complete and precise, it is important to know if the genetic algorithm is going in the right direction.

In our experiments, we have three elements: (i) the *original model* that is used to build the synthetic event log, (ii) the synthetic *event log* itself, and (iii) the *mined model* (or individual). Thus, to analyse our results, we have defined metrics that are based on two or more of these elements.

Checking for Completeness

To check for completeness, we only need the event log and the mined model. Recall that a model is complete when it can parse all the traces in the log without having missing tokens or leaving tokens behind. So, completeness can be verified by calculating the partial fitness $PF_{complete}$ (see Definition 21) for the event log and the mined model. Whenever $PF_{complete} = 1$, the mined model is complete. Moreover, even when the mined model has $PF_{complete} < 1$, this measure gives an indication of the quality of the mined model with respect to completeness.

Checking for Preciseness

To check for preciseness, we need the original model, the event log and the mined model[13]. The main reason why we could not define metrics only based on the event log and the mined model, or the original model and the mined model, is because it is unrealistic to assume that the event log has all the possible traces that the original model can generate. In other words, it is unrealistic to assume that the log contains all possible event traces. Recall that a model is precise when it does not allow for more behavior than the one

[13]Note that in reality we do not know the original (or initial) model in many situations. However, the only way to evaluate our results is to assume an initial model. Without an initial model, it is impossible to judge preciseness. In other words, there could be over-fitting or over-generalization, but it would be impossible to judge this.

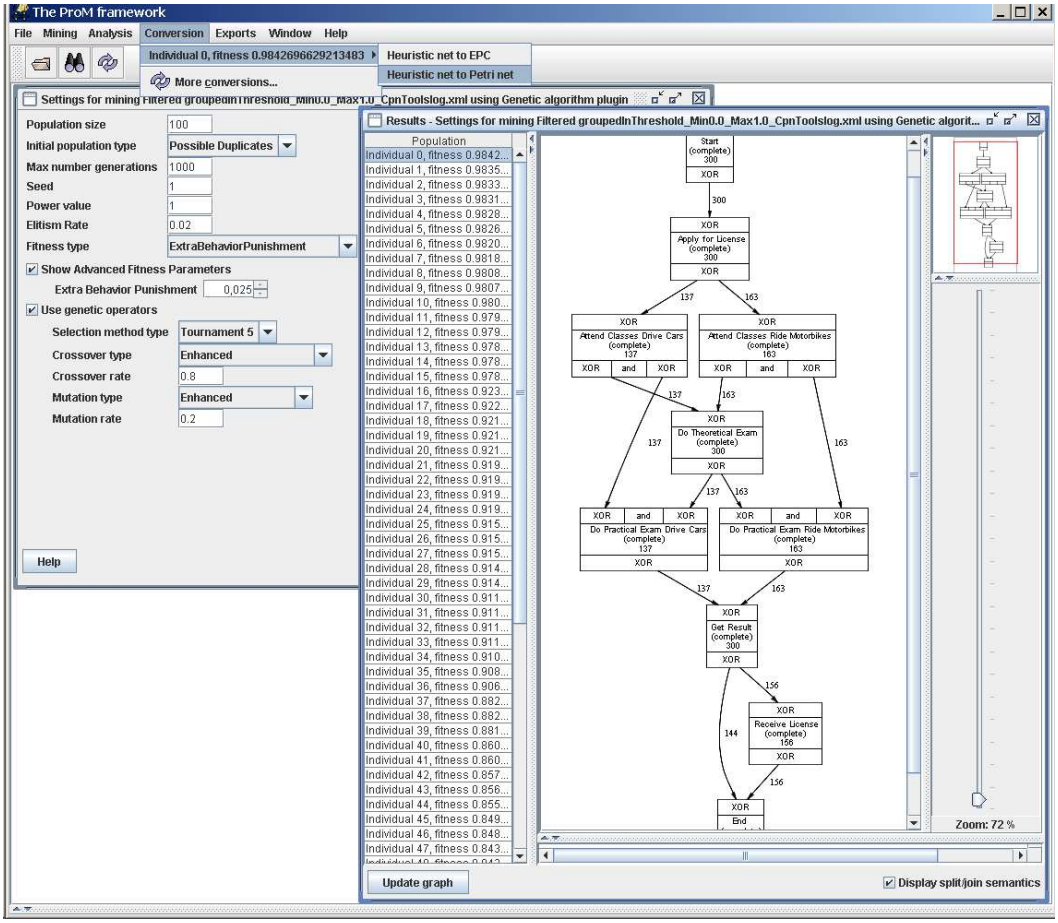


Figure 4.5: Screenshot of the "Genetic algorithm plug-in" in the ProM framework. This screenshot shows the result of mining an event log like the one in Table 4.1. This log has 300 process instances in total. The left-side window shows the configuration parameters (see Subsection 4.5.2). The right-side window shows the best mined individual (or causal matrix). Additionally, in the menu bar we show how to convert this individual (called "Heuristics Net" in the ProM framework) to a Petri net.

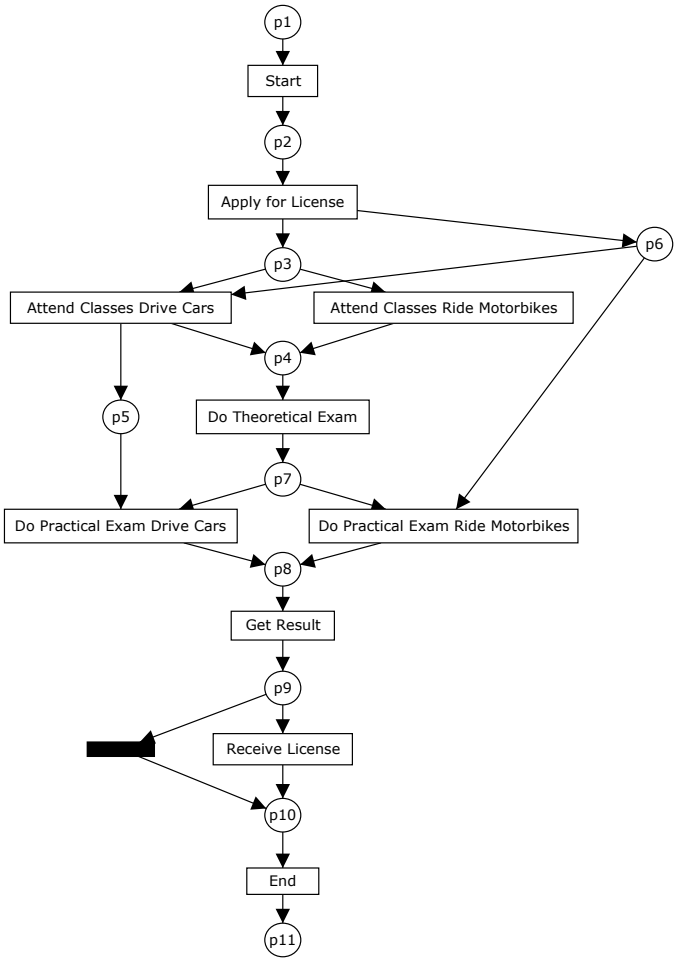


Figure 4.6: Another mined net for the log in Table 4.1. Note that this net is behaviorally equivalent to the net in Figure 4.1, although they are structurally different because the place "p6" has different input and output tasks in the two nets.

expressed in the log. Thus, if the log would be exhaustive, a possible metric to check for this preciseness could be to divide the number of traces that are in the log and that the mined model can generate by the amount of traces that the mined model can generate. Clearly, a precise mined model could not generate more traces than the ones in the log. Note that this metric would be based on the event log and the mined model. For similar reasons, metrics based on the *mined and original models* only would also be possible if the log would be exhaustive. For instance, we could compare the coverability graphs [62] of mapped Petri nets of the mined and the original models. In this case, the mined model would be precise whenever the coverability graphs would be equal. Note that sophisticated notions such as bisimulation [60] and branching bisimulation [41] could also be used. However, none of these metrics are suitable because in real-life applications the log does not hold all possible traces.

Note that a process model with n parallel tasks can generate $n!$ possible traces. In a similar way, a model with n independent choices can lead to 2^n possible traces. However, it is unrealistic to expect the logs to show all the possible interleaving situations. Therefore, to analyse the preciseness of a mined models, one needs the original model and the respective event log given as input to the GA. For instance, consider the situation illustrated in Figure 4.7. This figure shows the original model ("OriginalModel"), two synthetic logs ("Log1" and "Log2") and their respective mined models ("MinedModel1" and "MinedModel2"). "Log1" shows that the tasks A, B, C and D are (i) always executed after the task X and before the task Y and (ii) independent of each other. Thus, we can say that the "MinedModel1" is precise with respect to the behavior observed in the "Log1". However, note that the "MinedModel1", although precise, can generate more traces than the ones in the "Log1". A similar reasoning can be done for the "Log2" and the "MinedModel2". Moreover, the coverability graph of the "MinedModel2" is different from the one of the "OriginalModel". Actually, based on "Log2", the "MinedModel2" is more precise than the "OriginalModel". This illustrates that, when assessing how close the behavior of the mined and original models are, we have to consider the event log that was used by the genetic algorithm. Therefore, we have defined two metrics to quantify how similar the behavior of the original model and the mined model are *based on the event log used during the mining process.*

The two metrics are the *behavioral precision* (B_P) and the *behavioral recall* (B_R). Both metrics are based on the parsing of an event log by the mined model and by the original model. The B_P and B_R metrics are respectively formalized in definitions 25 and 26. These metrics basically work by checking, for the continuous semantics parsing of every task in every process

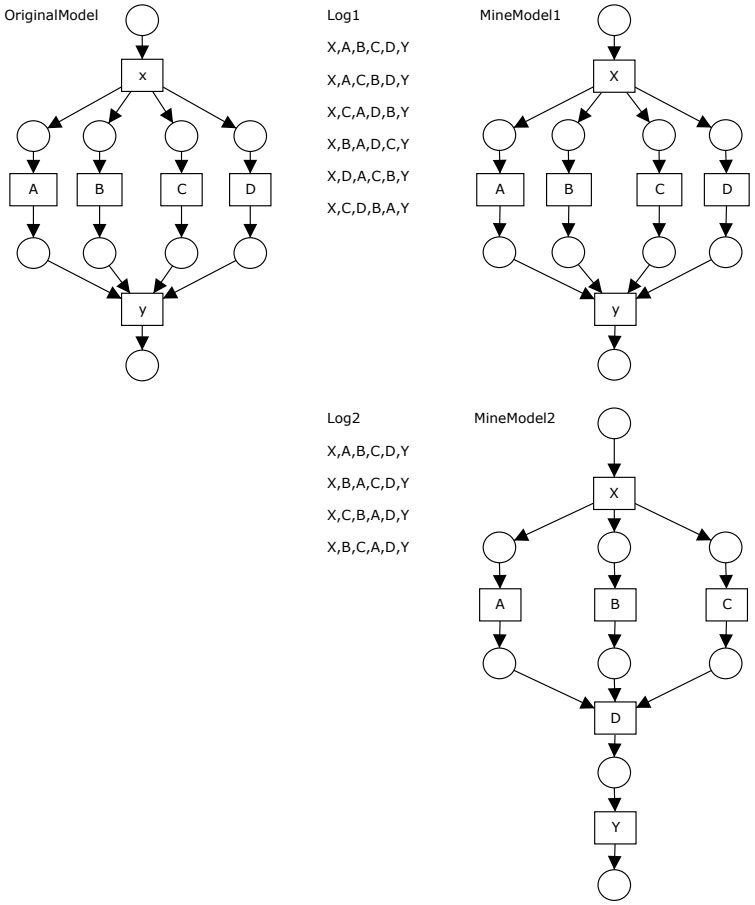


Figure 4.7: Example of two mined models that are complete and precise with respect to the logs, but both mined models can generate more traces than the ones in the log. Additionally, the coverability graph of the "MinedModel2" is different from the one of the "OriginalModel".

instance of the event log, how many activities are enabled in the mined model and how many are enabled in the original model. The more enabled activities the models have in common while parsing the traces, the more similar their behaviors are with respect to the event log. The behavioral precision B_P checks how much behavior is allowed by the mined model that is not by the original model. The behavioral recall B_R checks for the opposite. Additionally, both metrics take into account how often a trace occurs in the log. This is especially important when dealing with logs in which some paths are more likely than others, because deviations corresponding to infrequent paths are less important than deviations corresponding to frequent behavior. Note that, assuming a log generated from an original model and a *complete* mined model for this log, we can say that the closer their B_P and B_R are to 1, the more similar their behaviors. More specifically, we can say that:

- The mined model is *as precise as* the original model whenever B_P and B_R are equal to 1. This is exactly the situation illustrated in Figure 4.7 for the "OriginalModel", the "Log1" and the "MinedModel1".
- The mined model is *more precise than* the original model whenever $B_P = 1$ and $B_R < 1$. For instance, see the situation illustrated in Figure 4.7 for the "OriginalModel", the "Log2" and the "MinedModel2".
- The mined model is *less precise than* the original model whenever $B_P < 1$ and $B_R = 1$. For instance, see the situation illustrated for the original model in Figure 4.1, the log in Figure 4.1, and the mined models in Figure 4.3.

Definition 25 (Behavioral Precision - B_P). [14] *Let L be an event log. Let CM_o and CM_m be the respective causal matrices for the original (or base) model and for the mined one. Let the ranges of the labelling functions of the original and mined models be the same, i.e., $rng(Label_o) = rng(Label_m)$. Then the behavioral precision $B_P : \mathcal{L} \times \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ is a function defined as:*

$$B_P(L, CM_o, CM_m) =$$

$$\frac{\sum_{\sigma \in L} \left(\frac{L(\sigma)}{|\sigma|} \times \sum_{i=1}^{|\sigma|} \frac{|Enabled(CM_o, \sigma, i) \bigcap Enabled(CM_m, \sigma, i)|}{|Enabled(CM_m, \sigma, i)|} \right)}{\sum_{\sigma \in L} L(\sigma)}$$

[14] For both definitions 25 and 26, whenever the denominator "$|Enabled(CM, \sigma, i)|$" is equal to 0, the whole division is equal to 0. For simplicity reasons, we have omitted this condition from the formulae.

where

- *Enabled*(CM, σ, i) *gives the labels of the enabled activities at the causal matrix* CM *just before the parsing of the element at position* i *in the trace* σ*. During the parsing a continuous semantics is used (see Section 4.2.1).*

Definition 26 (Behavioral Recall - B_R). *Let* L *be an event log. Let* CM_o *and* CM_m *be the respective causal matrices for the original (or base) model and for the mined one. Let the ranges of the labelling functions of the original and mined models be the same, i.e.,* $rng(Label_o) = rng(Label_m)$*. Then the behavioral recall* $B_R : \mathcal{L} \times \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ *is a function defined as:*

$$B_R(L, CM_o, CM_m) =$$

$$\frac{\sum_{\sigma \in L} \left(\frac{L(\sigma)}{|\sigma|} \times \sum_{i=1}^{|\sigma|} \frac{|Enabled(CM_o, \sigma, i) \bigcap Enabled(CM_m, \sigma, i)|}{|Enabled(CM_o, \sigma, i)|} \right)}{\sum_{\sigma \in L} L(\sigma)}$$

Reasoning About the Quality of the Mined Models

When evaluating the quality of a data mining approach (genetic or not), it is common to check if the approach tends to find over-general or over-specific solutions. A typical extreme example of an over-general solution is the net that can parse any trace that can be formed from the tasks in a log. This solution has a self-loop for every task in the log (cf. Figure 4.8(a)). The over-specific solution is the one that has a branch for every *unique* trace in the log. Figure 4.8 illustrates both an over-general and an over-specific solution for the log in Table 4.1.

The over-general solution *does* belong to the search space considered in this chapter. However, this kind of solution can be easily detected by the metrics we have defined so far. Note that, for a given original model CM_o, a log L generated by simulating CM_o, and the mined over-general model CM_m, it always holds that: (i) the over-general model is complete (i.e., $PF_{complete}(L, CM_m) = 1$); (ii) while parsing the traces, all the activities that are enabled in the original model are also enabled in the over-general model (i.e., $B_R(L, CM_o, CM_m) = 1$); and (iii) while parsing the traces, all the activities of the over-general model are always enabled, i.e., the formula of the behavioral precision (see Definition 25) can be simplified to the formula

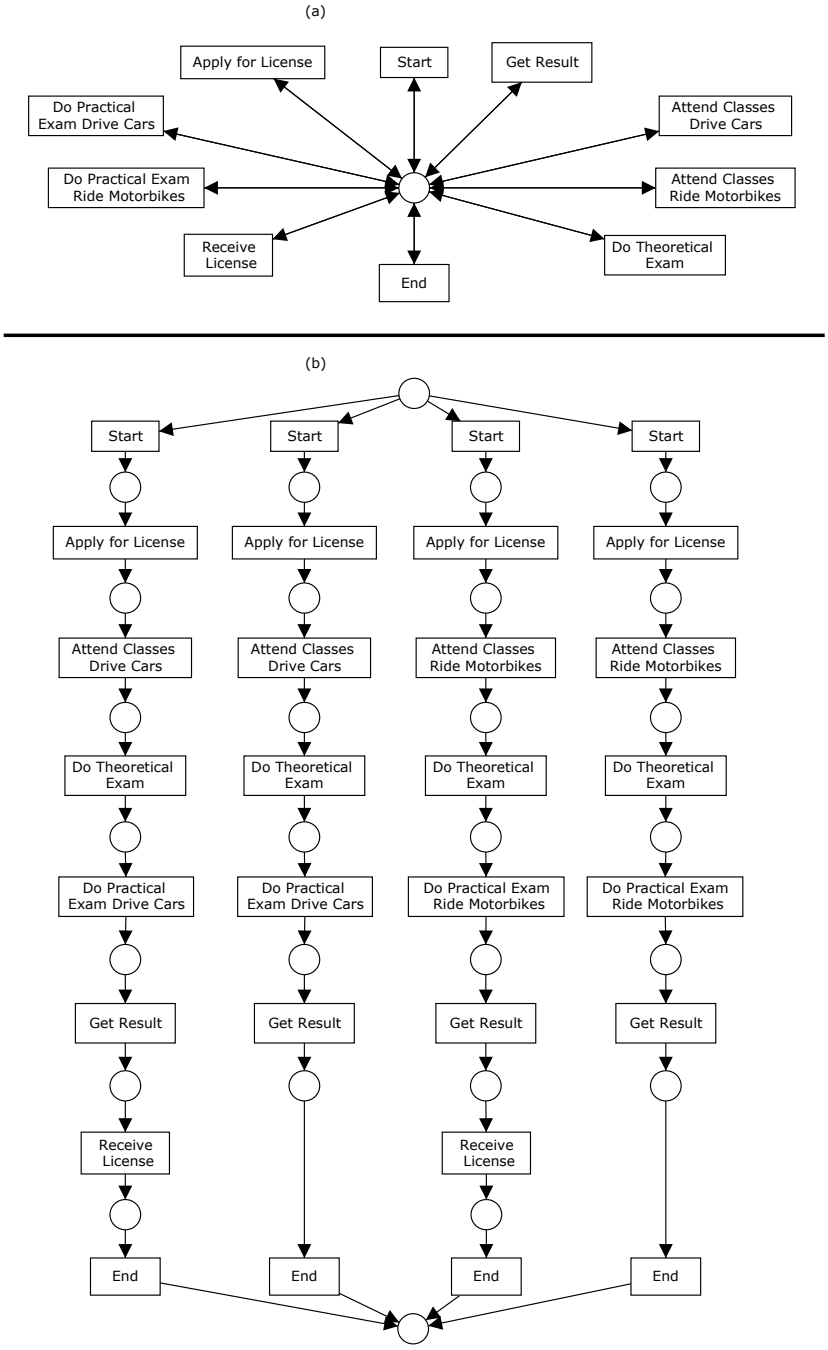


Figure 4.8: Example of nets that are (a) over-general and (b) over-specific for the log in the Table 4.1.

in Equation 4.1[15]. These three remarks helps us in detecting over-general mined models during the experiments analysis.

$$B_P(L, CM_o, CM_m) = \frac{\sum_{\sigma \in L} \left(\frac{L(\sigma)}{|\sigma|} \times \sum_{i=1}^{|\sigma|} \frac{|Enabled(CM_o, \sigma, i)|}{|A_m|} \right)}{\sum_{\sigma \in L} L(\sigma)} \tag{4.1}$$

Contrary to the over-general solution, the over-specific one *does not* belong to our search space because our internal representation (the causal matrix) does not support duplicate tasks. Note that the function *Label* is required to be injective (see Definition 18). However, because our fitness only looks for the complete and precise behavior (not the minimal representation, like the works on *Minimal Description Length (MDL)* [45]), it is still important to check how similar the structures of the mined model and the original one are. Differences in the structure may point out another good solution or an overly complex solution. For instance, have a look at the model in Figure 4.9. This net is complete and precise from a behavioral point of view, but it contains extra unnecessary places. Note that the places "p12" and "p13" could be removed from the net without changing its behavior. In other words, "p12" and "p13" are implicit places [17]. Actually, because the places do not affect the net behavior, all the nets in figures 4.1, 4.6 and 4.9 have the same fitness. However, a metric that checks the structure of a net would, for instance, point out that the net in Figure 4.9 is a "superstructure" of the net in Figure 4.1, and has many elements in common with the net in Figure 4.6. So, even when we know that the over-specific solution is out of the search space defined in this chapter, it is interesting to get a feeling about the structure of the mined models. That is why we developed two metrics to assess how much the mined and original model have in common *from a structural point of view*.

The two metrics are the *structural precision* (S_P) and the *structural recall* (S_R). Both metrics are based on the *causality relations* of the mined and original models, and were adapted from the precision and recall metrics presented in [64]. The S_P and S_R metrics are respectively formalized in definitions 27 and 28. These metrics basically work by checking how many causality relations the mined and the original model have in common. The more causality relations the two models have in common, the more similar

[15]Recall that we do not allow for duplicate tasks here, i.e., the function *Label* is required to be injective (see Definition 18).

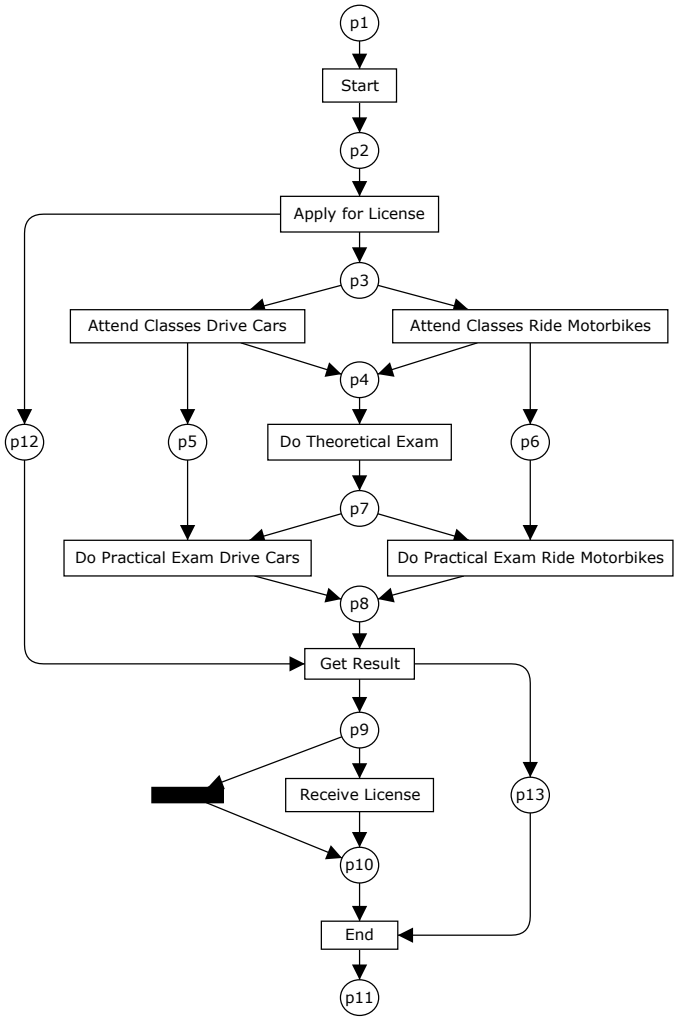


Figure 4.9: Example of a net that that is behavioral precise and complete w.r.t. the log in the Table 4.1, but that contains extra unnecessary (implicit) places (p12 and p13).

their structures are. The structural precision assesses how many causality relations the mined model has that are not in the original model. The structural recall works the other way around. Note that the structural similarity performed by these metrics does not consider the semantics of the split/join points. We have done so because the causality relations are the core of our genetic material (see Subsection 4.3). The semantics of the split/join tasks can only be correctly captured if the right dependencies (or causality relations) between the tasks in the log are also in place.

Definition 27 (Structural Precision - S_P). [16] *Let CM_o and CM_m be the respective causal matrices for the original and the mined models. Let the ranges of the labelling functions of the original and mined models be the same, i.e., $rng(Label_o) = rng(Label_m)$. The structural precision $S_P : \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ is a function defined as:*

$$S_P(CM_o, CM_m) = \frac{|mapToLabels(C_o, Label_o) \cap mapToLabels(C_m, Label_m)|}{|mapToLabels(C_m, Label_m)|}$$

where:

- *$mapToLabels(C, Label)$ is a function that applies the labelling function $Label$ to every element of a tuple in the causality relation C. For instance, if $C = \{(a_1, a_2), (a_2, a_3)\}$ and $Label = \{(a_1, a), (a_2, b), (a_3, c)\}$, then the function $mapToLabels(C, Label) = \{(a, b), (b, c)\}$.*

Definition 28 (Structural Recall - S_R). *Let CM_o and CM_m be the respective causal matrices for the original and the mined model. Let the ranges of the labelling functions of the original and mined models be the same, i.e., $rng(Label_o) = rng(Label_m)$. The structural recall $S_R : \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ is a function defined as:*

$$S_R(CM_o, CM_m) = \frac{|mapToLabels(C_o, Label_o) \cap mapToLabels(C_m, Label_m)|}{|mapToLabels(C_o, Label_o)|}$$

When the original and mined models have behavioral metrics B_R and B_P that are equal to 1, the S_R and S_P show how similar the structures of these models are. For instance, for the original model in Figure 4.1, the structural metrics would indicate that a mined model like the one in Figure 4.6 differs from the original one by the same amount of causality relations ($S_R = S_P$), and a mined model like the one in Figure 4.9 has extra causality relations ($S_R = 1$ and $S_P < 1$).

[16] For both definitions 27 and 28, whenever the denominator "$|mapToLabels(C, Label)|$" is equal to 0, the whole division is equal to 0. For simplicity reasons, we have omitted this condition from the formulae.

Recapitulation of the Analysis Metrics

This subsection has presented the five metrics that are used to analyse the results of the experiments: the partial fitness for the completeness requirement ($PF_{complete}$), the behavioral precision (B_P), the behavioral recall (B_R), the structural precision (S_P) and the structural recall (S_R). The $PF_{complete}$ quantifies how complete a mined model is. The B_P and B_R measures how precise the mined model is. The S_P and S_R express if the mined model has an overly complex structure or not. These metrics are complementary and should be considered together during the experiments analysis. For instance, for our experiments, the mined model is as complete and precise as the original model whenever the metrics $PF_{complete}$, B_P and B_R are equal to 1. More specifically, the mined model is exactly like the original model when all the five metrics are equal to 1. As a general rule, the closer the values of the five metrics are to 1, the better.

4.5.2 Setup

The genetic algorithm was tested over noise-free event logs from 25 different process models [17]. These models contain constructs like sequence, choice, parallelism, loops, non-free-choice and invisible tasks. From the 25 models, 6 were copied from the models in [50]. The other models were created by the authors. The models have between 6 and 42 tasks. Every event log was randomly generated and contained 300 process instances. To speed up the computational time of the genetic algorithm, the similar traces were grouped into a single one and a counter was associated to inform how often the trace occurs. The similarity criterion was the local context of the tasks in the trace. Traces with the same direct left and right neighbors for every task (i.e., traces with the same set of *follows* relation[18]) were grouped together. Besides, *to test how strong the use of the genetic operators and the heuristics influence the results*, we set up four scenarios while running the genetic algorithm: ("Scenario I") without heuristics to build the initial population and without genetic operators [19]; ("Scenario II") with heuristics, but without the genetic operators; ("Scenario III") without heuristics, but with genetic operators; and ("Scenario IV") with heuristics and genetic operators. For every log, 50 runs were executed for every scenario. Every run had a population size

[17] All models can be found in the Appendix B.

[18] See Definition 14 for details about the follows ($>_L$) relation.

[19] This scenario is a random generation of individuals. The aim of experimenting with this scenario is to assess if the use of genetic operators and/or heuristics is better than the pure random generation of individuals, given the same limited amount of computational time.

of 100 individuals, at most 1,000 generations, an elite of 2 individuals and a κ of 0.025 (see Definition 23). The experiments with heuristics used a power value of 1 while building the initial population (see Subsection 4.4.1). The experiments with the genetic operators have a respective crossover and mutation probabilities of 0.8 and 0.2 (see the respective subsections 4.3.1 and 4.3.2). All the experiments were run using the ProM framework. We implemented the genetic algorithm and the metrics described in this chapter as plug-ins for this framework (see Chapter 7 for details).

4.5.3 Results

The results in figures 4.10 to 4.17[20] indicate that the scenario for the hybrid genetic algorithm (Scenario IV) is superior to the other scenarios. More specifically, the results show that:

- Any approach (scenarios II to IV) is better than the pure random generation of individuals (Scenario I).
- Scenario II and IV mined more complete and precise models than the other scenarios.
- The hybrid genetic algorithm (Scenario IV) works the best. This approach combines the strong ability of the heuristics to correctly capture the *local* causality relations with the benefits of using the genetic operators (especially mutation) to introduce the non-local causality relations. For instance, consider the results for the nets `a6nfc`, `driversLicense` and `herbstFig6p36`. All these nets have non-local non-free-choice constructs. Note that, for these three nets, the results for Scenario II (cf. Figure 4.11) have a much lower behavioral precision than for Scenario IV (cf. Figure 4.13). This illustrates the importance of the genetic operators to insert the non-local causality relations.
- In general, the hybrid genetic algorithm (Scenario IV) mines more complete models *that are also precise* than the other scenarios (see Figure 4.17).
- Nets with short parallel branches (like `parallel5`, `a7` and `a5`) are more difficult to mine. This is due to the probabilistic nature of the genetic algorithm. Recall that the fitness measure always benefits the individuals that can parse the most frequent behavior in the log. So, in parallel situations, it is often the case that the algorithm goes for individuals that show the most frequent interleaving patterns in the log.

[20]The unmarked points in these figures correspond to experiments that were interrupted because they were taking more than 6 hours to finish one run.

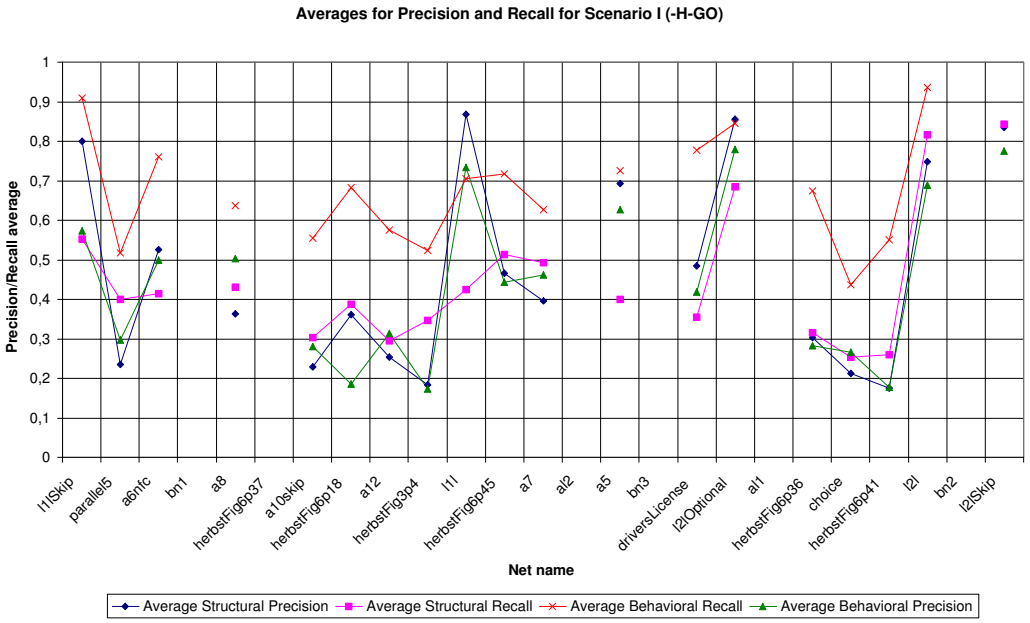


Figure 4.10: Average precision and recall values of the results for **Scenario I** (without heuristics to build the initial population and without using genetic operators to build the following populations).

- Although Scenario II led to better results than Scenario III, it is not fair to compare them. The reason is that Scenario III *starts from scratch*, in the sense that its initial population is randomly built, while Scenario II *is strongly helped by good heuristics* to detect local dependencies. In our experiments, Scenario III is used to show that (i) the use of the genetic operators improves the results (and this is indeed the case, since Scenario III gave better results than Scenario I), and (ii) the genetic operators help in mining non-local dependencies (again, note that the results for the nets with non-local non-free-choice constructs - `a6nfc` and `driversLicense` - are better in Scenario III than in Scenario II). Thus, in short, Scenario III shows that the GA was going on the right track, but it would need more iterations to reach or outperform the results of Scenario II for all nets.

4.6 Summary

This chapter explained how the genetic algorithm (GA) to tackle non-free-choice and invisible tasks works. The chapter has two main parts: the ex-

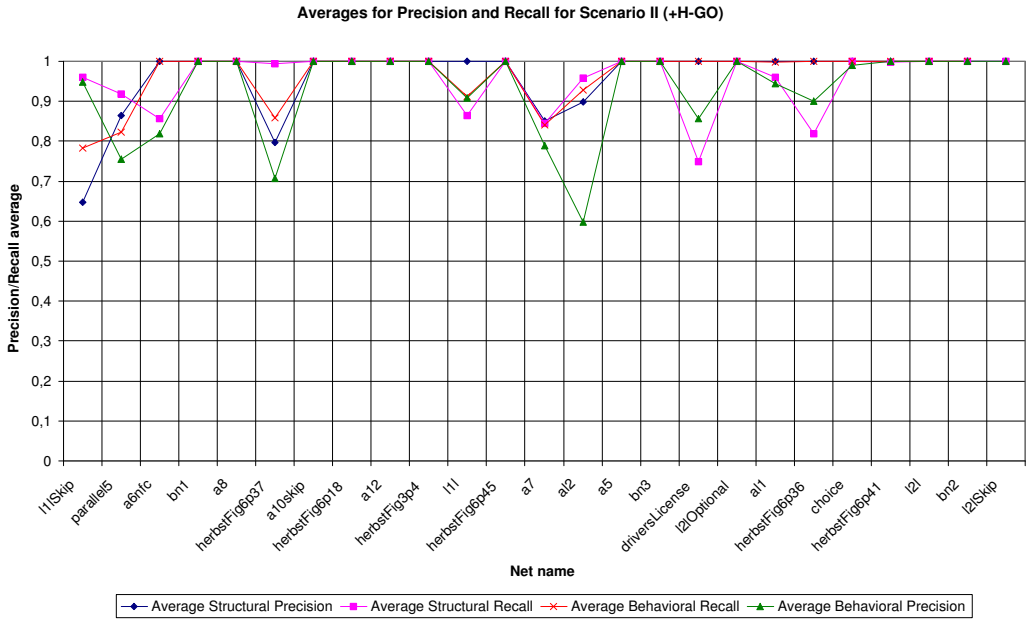


Figure 4.11: Average precision and recall values of the results for **Scenario II** (with heuristics to build the initial population, but without using genetic operators to build the following populations).

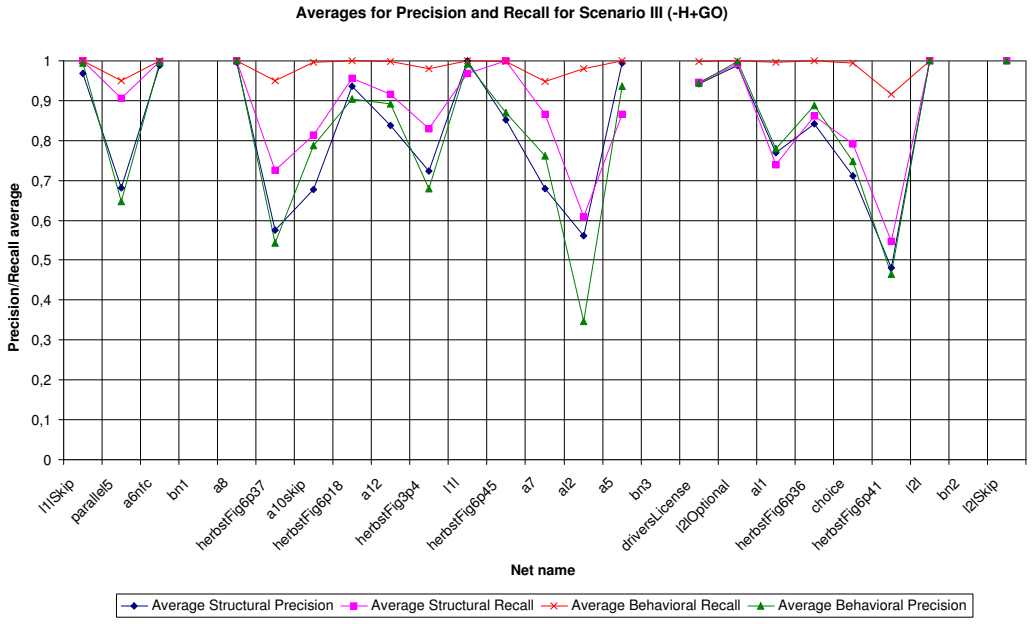


Figure 4.12: Average precision and recall values of the results for **Scenario III** (without heuristics to build the initial population, but using genetic operators to build the following populations).

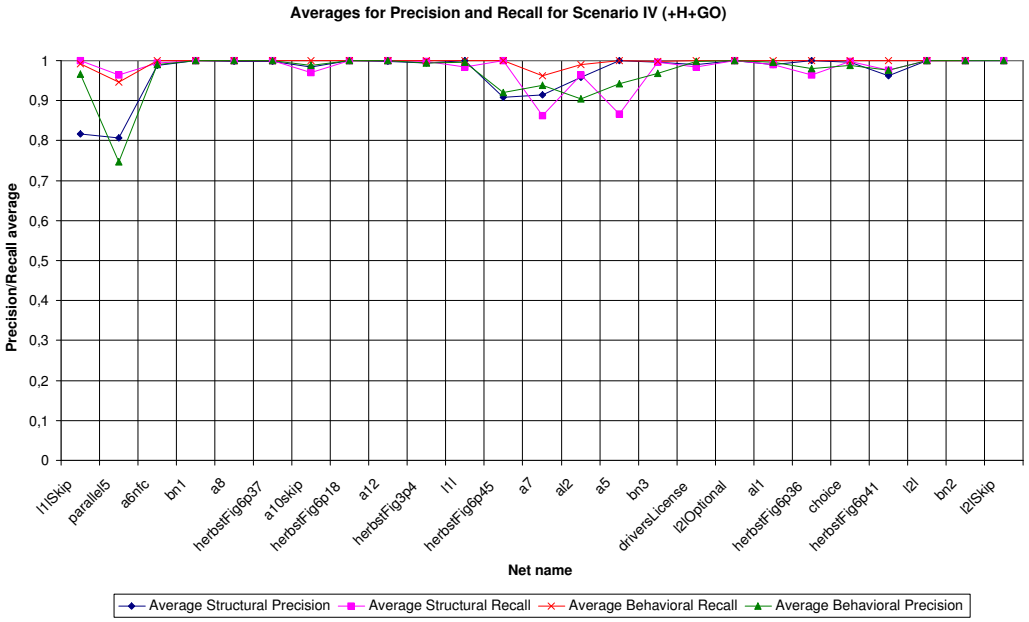


Figure 4.13: Average precision and recall values of the results for **Scenario IV** (with heuristics to build the initial population and using genetic operators to build the following populations).

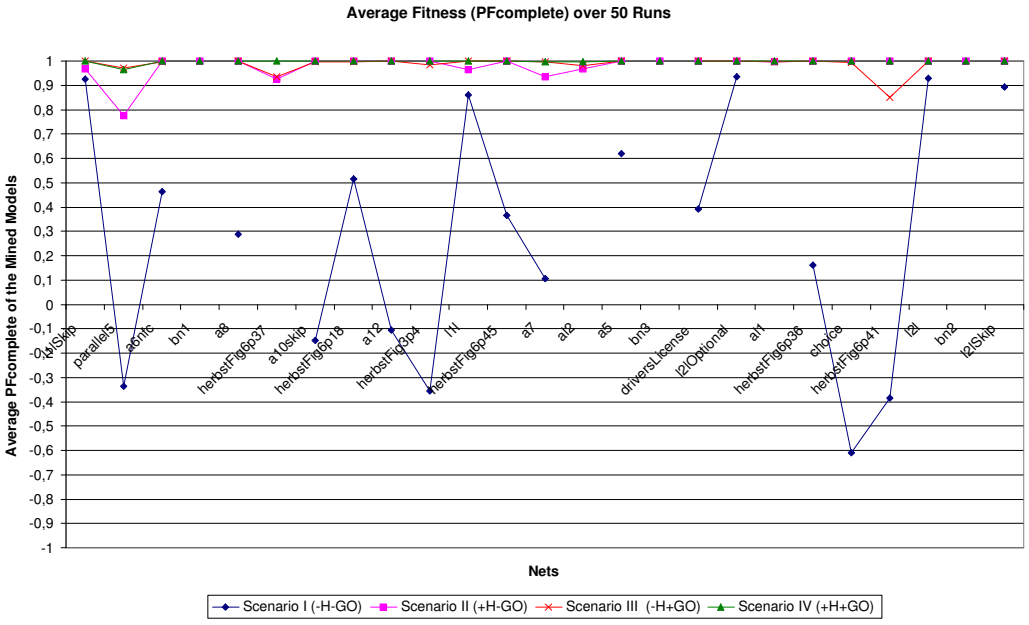


Figure 4.14: Average fitness ($PF_{complete}$) values of the mined models for 50 runs (each scenario depicted separately).

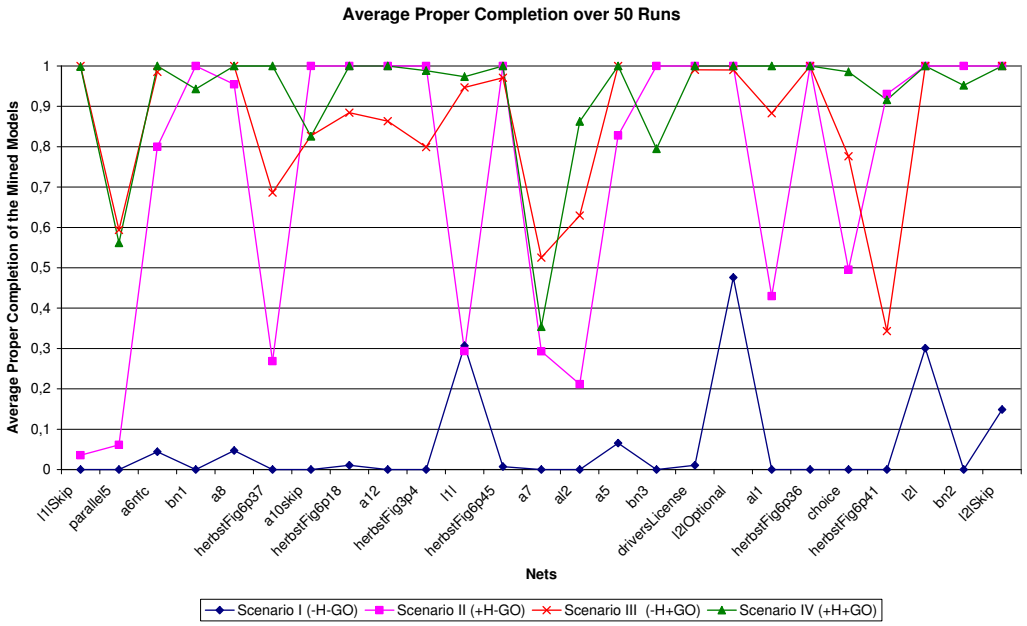


Figure 4.15: Average values of the proper completion fitness of the mined models for 50 runs. The proper completion fitness shows the proportion of the traces in the log that can be parsed without missing tokens or leaving tokens.

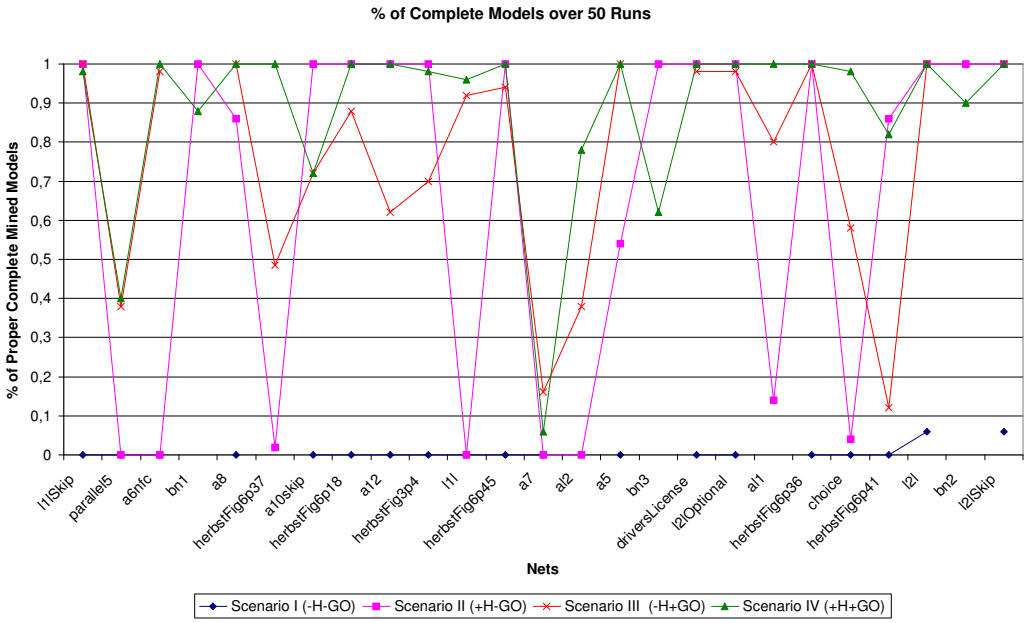


Figure 4.16: Percentage of the mined models that proper complete ($PF_{complete} = 1$) over 50 runs.

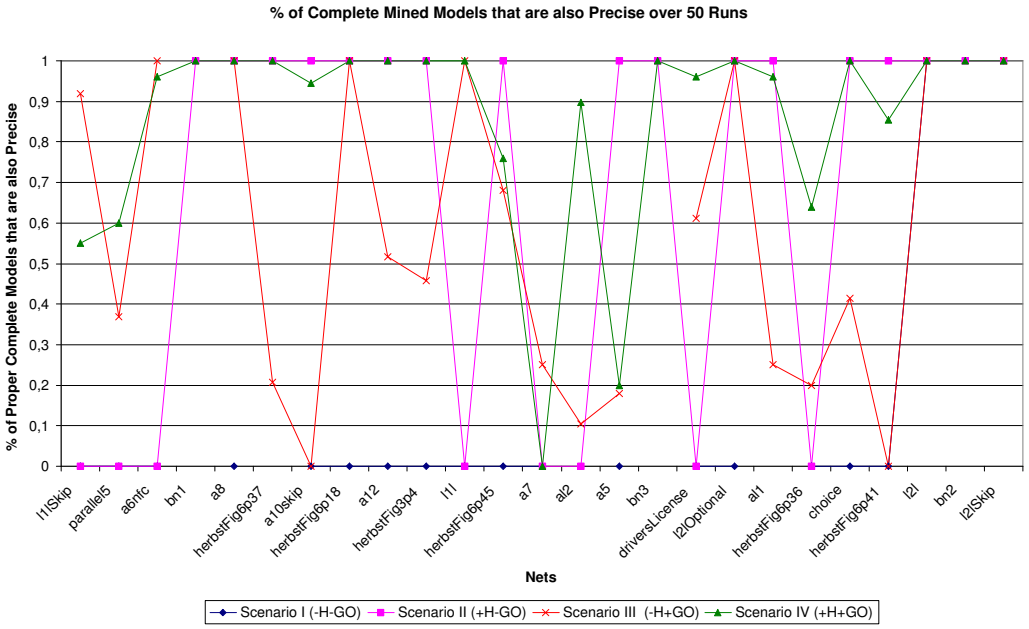


Figure 4.17: Percentage of the complete mined models ($PF_{complete} = 1$) that are also precise ($B_P = 1$) over 50 runs.

planation about the GA itself and the description of the developed metrics to evaluate the experiments.

The three main components of the GA are (i) the internal representation, (ii) the fitness function and (iii) the genetic operators. The internal representation defines the search space. So, the *causal matrix* supports the modelling of these kinds of constructs. The fitness function evaluates the quality of a candidate solution. In our case, this quality is measured by how *complete* and *precise* a candidate solution (or causal matrix or individual) is with respect to the event log given as input. A solution is complete when it can reproduce all the behavior observed in the log, it is precise when it does not describe much more behavior than the one that can be derived from the log. The crucial elements of the defined fitness are (i) the use of the continuous semantics when parsing the log, (ii) the weighed punishment for the amount of tokens that were not used during the parsing and the number of missing tokens, and (iii) the relative punishment for the number of possible behavior during the parsing of traces by a causal matrix. The genetic operators are the crossover and mutation. The underlying idea is that the causality relations are the core genetic material that the genetic operators manipulate.

For the evaluation, five metrics were defined: $PF_{complete}$, B_P, B_R, S_P and S_R. The metric $PF_{complete}$ is based on the log and the mined model. This metric assesses how complete the mined model is with respect to the log given as input. The metrics behavioral precision (B_P) and behavioral recall (B_R) are based on the log, the mined model and the original model. They quantify how precise the mined model is. The main idea is that, for a given log, the mined model should be at least as precise as the original model used to generate this log. The metrics structural precision (S_P) and structural recall (S_R) are based only on the mined model and the original model. These metrics measure how many causality relations the mined and original models have in common. All these five metrics are complementary when assessing the quality of a mined model. Moreover, the application of these metrics is not limited to the evaluation of genetic algorithms. They can be applied to another settings where models need to be compared.

The results show that the hybrid genetic algorithm indeed usually mines complete models that are also precise. However, the probabilistic nature of the GA seems to hinder the correct discovery of parallel branches (especially the ones with more than two branches).

The next chapter shows how we have extended the genetic algorithm presented here to tackle duplicate tasks as well. As it will be shown, the main challenge while defining a genetic algorithm to also tackle duplicate tasks is to avoid the mining of over-specific models.

Chapter 5

A Genetic Algorithm to Tackle Duplicate Tasks

This chapter explains the *Duplicates Genetic Algorithm* (DGA), an extension of the basic genetic algorithm (see Chapter 4) to mine process models with duplicate tasks as well. Duplicate tasks are used when a particular task is needed in *different contexts* in a process model. As an illustrative example, consider the model in Figure 5.1. This model shows the procedure for a one-day conference: the participants go to the conference by car or by train, they may give a talk, they join the social activities (guided tour and dinner) and, when the conference ends, they go back home. Furthermore, the model shows that (i) the participants use the same means of transportation (train or car) to come to the conference location and to go back home and (ii) the participants that come by car have to pay for the parking place. Note that the tasks "Travel by Car" and "Travel by Train" are duplicated in this model because, although they refer to a same task (being in a vehicle to go somewhere), they are used in different contexts in the process model. The top tasks "Travel by Train" and "Travel by Car" are in the context of "going to the conference". The bottom ones are in the context of "going home or leaving the conference". In other words, the *going somewhere* is different. This example illustrates that a genetic algorithm to mine process models with duplicate tasks should allow *two or more activities in an individual to be linked to a same task (or label) in the log.* Therefore, one trivial extension to the GA in Chapter 4 is to remove the constraint that the labelling function *Label* of a causal matrix (see Definition 18) is injective. If *Label* is not injective anymore, the internal representation (the causal matrix) naturally supports the modelling of processes with duplicate tasks. However, this simple extension has implications for the *size of the search space*, the *parsing* of traces and the *fitness measure*:

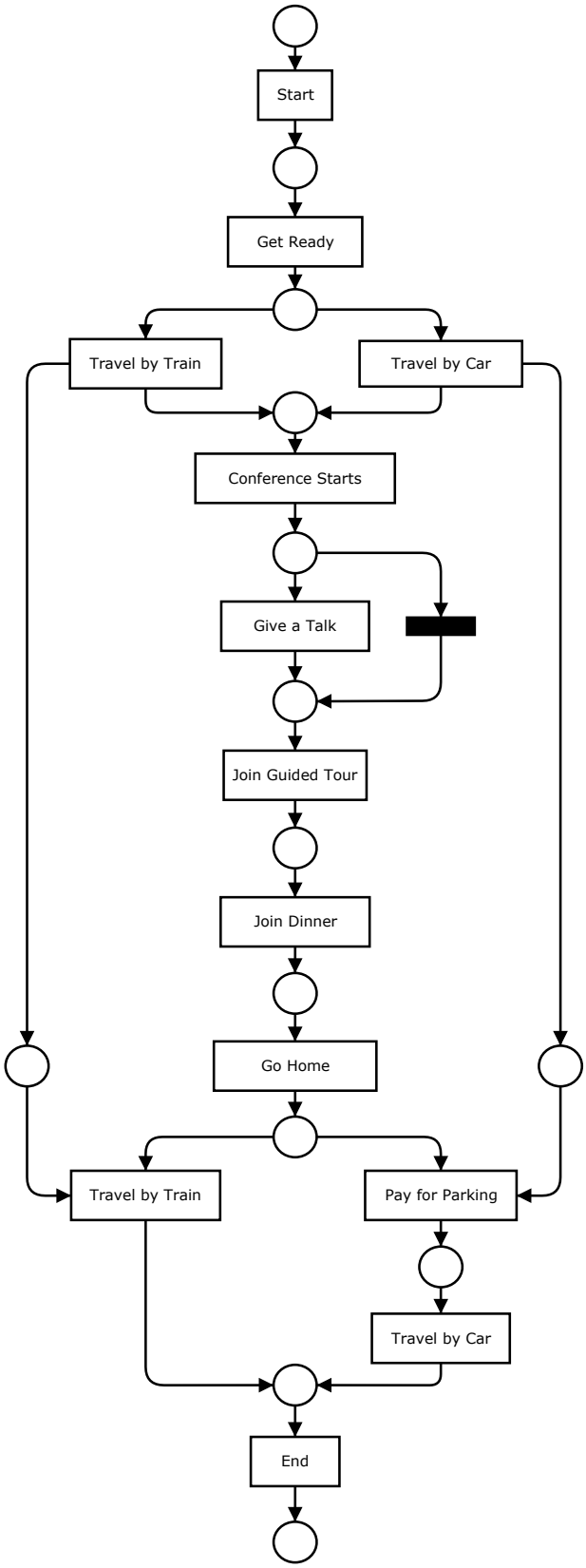


Figure 5.1: Example of a model containing duplicate tasks.

Identifier	Process instance
1	Start, Get Ready, Travel by Car, Conference Starts, Give a Talk, Join Guided Tour, Join Dinner, Go Home, Pay for Parking, Travel by Car, End.
2	Start, Get Ready, Travel by Train, Conference Starts, Give a Talk, Join Guided Tour, Join Dinner, Go Home, Travel by Train, End.
3	Start, Get Ready, Travel by Car, Conference Starts, Join Guided Tour, Join Dinner, Go Home, Pay for Parking, Travel by Car, End.
4	Start, Get Ready, Travel by Train, Conference Starts, Join Guided Tour, Join Dinner, Go Home, Travel by Train, End.

Table 5.1: Example of an event log (with four process instances) for the process model in Figure 5.1.

- The *search space* is defined by the internal representation. When duplicates are allowed, this search space increases considerably. Actually, the search space may even be infinite if no limit is set to the maximum amount of duplicates that every task in the log can have. Since our experience with the basic genetic algorithm (see Chapter 4) shows that running the GA is very time consuming, it is worth to set a *maximum amount of duplicates that a task in the log can have* before the DGA starts the search. This makes the search space finite and also speeds up the search. However, over-specific solutions may still be in the search space. For instance, consider the log in Table 5.1. If we set that there are as many duplicates of a task as the number of times this task appears in the log, the over-specific model portrayed in Figure 5.2 would be in the search space.
- The *parsing* is affected because it is not trivial to decide which duplicate to fire when multiple duplicates of a given task are enabled. Recall that the fitness measure of the basic GA is based on the replay of the log traces by the individuals. The main idea of replaying a trace is to fire the activities in the individual in the order in which their corresponding tasks appear in the trace. However, when duplicates are allowed, choosing among duplicates of a same task may become very costly because the correct choice may involve looking ahead more than one position in the individual. For instance, assume the situation in which the *process instance 1* (see Table 5.1) should be parsed by an

individual like the one in Figure 5.2. Assume that this individual has initially one token in place $p1$. In this situation, 4 duplicates of the task "Start" are enabled. When replaying *process instance 1* (cf. Table 5.1) for this individual, the parser has to choose which duplicate to fire. The correct duplicate to fire is the "Start" connected to the place $p4$ because its branch models the behavior of this process instance. One way to correctly decide which duplicate to fire is to *look ahead* in the individual structure and the log trace. In fact, this is the log replay semantics used in [67, 68] for conformance checking. However, this process becomes very expensive, from a computational time point of view, when it has to be done for every individual in the population.

- The *fitness* guides the search of any genetic algorithm. For the basic GA, the fitness guided the search towards individuals that are complete (can parse the behavior observed in the log) and precise (cannot parse much more than the behavior that can be derived from the log). However, when duplicates are allowed, the fitness also needs to incorporate some requirement to punish the individuals that have more duplicates than necessary. In other words, the fitness needs to promote the folding of duplicates when they are in excess. For instance, note that both individuals in figures 5.1 and 5.2 are complete and precise with respect to the log in Table 5.1[1], but the individual in Figure 5.2 has unnecessary duplicates that can be folded. This situation illustrates why the preciseness and completeness requirements, although necessary, are not sufficient anymore.

All these remarks led us to develop a genetic algorithm that aims at mining models in which the duplicates can be distinguished *based on their local context*. The local context of a duplicate is the set of input and output elements of the duplicate. Thus, the DGA presented in this chapter aims at the mining of *process models in which duplicates of a same task do not have input elements or output elements in common.*

The requirement that the duplicates can be distinguished based on their local context solves the implications we have mentioned for the search space, the parsing and the fitness measure. The *search space* can be easily made finite by using heuristics to set the maximum amount of duplicates per task. Since the duplicates are locally identifiable and they do not share input/output elements, the heuristics can use the *follows relation* (cf. the relation "$>_L$" in Definition 14) to determine the maximum amount of duplicates that a task can have. The *parsing* can be simplified from n-look-ahead to 1-look-ahead in the individual structure. Because duplicates do not have

[1]Note that the nets in Figure 5.1 and Figure 5.2 are trace equivalent.

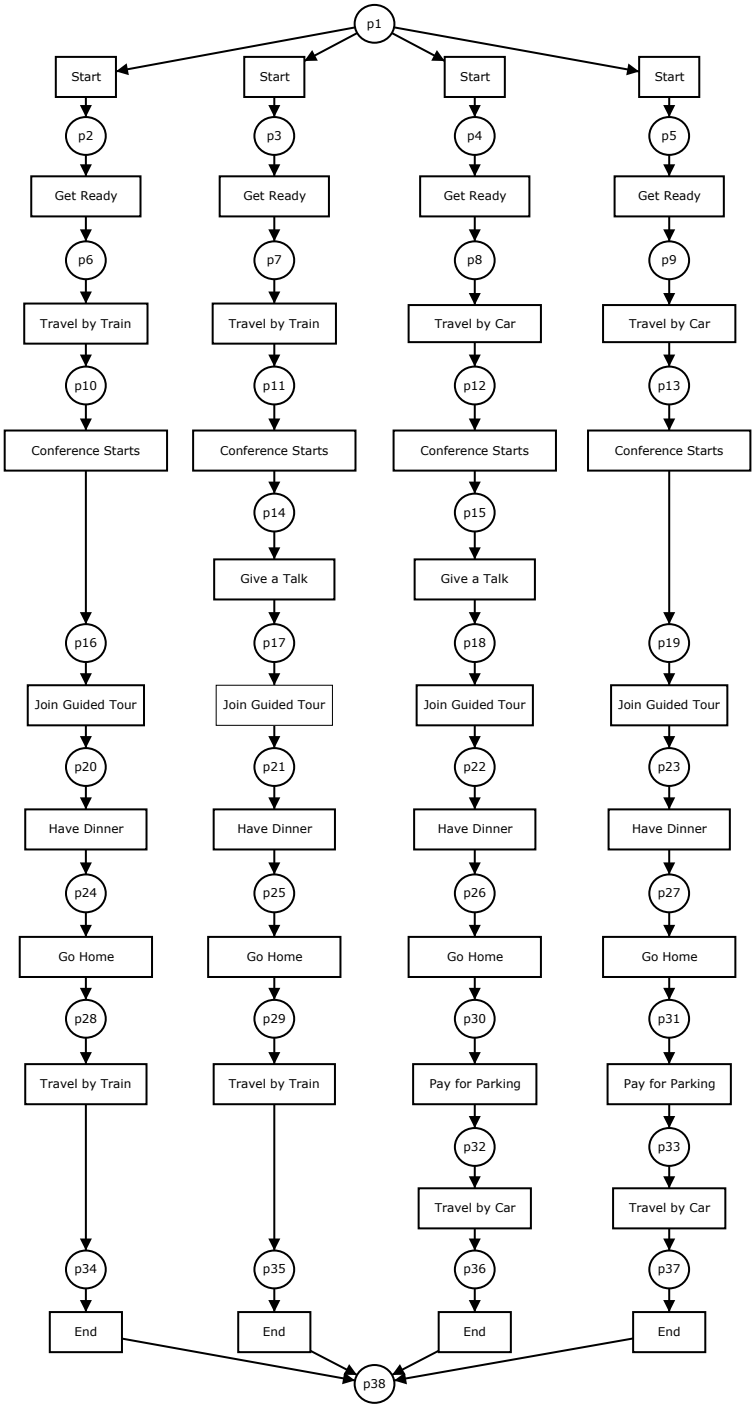


Figure 5.2: An over-specific mined model for the log in Table 5.1

common output elements, looking at their output elements should be sufficient to choose which duplicate to fire. The *fitness measure* can be easily extended with a punishment that is based on the structure of the individuals. In short, an individual is punished whenever duplicates of a same task have common input/output elements.

Finally, we point out that the use of *local* information to set the maximum amount of duplicates does not prevent the DGA of mining individuals with *non-local* dependencies between activities. For instance, the DGA illustrated in this chapter can mine the model in Figure 5.1. Note that this model has a non-local non-free-choice construct. The DGA can still mine non-local dependencies because, like for the basic GA, the genetic operators allow for the creation/exchange of non-local dependencies.

The remainder of this chapter explains the DGA in more detail. Section 5.1 presents the definition of the causal matrix to support duplicates. Basically, the function *Label* (see Definition 18) is not injective anymore. Section 5.2 explains the new "folding" requirement that needs to be added to the fitness measure. This section also describes in more detail how the continuous parsing semantics works in the presence of duplicates. Section 5.3 describes the small changes to the genetic operators, so that they work at the duplicates level, instead of at the activity level. Section 5.4 describes the DGA's steps. The main differences are related to the way the initial population is built because heuristics are also used to set the maximum amount of duplicates per task. Section 5.5 contains the experimental results. This section also shows how the analysis metrics are adapted to work for models with duplicates, and introduces new analysis metrics that check if the mined models have the correct amount of duplicates per task. Section 5.6 provides a short summary of this chapter.

5.1 Internal Representation and Semantics

The DGA needs an internal representation that supports duplicates. An internal representation supports duplicates when more than one activity of an individual can have the same label. Thus, we can easily extend the internal representation of the basic genetic algorithm (see Definition 18) by removing the constraint that the function *Label* should be injective. The *extended causal matrix* is, then, formally defined as:

Definition 29 (Extended Causal Matrix). *Let LS be a set of labels. An Extended Causal Matrix is a tuple $ECM = (A, C, I, O, Label)$, where*

- *A is a finite set of activities,*

- $C \subseteq A \times A$ *is the causality relation,*
- $I : A \rightarrow \mathcal{P}(\mathcal{P}(A))$ *is the input condition function,*[2]
- $O : A \rightarrow \mathcal{P}(\mathcal{P}(A))$ *is the output condition function,*
- $Label : A \rightarrow LS$ *is a labeling function that maps each activity in* A *to a label in* LS*,*

such that

- $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$,[3]
- $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$,
- $C \cup \{(a_o, a_i) \in A \times A \mid a_o \overset{C}{\bullet} = \emptyset \wedge \overset{C}{\bullet} a_i = \emptyset\}$ *is a strongly connected graph.*

Note that any causal matrix (CM) (see Definition 18) is also an extended causal matrix (ECM). Furthermore, ECMs have the same semantics of CMs (cf. Section 4.1) and the mappings in Appendix A (from Petri nets to Causal Matrix, and vice-versa) still hold. This is the case because both the semantics (explained in Section 4.1) and the mappings in Appendix A are independent of the activity labels. In the remainder of this document, we will use the term causal matrix to also refer to an extended causal matrix.

5.2 Fitness Measurement

The fitness measure of the DGA is based on two extensions to the basic GA. The first extension is the *addition of the folding requirement* to the fitness measure itself. The second extension is the *adaptation of the continuous semantics parser* of the basic GA to handle duplicates during the log replay performed by the fitness. In the following we explain these two extensions.

Extensions to the Fitness Measure

The *folding requirement* was added to the fitness of the DGA to avoid over-specific solutions. Recall that the fitness fitness F (see Definition 23) of the GA captures the *completeness* and *preciseness* requirements. In fact, the completeness and preciseness requirements are still necessary because (i) the mined model should represent the behavior in the log and (ii) over-general solutions are also in the search space. However, these two requirements are not sufficient anymore because they do not allow the fitness to distinguish the individuals that are complete and precise and *do not over-specialize* from the ones that *do over-specialize*. For instance, consider the log in Table 5.1, and the complete and precise mined models in figures 5.1 and 5.2. Note that

[2] $\mathcal{P}(A)$ denotes the powerset of some set A.

[3] $\bigcup I(a_2)$ is the union of the sets in set $I(a_2)$.

both mined models have the same fitness F value because they are complete and precise. However, the model in Figure 5.2 is over-specific. This example shows that, when over-specific solutions are in the search space, the fitness measure needs an extra metric to punish for over-specializations. In other words, the fitness measure should be able to benefit individuals like the one in Figure 5.1 over individuals like the one in Figure 5.2. That is why we have added the "folding" requirement to the fitness measure used by the DGA. But, how to detect that folding is necessary? The answer follows from the discussion about the kind of models the DGA can mine.

Recall that in the introduction of this chapter we explained that the DGA targets the class of models in which duplicates do not have input elements or output elements in common. Thus, the "folding" requirement is based on this constraint. The main idea is that individuals are punished whenever they do not satisfy this constraint. Moreover, the more the individuals have duplicates that violate this constraint, the higher the punishment. The "folding" requirement is formalized as follows:

Definition 30 (Partial Fitness - $PF_{folding}$). [4] *Let CM be a causal matrix. Let $CM[]$ be a bag of causal matrices that contains CM. Then the partial fitness $PF_{folding} : \mathcal{CM} \times \mathcal{CM}[] \rightarrow [0, 1]$ is a function defined as*

$$PF_{folding}(CM, CM[]) = \frac{DuplicatesSharingElements(CM)}{max(DuplicatesSharingElements(CM[]))}$$

where

- *$DuplicatesSharingElements : \mathcal{CM} \rightarrow \mathbb{N}$ is a function that calculates the number of different tuples in the causality function C of the causal matrix CM, that map to the same labels. Formally, $DuplicatesSharingElements(CM) = |\{(a_1, a_2) \in C | \exists (a'_1, a'_2) \in (C \backslash \{(a_1, a_2)\})[Label(a_1) = Label(a'_1) \wedge Label(a_2) = Label(a'_2)]\}|$.*
- *$DuplicatesSharingElements : \mathcal{CM}[] \rightarrow \mathbb{N}$ is a function that returns the set of values $DuplicatesSharingElements(CM)$ for every individual in the population. Formally, $DuplicatesSharingElements(CM[]) = \{DuplicatesSharingElements(x) | x \in CM[]\}$.*
- *$max : \mathcal{P}(\mathbb{N}) \rightarrow \mathbb{N}$ is a function that returns the maximum value of a set.*

The extended fitness - called F_{DGA} - adds to the fitness F (see Definition 23) of the basic genetic algorithm the "folding" requirement. The F_{DGA} is defined as follows:

[4] For Definition 30, whenever the denominator $max(DuplicatesSharingElements(CM[]))$ is equal to 0, the whole division is equal to 0.

Definition 31 (DGA Fitness - F_{DGA}). *Let L be an event log. Let CM be a causal matrix. Let $CM[]$ be a bag of causal matrices that contains CM. Let F be the fitness function given in Definition 23. Let $PF_{folding}$ be the partial fitness function introduced in Definition 30. Let γ be a real number greater than 0 and smaller or equal to 1 (i.e., $\gamma \in (0, 1]$) . Then the fitness $F_{DGA} : \mathcal{L} \times \mathcal{CM} \times \mathcal{CM}[] \rightarrow (-\infty, 1)$ is a function defined as*

$$F_{DGA}(L, CM, CM[]) = F(L, CM, CM[]) - \gamma * PF_{folding}(CM, CM[])$$

The fitness F_{DGA} weighs (by γ) the punishment for individuals that have duplicates with common input/output elements. Thus, if a set of individuals are complete and precise (they have all $F = 1$), the ones that have fewer duplicates sharing common input/output elements will have a higher fitness F_{DGA} value. Furthermore, once the duplicates satisfy the constraint, no further punishment is introduced. This means that the fitness F_{DGA} does not benefit certain constructs over others, given that the individual fits the class of targeted models. For instance, whenever a situation can be modelled in terms of non-free-choice or duplicates, the fitness will not benefit an individual over another. To illustrate this situation, consider the individual in Figure 5.3. This individual replaces the non-free-choice of the individual in Figure 4.1 by the duplication of the task "Do Theoretical Exam". Both individuals have the same fitness value for F_{DGA}. This is a situation in which the duplicates do not relate to over-specialization. However, the folding requirement does not prevent all kinds of over-specialization. Namely, over-specific individuals that do not violate the folding constraint are not punished. For instance, the fitness F_{DGA} does not distinguish between the nets in Figure 5.4 because the folding punishment is the same for both models. Note that, in the net Figure 5.4(b), neither the duplicates of task A nor the duplicates of task B have common input/output elements. However, although the folding requirement does not prevent all cases of specialization, this requirement is simple and works for many situations.

Extensions to the Continuous Semantics Parser

Like the basic GA, the DGA also uses continuous semantics parsing while calculating an individual's fitness. Recall that a continuous semantics means that the parsing process proceeds even when the activity to be parsed is not enabled. In other words, the activity that should be parsed always fires (or executes) (see Subsection 4.2). However, choosing which activity to fire becomes a less trivial task when duplicates are possible. More specifically, the parsing process needs to resolve the scenarios in which (i) more than one duplicate is enabled in the individual, or (ii) none of the duplicates are enabled. The latter scenario - *all duplicates are disabled* - is solved by

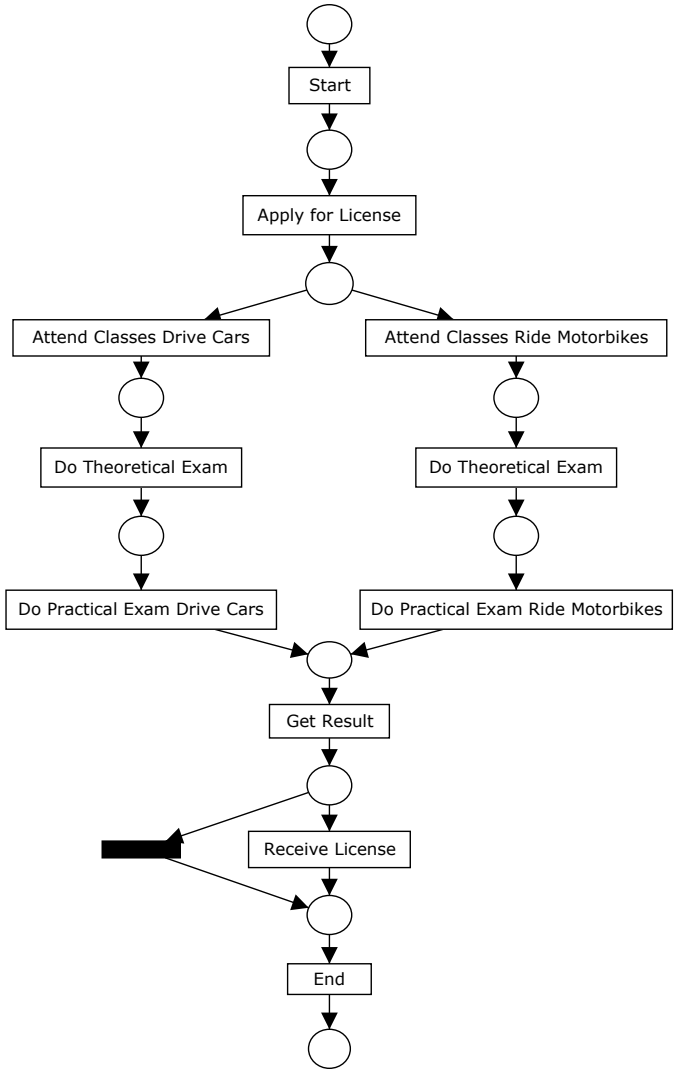


Figure 5.3: Net with the same behavior of the net in Figure 4.1, but that uses duplicates instead of the non-free-choice construct.

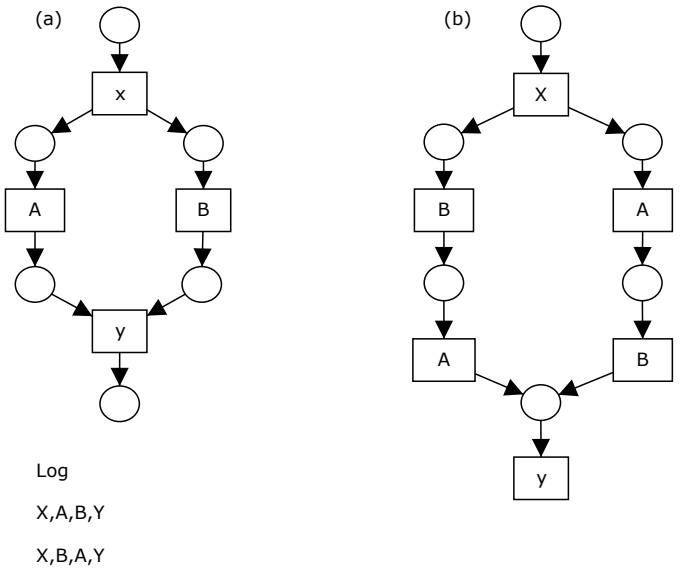


Figure 5.4: Example of a situation in which the fitness F_{DGA} will not punish the net (b) for its extra duplicates.

randomly firing one of the duplicates. The former scenario - *multiple enabled duplicates* - is solved based on the output elements of the enabled duplicates. The enabled duplicate that has an *output* element that occurs *first* in the trace, from the current position until the end of the trace, is the one to be fired. If multiple enabled duplicates satisfy this condition, one of the duplicates is randomly fired anyway. When this condition does not hold for any of the enabled duplicates, one of them is randomly fired. Note that this firing policy generates some unpredictability for the parsing process when the individuals have duplicates with output elements in common. However, due to the folding requirement of the fitness F_{DGA}, this non-determinism should decrease as the population evolves and the individuals tend to have fewer duplicates that share input/output elements.

5.3 Genetic Operators

Before we explain the extension for the genetic operators, let us step back a bit and explain the differences between the genotypes and the phenotypes of the individuals in the population. The *genotype* is the representation that the genetic algorithm deals with during the search. The *phenotype* is how the genetic algorithm presents its results [38]. Understanding this difference is important to comprehend how we extended the genetic operators.

When choosing how to represent the individuals, we had two possibilities. The *first possibility* would be to have a population in which the individuals might have a different number of activities (different sets A). In this case, the number of activities of every individual might shrink or grow as the population evolves, and all the activities should be involved in some causality relation C. The *second possibility* would be to have a population in which all the individuals have the same amount of activities (same set A), but some of these activities could be disconnected from the net. We selected the second possibility because it is faster to manipulate. The "trick" here is that the disconnected activities are not shown in the phenotypes of the individuals. The gain is that the genetic operators can be easily adapted and do not have to deal with keeping track of the shrinking and growing of the individuals.

Actually, only the crossover operator needs to be adapted in order to support duplicates. Recall that the crossover operator of the basic GA works by selecting an *activity* at both parents and randomly exchanging some parts of these activities' input/output sets. Now that duplicates are possible, the crossover operator works at the *label* level. In other words, the crossover point is now a *label* that belongs to the range of the function *Label* of every individual. Based on this label, one of the duplicates linked to this label is selected in one parent, and another duplicate (perhaps the same), that is also associated to the same label, is selected in the other parent. The rest of the procedure is the same as for the crossover of the basic GA (see Section 4.3.1 for more details). The change from activity level to label level makes the crossover promote the exchange of genetic material (causality relations) between different duplicates of individuals in the population. The mutation operator remains the same because its work is independent of the activity being a duplicate or not.

5.4 Algorithm

The DGA basically works like the GA (see Section 4.4), and also has the five main steps depicted in Figure 4.4. The differences are that the DGA (i) in Step II, builds the initial population using heuristics to set the maximum amount of duplicates; (ii) in Step III, uses the fitness F_{DGA} (see Definition 31) instead of the fitness F (see Definition 23) and the continuous semantics parsing that handles duplicates (see Section 5.2); and (iii) in Step IV, also removes the dangling duplicates while cleaning the individuals to be returned. The following subsection has more details about how the initial population is built.

5.4.1 Initial Population

The most important issue while building the initial population is to set the amount of duplicates per task that each individual will have. This is important because it defines if the desired solutions are in the search space or not. For instance, to mine an individual like the one in Figure 5.1 for the log in Table 5.1, the internal representation should support at least two duplicates of the task "Travel by Train" and two duplicates of the task "Travel by Car". If this is not the case, individuals like the one in Figure 5.1 could never be found because they will not belong to the search space. In terms of the DGA's internal representation (see Definition 29), setting the amount of duplicates per task implies determining the set A of activities and the labelling function *Label*. The range of the function *Label* is the set of tasks in the log (every task is a label). The DGA uses heuristics based on the *follows relation* (cf. $>_L$ in Definition 14) to determine the set of activities A.

Since the DGA targets models in which duplicates do not have input or output elements in common, the follows relation ($>_L$ in Definition 14) derived from the log can be safely used to set the maximum amount of duplicates per task. The main idea is that, for a given task t in the log, the individuals have as many duplicates as the *minimum of the number of tasks that directly precede t in any trace in the log and the number of tasks that directly follow t* [5]. This can be formalized as follows: $\forall_{l \in rng(Label)} |Label^{-1}(l)| = max(min(|\{l' \in LS | l >_L l'\}|, |\{l' \in LS | l' >_L l\}|), 1)$. The minimum is sufficient because the input/output elements of any two duplicates of a same task should be disjoint. For instance, if we compute the follows relation for the log in Table 5.1, we see that the task "Travel by Train" is directly preceded by the tasks "Get Ready" and "Go Home", and is directly followed by the tasks "Conference Starts" and "End". Therefore, for this log, all individuals should have *two* duplicates for task "Travel by Train". If a similar reasoning is done for the other tasks in the log, the result is that every individual in the population will have two duplicates of the tasks "Travel by Car", "Travel by Train" and "Conference Starts". The other tasks will not have duplicates. This means that for this log, the search space has the size of all possible process models that are compliant with the causal matrix definition for this A and this *Label*. However, as the reader might already have noticed, the "ideal" individual for the log in Table 5.1 does not duplicate the task "Conference Starts". The reasoning we use here is that *we set an upper bound for the amount of duplicates per task for all individuals in the population, but we*

[5] When a task is not directly preceded or directly followed by any other task in the log, we set the minimum to 1. In other words, there is at least one activity in A that is linked to a task in the log.

allow for dangling activities. Thus, there is an upper limit, but activities may be disconnected from the rest of the net. So, given the A and *Label* for the previous example, the genotype of an individual like the one in Figure 5.1 would have one of the duplicates for "Conference Starts" disconnected from the rest of the net. Whenever an individual has disconnected activities, we do not show these disconnected activities in the individual's phenotype and, consequently, in the mapped Petri net for this individual.

Once the maximum amount of duplicates is set, the dependency measure is used to create the causality relations C for every individual. Again, we reinforce that all individuals have a genotype with the same A and *Label*, but their C and I/O functions may differ. We use the dependency measure D (see Definition 24) to build the causality relation C of every individual. The dependency measure indicates how strongly a task t in the log is a cause for another task t'. Note that this measure is defined over the log. Therefore, in the context of the DGA, the dependency measure indicates *how strongly an activity "a" linked to a label "t" is a cause to another activity "a′" linked to a label "t′"*. For this reason, the pseudo-code to set the causality relations for the individuals of an initial population (see Subsection 4.4.1, on page 69) needed to be adapted as follows:

Pseudo-code:
input: An event-log L, the set A, the function *Label*, an odd power value p, a dependency function D.
output: A causality relation C.

1. $T \longleftarrow$ The range of the function *Label*.
2. $C \longleftarrow \emptyset$.
3. FOR every tuple (t_1, t_2) in $T \times T$ do:
   (a) Randomly select a number r between 0 (inclusive) and 1.0 (exclusive).
   (b) IF $r < D(t_1, t_2, L)^p$ then:
      i. $c \in \{(a_1, a_2) \in A \times A \mid \mathit{Label}(a_1) = t_1 \wedge \mathit{Label}(a_2) = t_2\}$.
      ii. $C \longleftarrow C \cup \{c\}$.
4. Return the causality relation C.

Note that the Step 3b of the pseudo-code has been modified to take the duplicates into account. Besides, there is at most one causality relation between any two pairs of duplicates that map to the same labels (cf. Step 3(b)i). This is the case because we assume that no duplicates have input/output elements in common. When the causality relations of an individual are determined,

the condition functions I and O are randomly built as described for the basic GA (cf. Subsection 4.4.1, on page 68).

5.5 Experiments and Results

This section explains how we conducted the experiments and analyzed the quality of the models that the DGA mined. To conduct the experiments we had (i) to implement the DGA and (ii) collect a representative set of event logs. The *Duplicates Genetic Algorithm* described in this chapter *is implemented as the "Duplicate Tasks GA plug-in"* in the ProM framework (see Figure 5.5 for a screenshot). Chapter 7 provides more details about ProM and the DGA plug-in. The *logs* used in our experiments are *synthetic*. In brief, we constructed models (or copied them from related work) and simulated them to create synthetic event logs[6]. We then ran the DGA over these sets of logs. Once the algorithm finished the mining process, we analyzed the results. A DGA run is successful whenever the mined model is *complete* (it can parse all the traces in the log), *precise* (it does not allow for more behavior than the one that can be derived from the log) and *folded* (it does not contain unnecessary duplicates). The completeness and preciseness requirements can be analyzed by using the metrics we have defined for the basic genetic algorithm (see Subsection 4.5.1). However, the structural precision (S_P) and recall (S_R) need to be extended to work with bags. The reason is that the labelling function *Label* is not injective anymore, so there can be more than one tuple with the same pair of labels. To check for the folding requirement, we have defined two new analysis metrics: *duplicates precision* and *duplicates recall*. These metrics basically check if the original and mined models have the same amount of duplicates per task. The remainder of this section is divided into three parts: (i) Subsection 5.5.1 presents the extensions to the structural precision/recall metrics as well as the new metrics that check for the folding requirement, (ii) Subsection 5.5.2 describes the experiments setup, and (iii) Subsection 5.5.3 shows the results.

5.5.1 Evaluation

The analysis metrics *structural precision* (S_P) and *Structural Recall* (S_R) measure how many connections (causality relations) two individuals have in common (see Subsection 4.5.1, on page 79). Two individuals have a common causality relation when the activities involved in this relation - say $(a_1, a_2) \in C$ in one individual and $(a'_1, a'_2) \in C'$ in the other individual

[6]Section 8.1 provides more details about this.

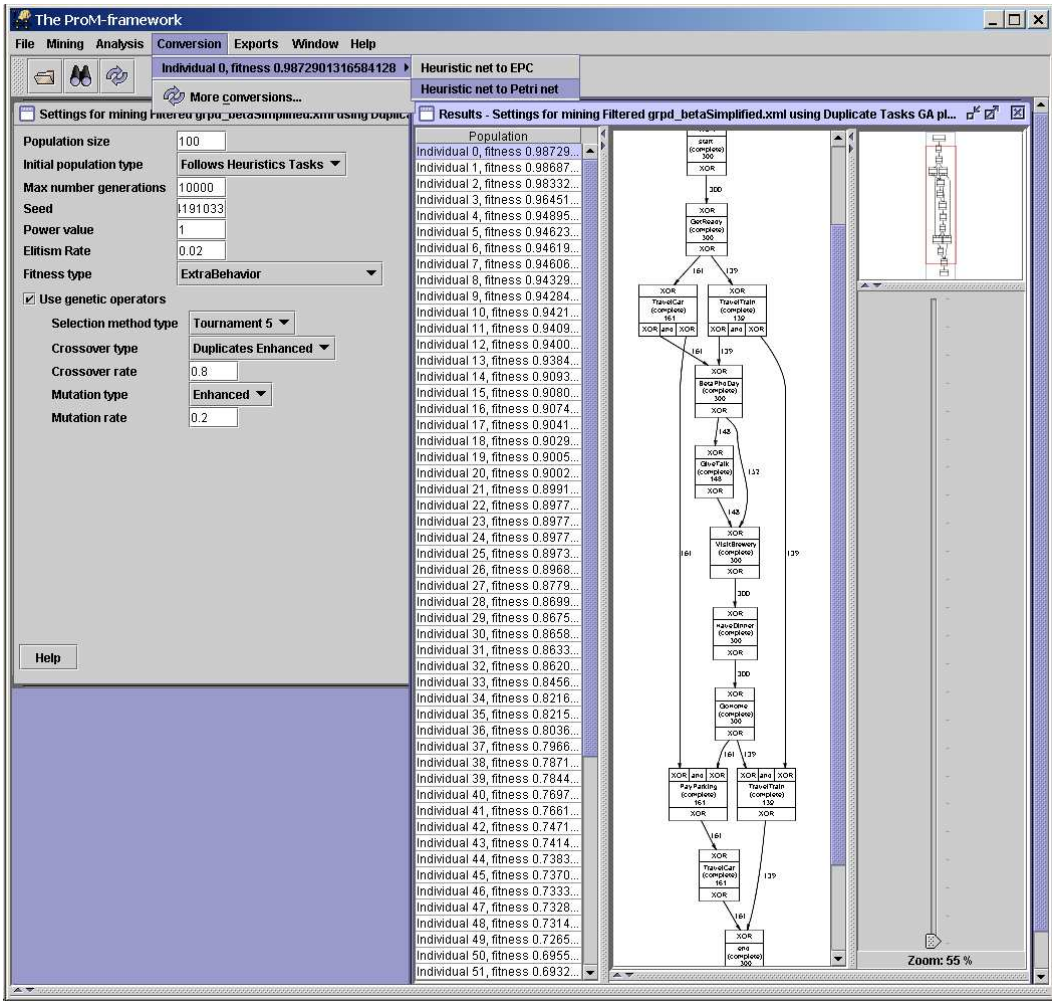


Figure 5.5: Screenshot of the "Duplicate Tasks GA plug-in" in the ProM framework. This screenshot shows the result of mining an event log like the one in Table 5.1. This log has 300 process instances in total. The left-side window shows the configuration parameters (see Subsection 5.5.2). The right-side window shows the best mined individual. Additionally, in the menu bar we show how to convert this individual (called "Heuristic net" in the ProM framework) to a Petri net.

- are respectively linked to a same label - i.e., $Label(c_1) = Label(c'_1)$ and $Label(c_2) = Label(c'_2)$. Since for the basic genetic algorithm the labelling function is required to be injective, two individuals have at most one causality relation in common for every pair of labels. However, when duplicates are allowed, this is not the situation anymore because *causality relations involving different activities may map to the same pair of labels*. For this reason, we had to extend the analysis metrics structural precision and recall to deal with bags. More specifically, the function *mapToLabels* in definitions 27 and 28 was extended to return a bag of labelled pairs. In a similar way, the intersection operator ($\cap$) now works over bags. The intersection of two bags returns the minimum amount of elements that are common to both bags. The "extended" version of the structural precision and recall are respectively formalized in Definitions 32 and 33. A more detailed discussion about the structural precision and recall metrics can be found in Subsection 4.5.1 of the basic genetic algorithm.

Definition 32 (Structural Precision - S_P). [7] *Let CM_o and CM_m be the respective causal matrices for the original and the mined models. Let the ranges of the labelling functions of the original and mined models be the same, i.e., $rng(Label_o) = rng(Label_m)$. Then the structural precision $S_P : \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ is a function defined as:*

$$S_P(CM_o, CM_m) = \frac{|mapToLabels(C_o, Label_o) \cap mapToLabels(C_m, Label_m)|}{|mapToLabels(C_m, Label_m)|}$$

where:

- *$mapToLabels(C, Label)$ is a function that applies the labelling function $Label$ to every element of a tuple in the causality relation C. For instance, if $C = \{(a_1, a_2), (a_2, a_3), (a_2, a_4)\}$ and $Label = \{(a_1, a), (a_2, b), (a_3, c), (a_4, c)\}$, then the function $mapToLabels(C, Label) = [(a, b), (b, c)^2]$, i.e., a bag containing three elements.*

Definition 33 (Structural Recall - S_R). *Let CM_o and CM_m be the respective causal matrices for the original and the mined model. Let the ranges of the labelling functions of the original and mined models be the same, i.e., $rng(Label_o) = rng(Label_m)$. Then the structural recall $S_R : \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ is a function defined as:*

$$S_R(CM_o, CM_m) = \frac{|mapToLabels(C_o, Label_o) \cap mapToLabels(C_m, Label_m)|}{|mapToLabels(C_o, Label_o)|}$$

[7] For both definitions 32 and 33, whenever $|mapToLabels(C, Label)|$ is equal to 0, the whole division is equal to 0.

The analysis metrics *duplicates precision* (D_P) and *duplicates recall* (D_R) check if the original model (the one used to create the synthetic logs for the experiments) and the mined model (the one returned by the DGA) have the same amount of duplicates. These two metrics are respectively formalized in definitions 34 and 35. The duplicates recall (D_R) assesses how many duplicates the original model has that are not in the mined model. The duplicates precision (D_P) quantifies how many duplicates the mined model has that are not in the original model. The duplicates precision indicates if the mined models tend to over-specialize. Thus, if we assume that a mined model is complete to a certain log, we can say that this mined model (i) has as many duplicates as the original model if their D_P and D_R are equal to 1; (ii) has more duplicates than the original model if $D_P < 1$ and $D_R = 1$ (for instance, compare the original model in Figure 5.4(a) with the mined models in Figure 5.4(b)); and (ii) has fewer duplicates than the original model if $D_P = 1$ and $D_R < 1$ (for instance, compare the original model in Figure 5.3 with the mined model in Figure 4.1, for the log in Table 4.1). In short, the closer the metric *duplicates precision* is to one the better. In other words, over-specific individuals have a D_P value that is closer to zero. The metrics duplicates precision and recall allow us to check for the *folding* requirement introduced in Section 5.2.

Definition 34 (Duplicates Precision - D_P). [8] *Let CM_o and CM_m be the respective causal matrices for the original and the mined models. Let the ranges of the labelling functions of the original and mined models be the same, i.e., $rng(Label_o) = rng(Label_m)$. Then the duplicate precision $D_P : \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ is a function defined as:*

$$D_P(CM_o, CM_m) = \frac{|toLabels(A_o, C_o, Label_o) \cap toLabels(A_m, C_m, Label_m)|}{|toLabels(A_m, C_m, Label_m)|}$$

where:

- *$toLabels(A, C, Label)$ is a function that applies the labelling function $Label$ to every element of the set A that belongs to at least one tuple in the causality relation C. For instance, if $A = \{a_1, a_2, a_3, a_4\}$, $C = \{(a_1, a_3), (a_1, a_4)\}$ and $Label = \{(a_1, a), (a_2, b), (a_3, c), (a_4, c)\}$, then the function $toLabels(A, C, Label) = [a, c^2]$.*

Definition 35 (Duplicates Recall - D_R). *Let CM_o and CM_m be the respective causal matrices for the original and the mined models. Let the ranges of the labelling functions of the original and mined models be the same, i.e.,*

[8] For both definitions 34 and 35, whenever the denominator $|toLabels(A, C, Label)|$ is equal to 0, the whole division is equal to 0.

$rng(Label_o) = rng(Label_m)$. Then the duplicates recall $D_R : \mathcal{CM} \times \mathcal{CM} \rightarrow [0, 1]$ is a function defined as:

$$D_R(CM_o, CM_m) = \frac{|toLabels(A_o, C_o, Label_o) \cap toLabels(A_m, C_m, Label_m)|}{|toLabels(A_o, C_o, Label_o)|}$$

The metrics presented in this section, as well as the metric for *completeness* (cf. $PF_{complete}$ in Definition 21) and the metrics for *preciseness* (behavioral precision B_P and recall B_R as introduced in definitions 25 and 26), are used to analyse the experiments results. These metrics are complementary and should be considered together. As for the basic genetic algorithm, the DGA mines a model that is as complete and precise as the original one whenever the metrics $PF_{complete}$, B_P and B_R are equal to 1. More specifically, the mined model is exactly like the original model when all the seven metrics (i.e, $PF_{complete}$, B_P, B_R, D_P, D_R, S_P and S_R) are equal to 1. As a general rule, the closer the values of all seven metrics are to 1, the better.

5.5.2 Setup

The Duplicates Genetic Algorithm was tested using noise-free event logs from 32 different process models [9]. These models contain constructs like sequence, choice, parallelism, loops, non-free-choice, invisible tasks and duplicate tasks. From the 32 models, 14 nets have duplicate tasks and were mostly copied from the models in [50]. The other models were mostly created by the authors and they do not contain duplicates. The 32 models had between 6 and 42 tasks. Every event log was randomly generated and contained 300 process instances. To speed up the computational time of the genetic algorithm, the similar traces were grouped into a single one and a counter was associated to inform how often the trace occurs. Traces were grouped together if they had exactly the same execution sequence of tasks. For every log, 50 runs were executed. Every run had a population size of 100 individuals, at most 1,000 generations, an elite of 2 individuals, a κ and γ of 0.025 (see Definition 31), a power value of 1 while building the initial population (see Subsection 5.4.1), and respective crossover and mutation probabilities of 0.8 and 0.2 (see the Section 5.3). All the experiments were run using the ProM framework. We implemented the DGA and the metrics described in this chapter as plug-ins for this framework (see Chapter 7 for details).

[9] All models can be found in the Appendix B.

5.5.3 Results

The results for the setup explained in Subsection 5.5.2 are depicted in Figure 5.7 to 5.9. In the graphs, the nets that contain duplicate tasks are signaled with an "∗" (asterisk) after their names. Similarly, the nets that are not compliant with the constraints of the models the DGA targets (i.e., these nets have duplicates sharing input/output elements) have an "nc" (non-compliant) after their names. From the results, we can conclude that:

- Nets with constructs that allow for more interleavings are more difficult to mine for the following two reasons. The *first reason* is that more interleaving situations usually lead to the inferring of more duplicates per task while building the initial population (cf. Subsection 5.4.1). Note that, in these situations, more tasks tend to be directly preceded/followed by other tasks and, consequently, the DGA has to explore a bigger search space. The *second reason* is that, because the DGA's fitness is an extension of the GA's one, the DGA also has a tendency to benefit the individuals that correctly portrait the most frequent interleavings in the log. As an illustration, consider the results for the nets:
  - parallel5, herbst6p45 and herbstFig5p1AND. These nets respectively have a five-, three- and two-branch parallel construct. Furthermore, all branches have a single task, except that one of the branches of herbst6p45 contains two tasks. From these three nets, the amount of possible interleavings are 120 (= 5!) for parallel5, 12 for herbst6p45 and 4 for herbstFig5p1AND. Note that, except for the duplicates precision metric, the net parallel5 has the worst results (among the three nets) for the metrics in Figure 5.7 to 5.9. In other words, the mined models for parallel5 are less complete and precise than the mined models for the other two nets (cf. Figure 5.8 and 5.9). Furthermore, the mined models for parallel5 contain more extra duplicates than the ones for herbst6p45 (cf. Figure 5.7). On the opposite way, the net herbstFig5p1AND has the best results of the three nets. The only exception is for the metric duplicates precision. This happens because the mined models for the net herbstFig5p1AND contain the specialization illustrated in Figure 5.4. Note that this kind of specialization is not punished by the folding requirement of the fitness measure in Definition 31.
  - herbstFig6p34 and herbstFig6p10. These two nets are very similar and have the same bag of tasks. However, they differ because the

net herbstFig6p34 (i) has one extra loop construct that is not in herbstFig6p10 and (ii) has one extra task in one of the branches of the parallel construct that also appears in herbstFig6p10. The presence of this longer parallel branch and the extra loop allow for more possible interleavings for the net herbstFig6p34 than for the net herbstFig6p10. As the results in Figure 5.7 to 5.9 point out, the mined models for herbstFig6p10 are better than the ones for herbstFig6p34. The difference is more critical for the completeness requirement (cf. Figure 5.9). While the mined models for herbstFig6p10 correctly capture on average 70% of the behavior in the logs (see "Average Proper Completion Fitness"), the ones for herbstFig6p34 can correctly parse less than 30% of the behavior in the log.

- The mined models for compliant nets that proportionally have more duplicates tend to be more folded than necessary. For instance, consider the results for the duplicates precision metric (see Figure 5.7). Note that the compliant nets herbstFig6p34, herbstFig6p10 and flightCar are more folded (have smaller duplicates precision values) than the other compliant nets. The common characteristic of these nets is that at least 40% of their tasks are duplicates and some of these duplicates are in parallel constructs.
- Low-frequent skipping of tasks is difficult to mine. For instance, consider the net herbstFig6p25. As indicated by the results in Figure 5.8, the mined models for this net are among the less precise ones, and the average value for the strutural precision points out that the mined models did not correctly capture a considerable amount of connections that are in the original model (structural precision inferior to 0.9). Furthermore, the results in Figure 5.9 indicate that the average proper completion fitness of the mined models is very low (less than 0.3). By inspecting the mined models for this net, we discovered that some of its tasks did not contain output elements. Thus, this justifies the inability to proper complete. Actually, the mined models have problems to correctly capture a part of the net that contains many skipping of tasks. This construct is highlighted in Figure 5.6.

- As expected, the non-compliant nets herbstFig6p42 and herbstFig6p39 could not be correctly mined. These nets are more folded (cf. Figure 5.7) than necessary and, consequently, less precise (cf. Figure 5.8). Besides, the results are worse for herbstFig6p42 than for herbstFig6p39. In both cases, the heuristics used to build the initial population (cf. Subsection 5.4.1) are enough to capture the minimum amount of duplicates, but the original models have duplicates sharing input/output elements. Thus, mined models are going to be punished by the folding requirement of the fitness measure. Actually, as shown in Figure 5.9, none of the mined models is precise for these two non-compliant models (see metric "% of Complete Models that are Precise").
- Whenever the behavior modelled with non-free-choice constructs could also be modelled with duplicate tasks, the DGA mined models with duplicate tasks. This is the case for the nets a6nfc and driversLicense. Note that these nets have more duplicates than necessary (as indicated by the duplicates precision metric in the top graph in Figure 5.7), but many mined complete models are also precise (cf. top graph in Figure 5.9). Since during the analysis of these two nets we have compared the mined models with original models that use non-free-choice constructs, we can conclude the non-free-choice constructs were replaced by duplicate tasks.
- No mined model for the net betaSimplified is precise. betaSimplified is the only net that contains duplicates and non-local non-free-choice construct at the same time. Although many of the mined models for this net are complete, they are not precise. Since they are complete, and the duplicates precision and recall values in the top graph of Figure 5.7 indicate that these models are as folded as the original ones, we can conclude that the duplicates are correctly captured and the non-preciseness is due to the absence of (part of) the non-local non-free-choice construct. This situation suggests that the DGA would need more iterations to correctly capture the non-free-choice construct.

Based on the remarks above and the results in Figure 5.7 to 5.9, we can conclude that the DGA is able to discover the right model in many situations. However, the DGA approach requires more iterations than the GA to find a complete and precise model. This is probably because it needs to explore a bigger search space. Additionally, although the heuristics based on the follows relation seems to be enough to capture the minimal amount of duplicates per task in the log (since the duplicates precision is usually lower than the duplicates recall), the returned models tend to have more duplicates than necessary. The models are not exactly over-specific (since the

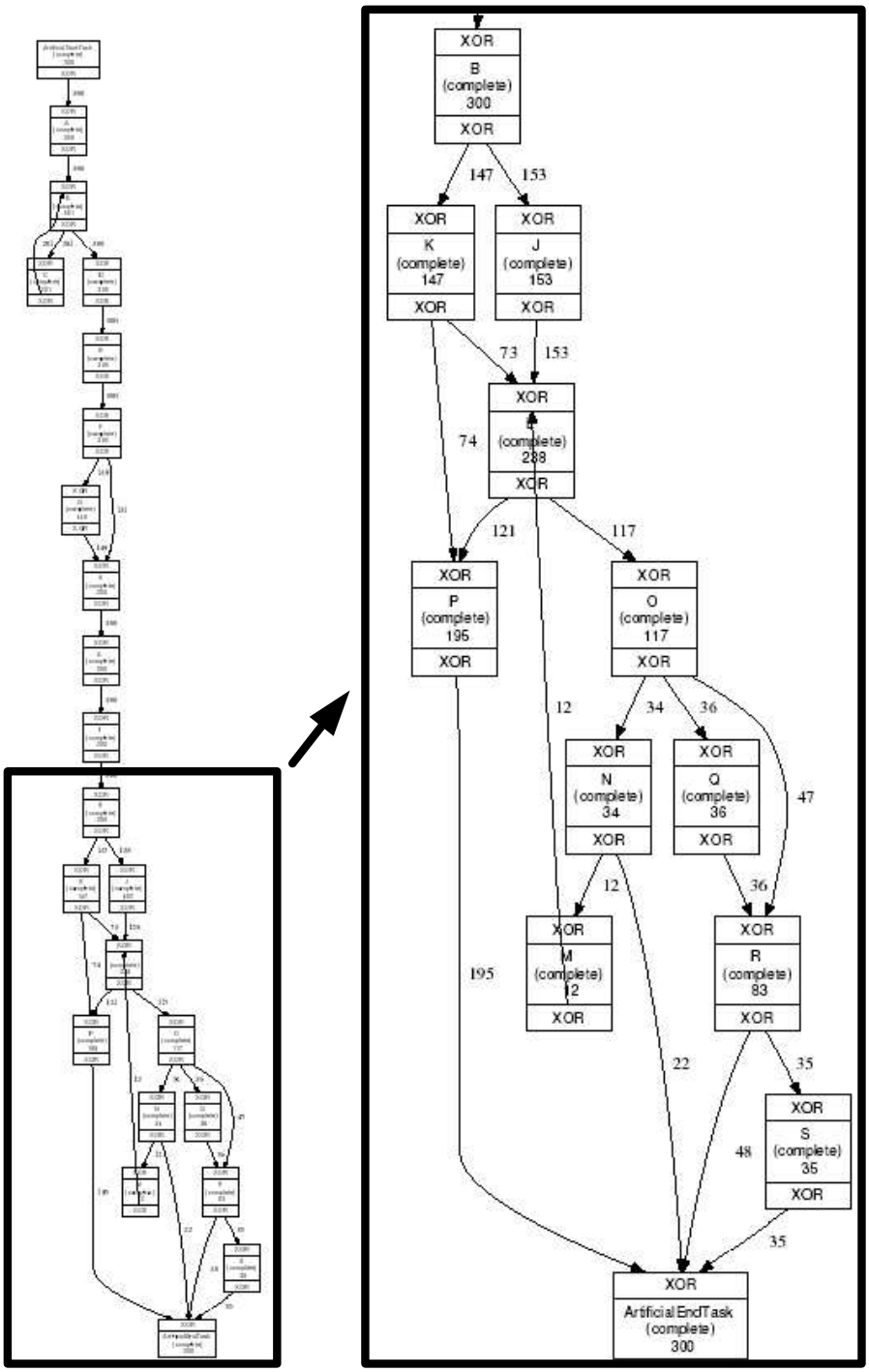


Figure 5.6: Heuristic net for the original model of the net herbstFig6p25. The zoomed part shows the construct with the low-frequent skipping of tasks. The mined models could not correctly mine this construct.

average values for the duplicates precision metrics are above 0.75), but they do tend to be more specific than necessary. Thus, to check if the DGA indeed would return better models if it would iterate more, we ran new experiments for a set of nets with duplicates. The setup was like the one described in Subsection 5.5.2, but using the 10,000 generations (10 times the initial number of generations). The set of nets with duplicates was chosen based on the amount of time it would take to complete one run (the faster ones were selected). In total, 11 nets with duplicate tasks were selected. The results are depicted in Figures 5.10 to 5.12. For every figure, we show the situation for 1,000 generations (top) and the situation for 10,000 generations (bottom). As can be seen, the DGA did perform better for the 10,000 generations than for 1,000 generations. The difference is more remarkable for the results shown in Figure 5.12. Note that all mined models, except for the non-compliant ones, have much better values for the average proper completion fitness, the percentage of complete models and the percentage of complete models that are precise. For instance, some of the mined models for the net betaSimplified also captured the non-local non-free-choice construct (for 1,000 iterations, none of the mined models were complete and precise). Furthermore, all the mined models for the net flightCar are complete (before, only 70% of the models were complete). The nets herbstFig6p33 and herbstFig6p31 also increased significantly the number of complete and precise mined models. This means that the DGA was able to mine the missing connections for the experiments with 1,000 generations. In short, all the average values for the metrics of the mined models improved after it has run for 10,000 generations.

5.6 Summary

This chapter presented the *Duplicates Genetic Algorithm* (DGA). The DGA is an extension of the basic GA explained in Chapter 4 to allow the mining of process models with duplicate tasks. The DGA aims at mining process models in which the duplicates can be distinguished based on the local context in which they are inserted. The local context of a task t is determined by the tasks that are input and output elements of t. Thus, the DGA only aims to mine models in which the duplicates do not share input elements and do not share output elements.

The extensions to the GA consisted of (i) removing the constraint that the labelling function *Label* of a causal matrix must be injective, (ii) the use of the follows relations to set the maximum amount of duplicates that a task in the log can have, (iii) adding the folding requirement to the fitness measure and adapting the continuous semantics parsing to work with duplicates,

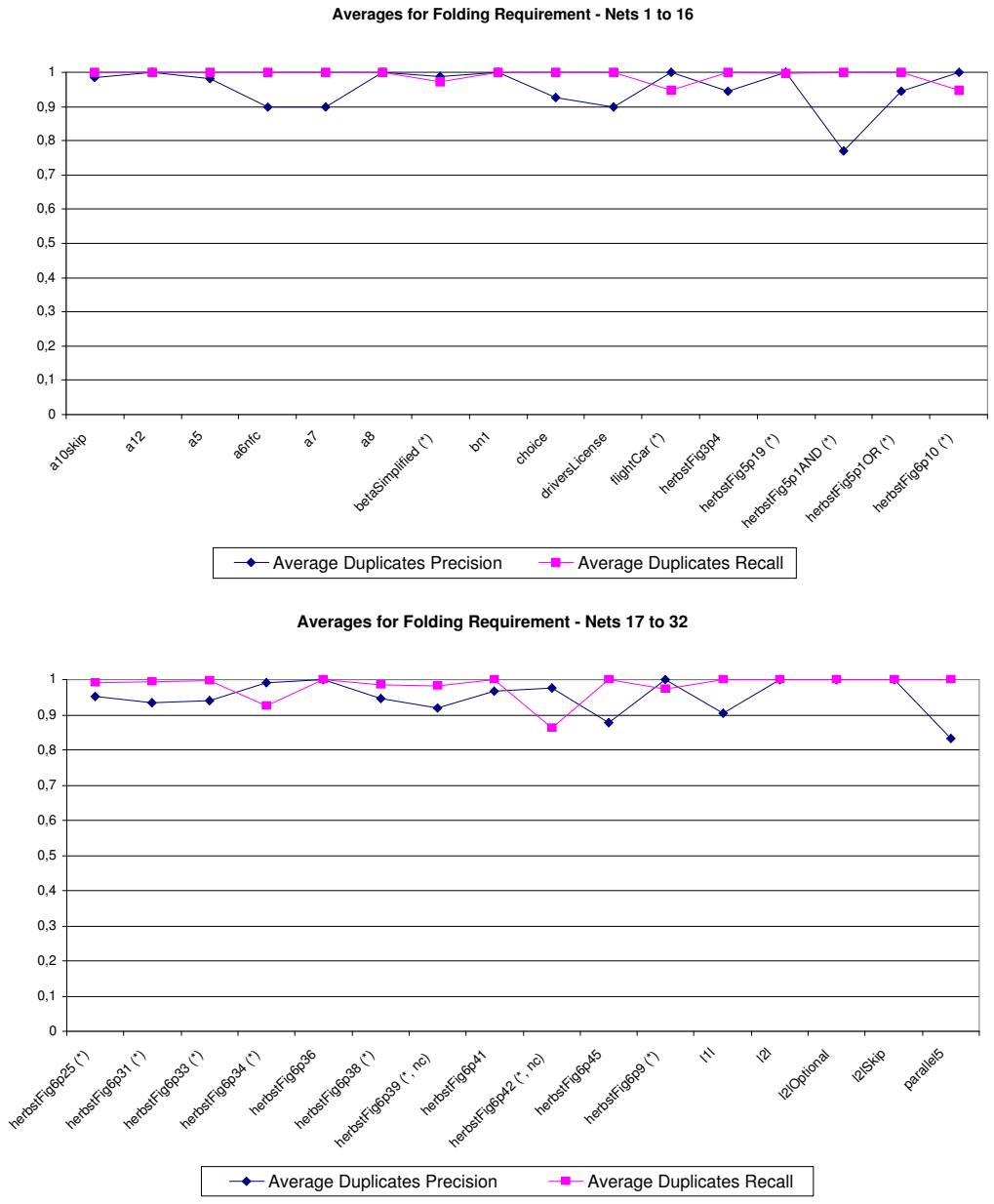


Figure 5.7: Average values for the duplicates precision and recall. Although the mined models are not over-specific (average duplicates precision ≥ 0.75), they tend to have more duplicates than necessary (average duplicates recall < 1.0). Additionally, some models have fewer duplicates than necessary. These models are either non-compliant (like herbstFig6p42), or have proportionally more duplicates than the other compliant models (like herbstFig6p34 and herbstFig6p10).

Figure 5.8: Averages values for the behavioral/structural precision and recall. The results show that the mined models do not tend to be over-general (behavioral precision ≥ 0.74), but the mined models for non-compliant nets (like herbstFig6p42) and nets with (i) more than two parallel branches (like parallel5 and herbstFig6p41) and/or (ii) longer parallel branches (like herbstFig6p34) are less precise. In addition, the mined models have a tendency to contain duplicates tasks instead of non-free-choice constructs whenever they can be used to model a same behavior (like in the net driversLicense).

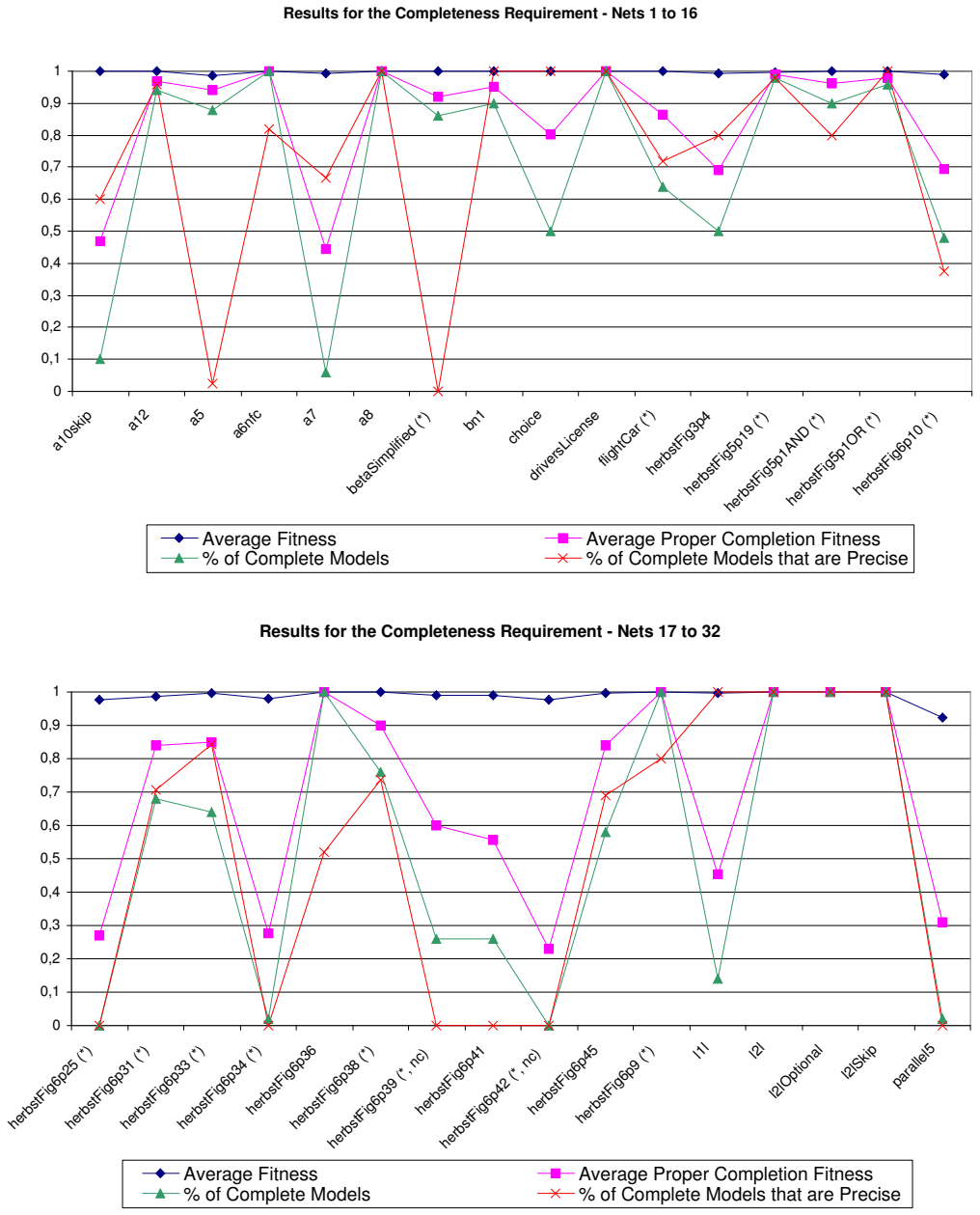


Figure 5.9: Averages values for the completeness requirement and derived metrics. The average fitness indicates that the mined models can correctly parse many substructures in the original nets (average fitness close to 1), but there are also mined models that do not correctly express the most frequent behavior in the log (see nets with average process completion fitness inferior to 0.5). The problems are again related to parallelism (e.g. parallel5 and herbst6p34), non-compliance (e.g. herbstFig6p42) and low-frequent observations of dependencies to skip parts of the process (e.g. herbstFig6p25).

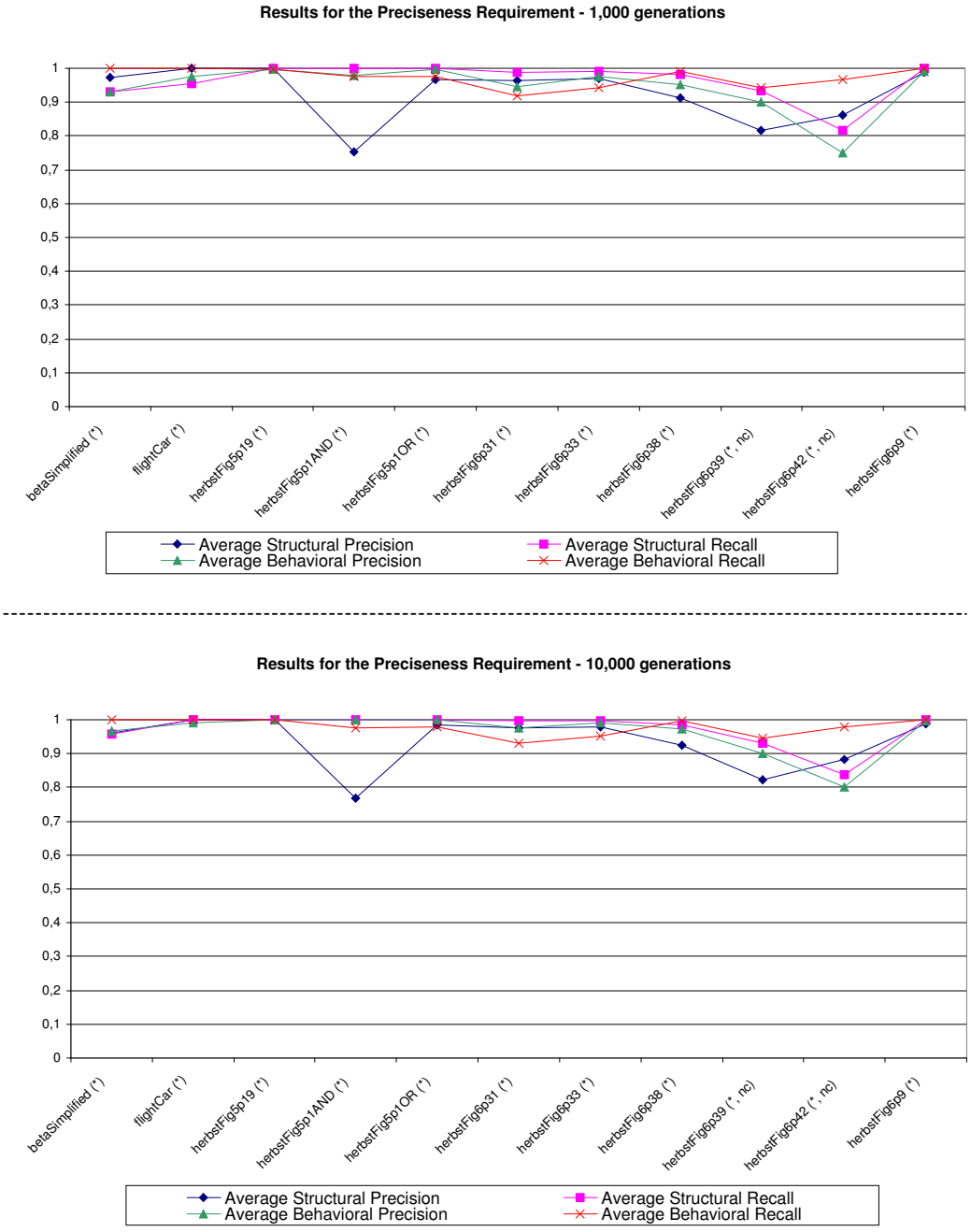


Figure 5.10: Comparison between the results for 1,000 generations and 10,000 generations for the average values of the behavioral/structural precision and recall. Note that all the average values improved when the DGA was given more iterations. As an illustration, consider the net betaSimplified. Note that this net had a significant improvement for the metrics behavioral precision and structural recall. betaSimplified looks like the net in Figure 5.1. Thus, this improvement means that more arcs related to the non-local non-free-choice construct were correctly captured in the mined models.

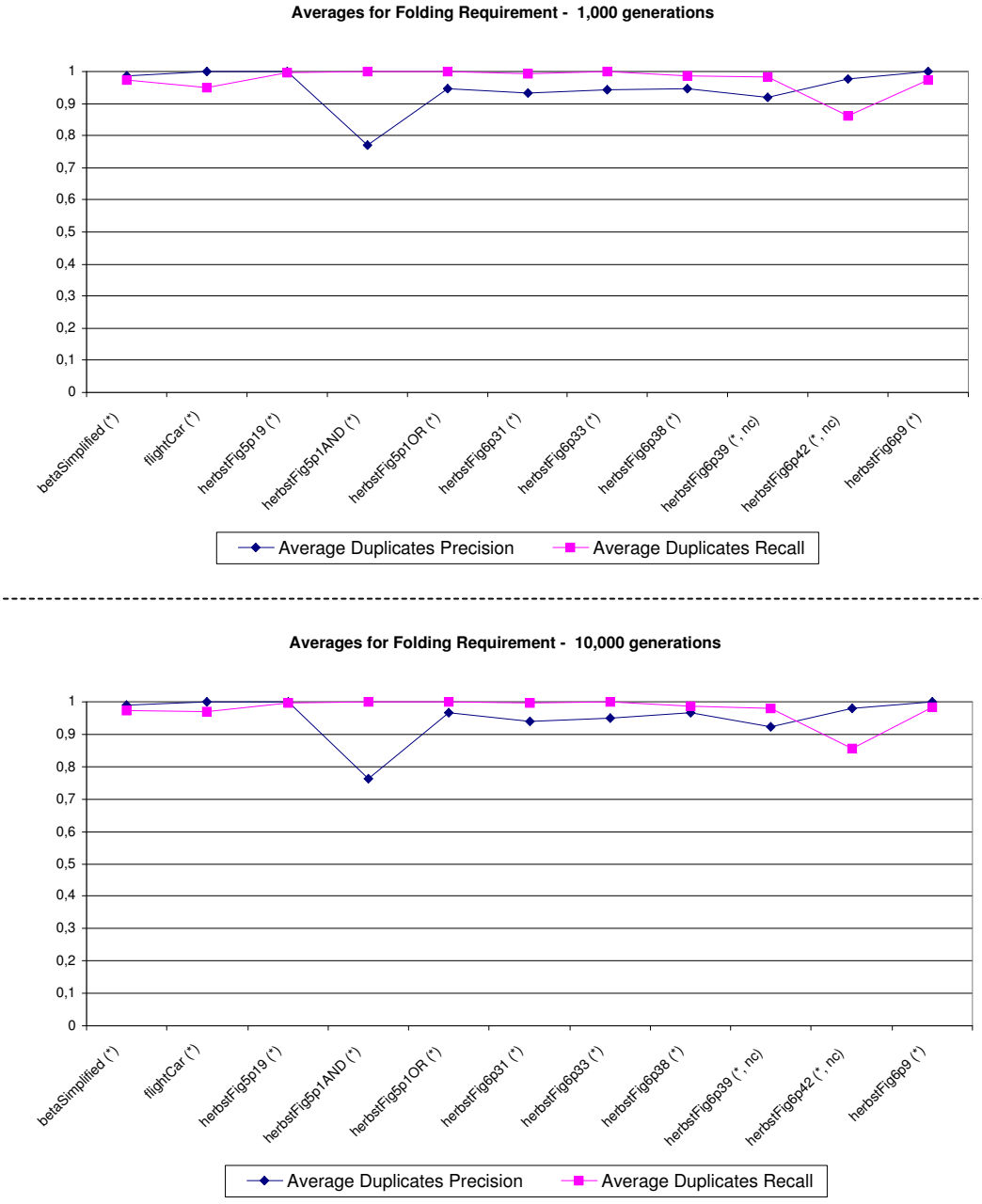


Figure 5.11: Comparison between the results for 1,000 generations and 10,000 generations for the average values of the duplicates precision and recall. All values improved slightly. Note that the values for the net herbstFig5p1AND did not change because the extra duplicates are due to the kind of specialization illustrated in Figure 5.4. This specialization is not punished by the folding requirement of the DGA's fitness.

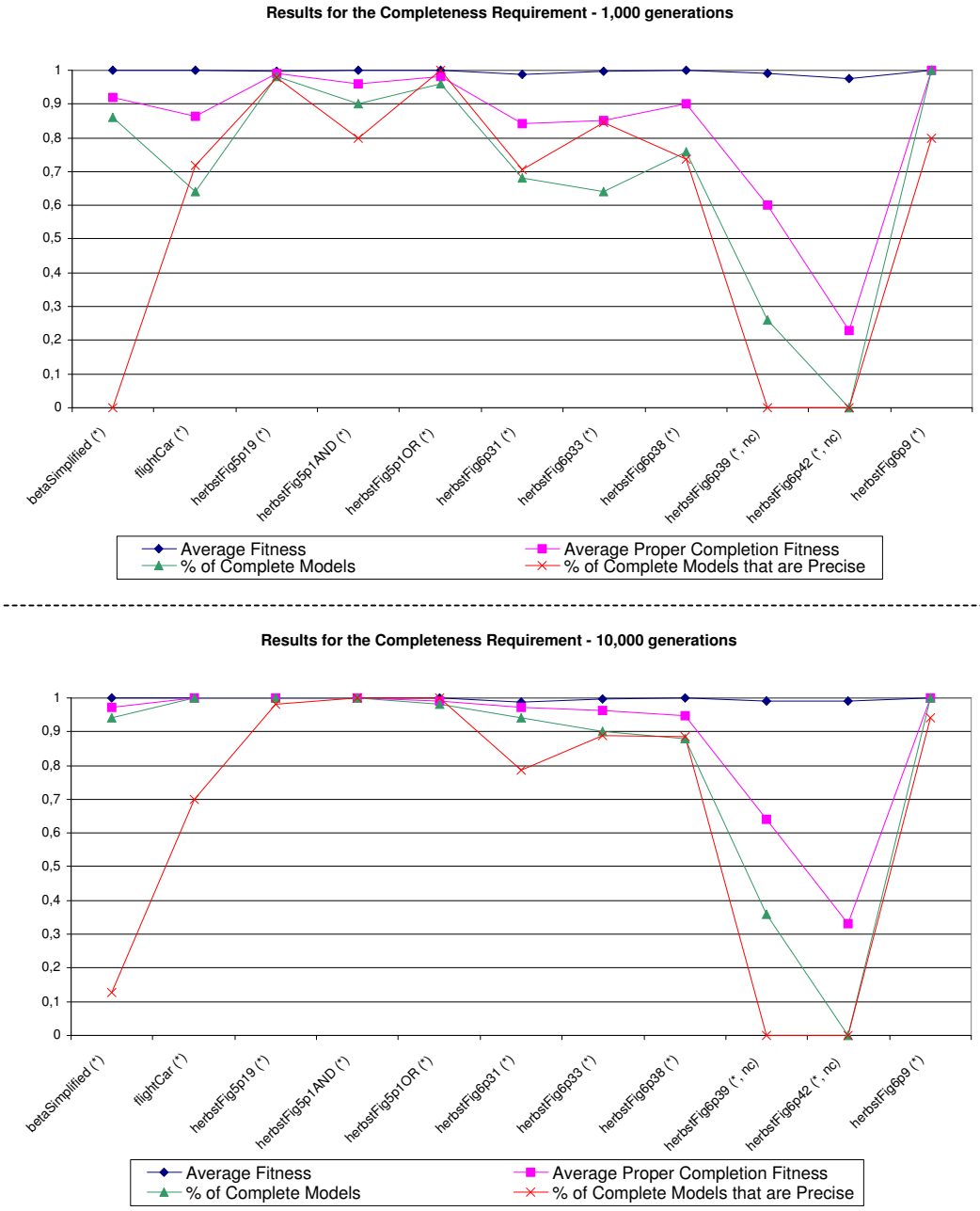


Figure 5.12: Comparison between the results for 1,000 generations and 10,000 generations for the average values of the the completeness metrics. Note that the number of complete (and precise) models increased remarkably when the DGA ran for a longer period of time. This is especially the case for the nets flightCar, herbstFig6p31 and herbstFig6p33. The only exceptions are the non-compliant nets because their results did not improve much.

and (iv) changing the crossover point from the activity level to the label level. Additionally, two new analysis metrics were introduced: duplicates precision and duplicates recall. These metrics quantify how close the mined and original models are with respect to the amount of duplicates that they contain.

The experiments show that the DGA can mine process models with duplicate tasks. However, the models tend to have more duplicates than necessary. The more the DGA iterates, the better the results. This shows that the fitness is guiding the search in the right direction. However, the overhead of the extra computational time cannot be neglected and, for sure, this is a drawback of the DGA.

The next chapter explains the arc post-pruning mechanism that we provide to "clean" the visual representation of mined models that the GA and DGA return for given (noisy) logs.

Chapter 6

Arc Post-Pruning

This chapter explains the approach we chose to handle mined models from noisy logs. Noise can be defined as *low-frequent incorrect* behavior in the log. A log may contain noise because some of its traces are incomplete (e.g., they correspond to running cases in the system that have not been completed yet), or the traces reflect incorrect behavior (e.g., due to some temporal system misconfiguration). Either way, the presence of noise may hinder the correct discovery of a process model. Noisy behavior is typically difficult to detect because it cannot be easily distinguished from other *low-frequent correct* behavior in the log (for instance, the execution of exceptional paths in the process). For this reason, and because both the GA and the DGA are designed to always benefit individuals that can correctly parse the *most frequent* behavior in the log, we have opted for a post-processing step to "clean" mined models from the effects of noise. In short, the post-processing step works by *pruning the arcs of a (mined) model that are used fewer times than a certain threshold*.

The main advantage of a post-pruning step is that, because it works independently of the process mining algorithm, it does not avoid the discovery of low-frequent behavior. Thus, if the mined low-frequent behavior is a correct one, it can remain in the model. If the mined low-frequent behavior corresponds to noisy behavior, the end user has the possibility to clean the mined model. Furthermore, arc post-pruning can also be used over any model to get a more concise view (in terms of the number of arcs in the model) of the most frequent behavior. As a final remark, we point out that arc post-pruning is also the approach adopted by the other related process mining techniques in [18, 23, 44, 52] to clean mined models. The remainder of this chapter provides more details about this arc post-pruning approach. Section 6.1 explains how the post-pruning works. Section 6.2 presents the experiments setup and the results. The experiments included logs with different types

of noise. The main aim of the experiments was to detect to which kinds of noise the GA and the DGA seem to be more sensitive. Section 6.3 provides a short summary of this chapter.

6.1 Post Pruning

The post-pruning step removes the arcs that are used fewer times than a given threshold from a (mined) model. The threshold refers to the arc usage percentage. The arc usage indicates the number of times that an arc (or dependency) is used when a log is replayed by an individual. The arc usage percentage defined by the threshold is relative to the most frequently used arc. As an illustration, assume that the most frequent arc usage of a mined model to a given log is 300. If the threshold is set for 5%, all arcs of this model that are used 15 or fewer times are removed during the post-pruning step. This situation is depicted in Figure 6.1. The mined model is in Figure 6.1(a). The pruned model is in Figure 6.1(b). Notice that the arcs of the mined model that were used (from left to right) 7, 5, 3 and 6 times are not shown in the pruned model. When the removal of arcs leads to dangling activities (i.e., activities without ingoing and outgoing arcs), these activities are also omitted in the post-pruned model.

6.2 Experiments and Results

This section explains the experimental setup and results. Our aim is to test how the GA and the DGA perform in the presence of noise. So, the first requirement to perform the experiments would be to have noisy logs. As explained in [58], different types of noise may affect a mining algorithm in different ways. Therefore, we have experimented with noisy logs that contain different noise types. The noise types are explained in Subsection 6.2.1. After deciding on the types of noise to consider, we selected some logs to experiment with. Due to the high computational time required by the GA and, especially, by the DGA, we have selected the nets that were reasonable fast (less than 3 hours to process one seed for at most 1,000 iterations) to compute the results. Additionally, these nets should have returned good results for the noise-free logs. This requirement makes sure that any failure in mining complete and precise models is exclusively due to the noise in the log, since the genetic algorithm can successfully mine models for the noise-free log. To facilitate the explanation, the remainder of this section is divided into two parts. First, in Subsection 6.2.1, the noise types we have experimented with are explained.

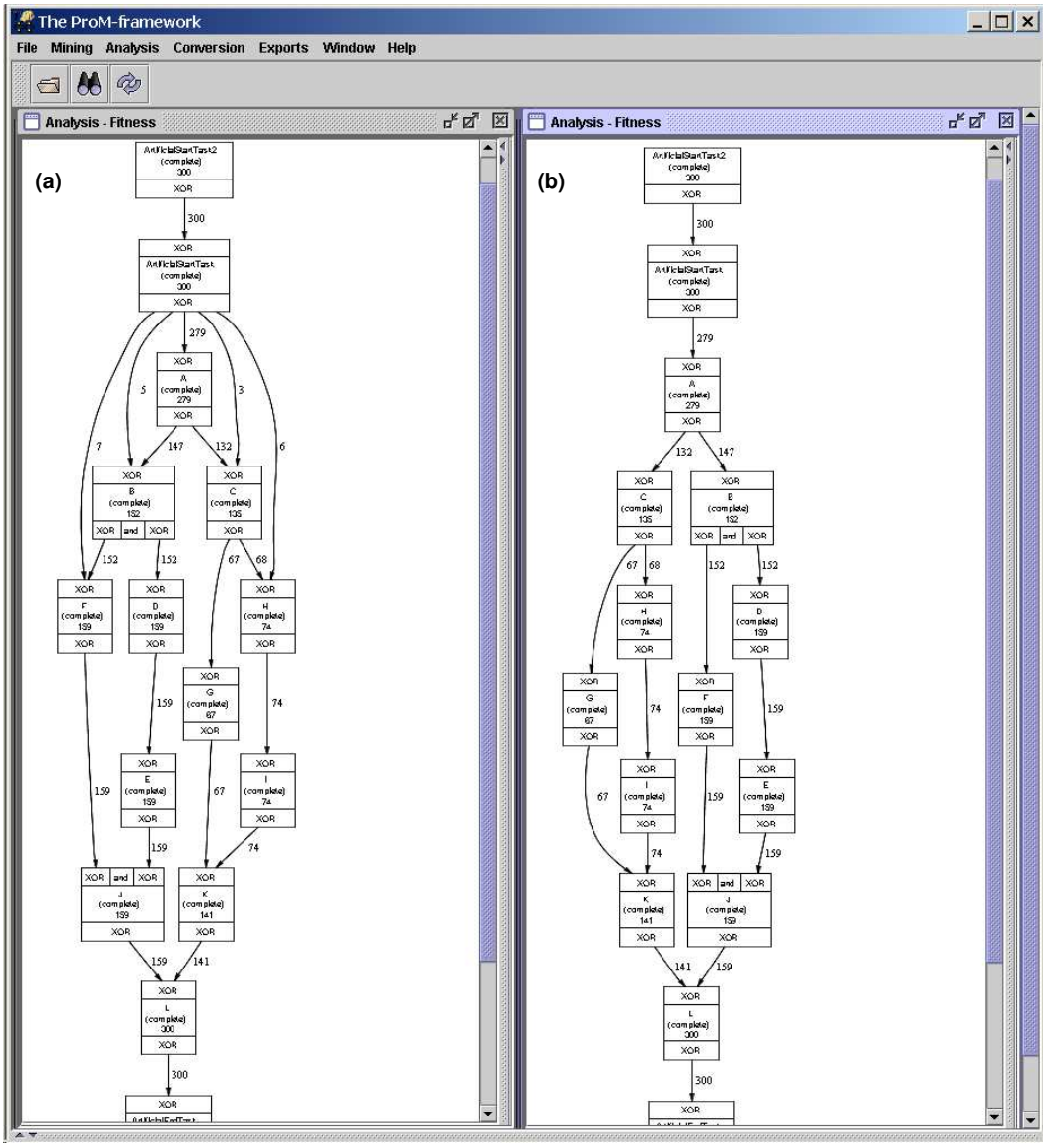


Figure 6.1: Illustration of applying post-pruning to arcs of a mined model. The mined model is in (a), and the resulting post-pruned model is in (b). The numbers next to the arcs in these models inform how often these arcs have been used while replaying the log for the models. The post-pruning illustrated here has a threshold of 5%. Thus, since the highest arc usage of the mined model in (a) is 300, all of its arcs that are used 15 or fewer times are not shown in the resulting pruned model in (b).

Second, in Subsection 6.2.2, the experiments setup and results for the GA and DGA are provided.

6.2.1 Noise Types

For the experiments, we used 6 different noise types: *missing head*, *missing body*, *missing tail*, *swap tasks*, *remove task* and *mix all*. These noise types are the ones described in [58]. If we assume a trace $\sigma = t_1...t_{n-1}t_n$, these noise types behave as follows. *Missing head, body* and *tail* respectively randomly remove sub-traces of tasks in the head, body and tail of σ. The head goes from t_1 to $t_{n/3}$ [1]. The body goes from $t_{(n/3)+1}$ to $t_{(2n/3)}$. The tail goes from $t_{(2n/3)+1}$ to t_n. The removed sub-traces contain at least one task and at most all the tasks in the head, body or tail. *Swap task* exchanges two tasks in σ. *Remove task* randomly removes *one* task from σ. *Mix all* randomly performs (with the same probability) one of the other 5 noise types to a traces in a log. Real life logs will typically contain mixed noise. However, the separation between the noise types allows us to better assess how the different noise types affect the GA and the DGA.

For every noise type, we generated logs with 5% and 10% of noise. So, every selected net in our experiments had $6 \times 2 = 12$ noisy logs. Note that the experiment was not exhaustive because we did not generate logs with multiple seeds for every noise type. We are aware that a more rigorous experimentation would require that. However, because we play with different nets, our setting is good enough to give insight into the sensitivity of the algorithms GA and DGA. The insertion of noise in a log works as follows. First we got the noise-free log for a net. Then, every trace of the log has the selected percentage (5% or 10%) of being modified. After the process has been repeated for every trace, we inserted two artificial start tasks and two artificial end tasks at every trace. This is necessary because our implementations for the GA and the DGA assume that the target model has a unique XOR-split/join start activity and a unique XOR-split/join end activity. However, this last step is implementation dependent and can be safely skipped in cases the implementation does not require that.

6.2.2 Genetic Algorithms

This section explains the experimental setup and the results of mining noisy logs with both the GA and the DGA. The aim of the experiments is to

[1] The division $n/3$ is rounded to the largest double value that is not greater than $n/3$ and is equal to a mathematical integer.

check how sensitive the GA and the DGA are to the different noise types (cf. Subsection 6.2.1). Finally, the experiments were analyzed based on the seven analysis metrics described in Chapter 5. The analysis metrics are: the completeness metric ($PF_{complete}$ in Definition 21), the behavioral precision and recall (B_P and B_R in Definition 25 and 26), the structural precision and recall (S_P and S_R in Definition 32 and 33), and the duplicates precision and recall (D_P and D_R in Definition 34 and 35). However, we did not use the metrics D_P and D_R to analyse the results obtained for the GA because this algorithm does not allow for duplicates.

Setup

Both the GA and DGA were tested over noisy logs from 5 different process models. The process models used for the GA are: a12, bn1, herbstFig3p4, herbstFig6p36 and herbstFig6p37. These models contain constructs like sequences, choices, parallelism, structured loops and non-local non-free-choice constructs, and have between 10 and 40 tasks. The process models used for the DGA are: a12, herbstFig3p4, herbstFig5p19, herbstFig6p36 and herbstFig6p9. These models contain constructs like sequences, choices, parallelism, non-local non-free-choice constructs, invisible tasks and duplicate tasks, and have between 7 and 16 tasks. All the models used to test the GA and DGA can be found in Appendix B. The noise-free log of every net has 300 traces (actually, these are the same noise-free logs used during the experiments reported in chapters 4 and 5). For every noise-free log, 12 noisy logs were generated: 6 logs with 5% noise and 6 logs with 10% noise. The 6 noise types used are the ones described in Subsection 6.2.1: missing head, missing body, missing tail, swap tasks, remove task and mix all. To speed up the computational time of the genetic algorithm, the similar traces were grouped into a single one and a weight was added to indicate how often the trace occurs. Traces with the same sequence of tasks were grouped together. For every noisy log, 50 runs were executed. For the GA, the configuration of every run is the same used for the noise-free experiments of the Scenario IV (see Subsection 4.5.2). For the DGA, the configuration of every run is like the one described in Subsection 5.5.2. After a run was complete, the mined model was used as input for a post-processing step to prune its arcs. Every mined model went two post-pruning steps: one to prune with a threshold of 5% and another to prune with a threshold of 10%. So, when analyzing the results, we look at (i) the mined model returned by the GA/DGA, (ii) the model after applying 5% pruning, and (iii) the model after 10% pruning. The post-processing step is implemented as the *Prune Arcs* analysis plug-in in the ProM framework (see Chapter 7 for details).

Results

The results for the experiments with logs of the net a12 are in Figure 6.2 to 6.5 for the GA and in Figure 6.6 to 6.11 for the DGA. We only show the results for the net a12 because the obtained results for the other nets lead to the same conclusions that can be drawn based on the analysis of the results for a12. Every figure plots the results before and after pruning. However, we have omitted the results for 10% arc-pruning because the results are just like the results for 5% arc-pruning. Furthermore, for every graph, the x-axis shows, for a given net (or original model), the noise type and the percentage of noise in the log. For instance, a12All10pcNoise is a short for "Noisy log for the net a12 (a12). The noise type is *mix all* (All) and this log contains at most 10% (10pc) of noise (Noise).". The y-axis contains the values for the analysis metrics.

Additionally, we have plotted the metric values for the mined model and original model with respect to the noisy log and the noise-free one. The reason is that the analysis with the noisy logs allow us to check if the mined models over-fit the data (since these noisy logs were given as input to the genetic algorithms). For instance, if some mined model can proper complete the noisy log (can parse all the traces without missing tokens or tokens left-behind), this mined model has over-fitted the data. On the other hand, when the model does not over-fit the data, the analysis with the noise-free logs can check if the mined model correctly captures the most frequent noise-free behavior (since the noisy logs used for the experiments were created by inserting noisy behavior into these noise-free logs).

In a nutshell, we can conclude that (i) both the GA and the DGA are more sensitive to the noise type *swap tasks* (and, consequently, *mix all*), (ii) the mined models do not tend to over-fit the noisy logs, and (iii) the DGA is more sensitive to noisy logs than the GA. More specifically, the results point out that:

- The GA and the DGA are more robust to 5% noise than to 10% noise. But the 5% arc post-pruning gave the same results as the 10% one.
- The GA is more robust to noisy logs than the DGA. As an illustration, compare the results in Figure 6.2 (for the GA) with the ones in Figure 6.6 (for the DGA). Note that the models mined by the GA capture a behavior that is much closer to the behavior of the original models. In other words, the values of the behavioral precision and recall metrics of the models returned by the GA (cf. Figure 6.2) are much closer to 1 (before and after pruning) than the values of these metrics for the models returned by the DGA (cf. Figure 6.6). This probably happens because the presence of noise may increase the search space of

the DGA. Note that the maximum amount of duplicates is set based on the follows relation. So, if noise is introduced, some tasks may have more direct predecessors/successors in the log than in the noise free situation. Furthermore, it is easier to over-fit when the model has duplicates.

- The pruning is more effective for the noise types *missing head*, *missing body*, *missing tail* and *remove task*. This makes sense because these noise types usually can be incorporated to the net by adding causality dependencies to *skip* the "removed" or "missing" tasks. In other words, the main net structure (the one also contained in the original model) does not change, only extra causality dependencies need to be added to it. This explains why the arc post-pruning works quite fine for these noise types.
- Related to the previous item, the noise type *swap tasks* affects the quality of the mined results the most. By looking at figures 6.2 and 6.4, one can see that the behavioral/structural precision and recall of the mined models for logs with swapped tasks (`a12Swap5pcNoise` and `a12Swap10pcNoise`) did not change dramatically after the pruning. This is probably because the over-fitting of the mined models to the logs involves more than the simple addition of causality dependencies. I.e., the main structure of mined models is more affected by the *swap tasks* noise type. A similar reasoning holds for the results in Figure 6.6 and 6.9.
- The mined models returned by the DGA have at least one activity for every task in the log (cf. Figure 6.8 and 6.11), but the pruned models do not. For instance, note that the average values of the metric duplicates precision of the pruned models `a12Swap5pcNoise` and `a12Swap10pcNoise` are lower than the values of metric duplicates recall. This means that some of the duplicates were connected to low-frequent input/output arcs.

6.3 Summary

This chapter explained the post-pruning step that can be executed over (mined) models. This post-pruning step works by eliminating from a (mined) model the arcs that are used fewer times than a given threshold. The threshold is proportional to the most frequently used arc. The choice for a post-pruning step is motivated by the fact that (i) it is very difficult to distinguish up-front between low-frequent correct behavior and low-frequent incorrect

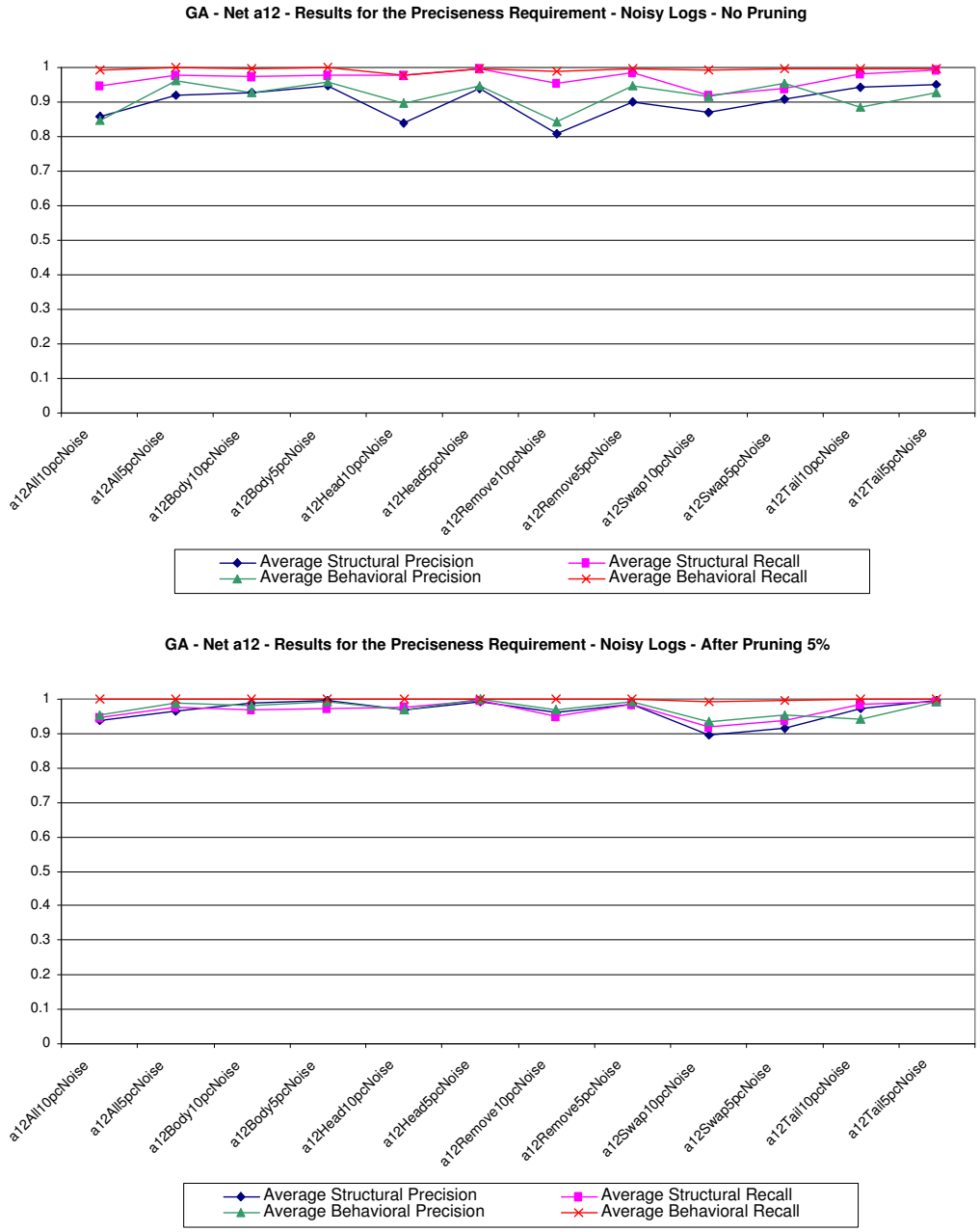


Figure 6.2: Average values for the behavioral and structural precision/recall metrics of the models mined by the GA for noisy logs of the net a12. The top (bottom) graph shows the results for the mined models before (after) arc post-pruning. The results show that (i) the mined models have a behavior that is quite similar to the original models (since behavioral precision > 0.8 and behavioral recall > 0.95), and (ii) the arc post-pruning is more effective for the noise types *missing head/body/tail* and *remove task* (since all the values plotted in the bottom graph are better than the ones in the top graph).

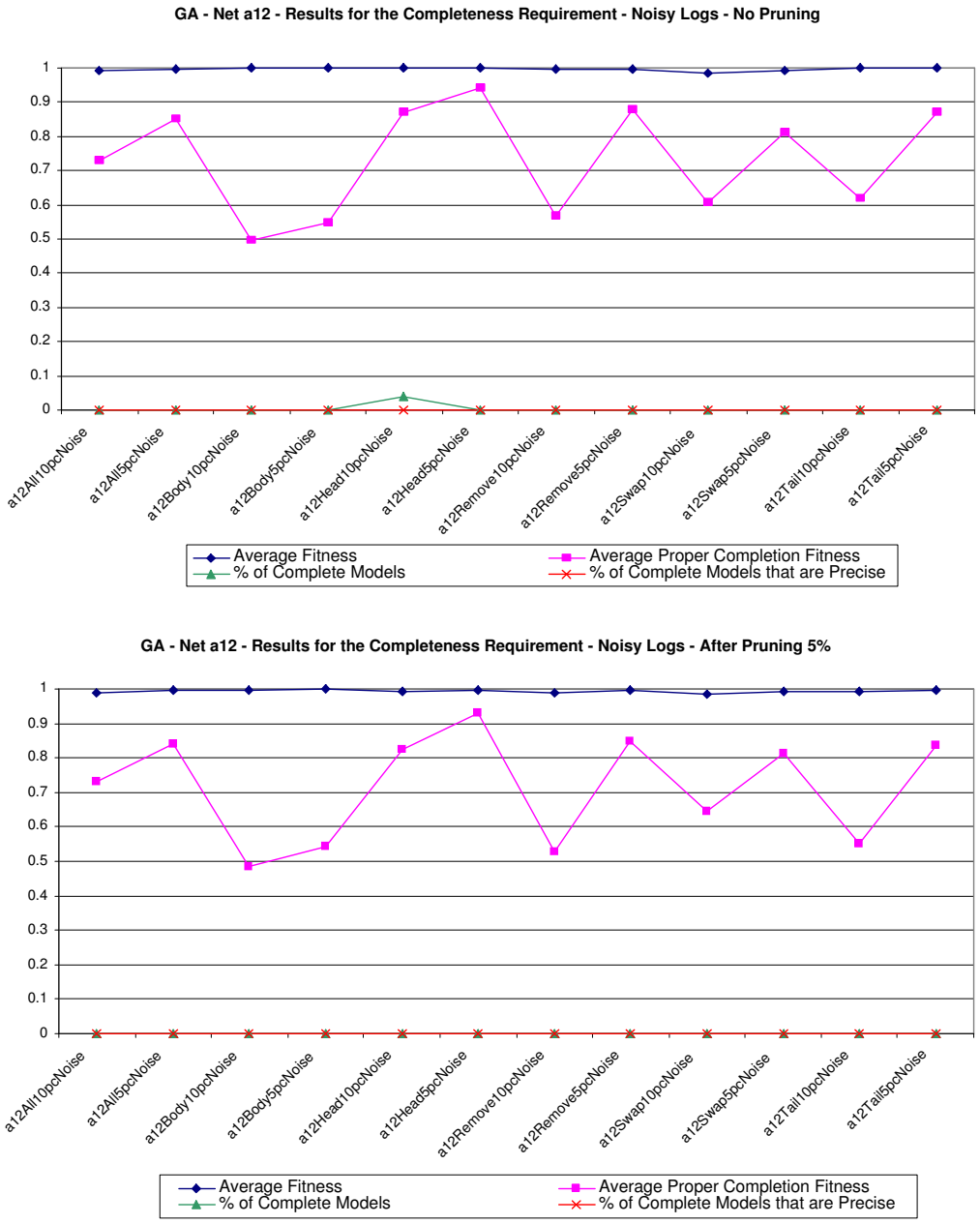


Figure 6.3: Results of the GA for the noisy logs of the net a12: completeness metrics. The metrics were calculated based on the noisy logs used during the mining. The top graph shows the results for the mined models. The bottom graph shows the results after the mined models have undergone 5% arc post-pruning. Note that the top graph indicates that only 2 models for the noise type *missing head* (10% noise) over-fit the data. However, overall the mined models did not over-fit the data, since the average proper completion < 0.95 and the % of complete models is 0 for almost all logs.

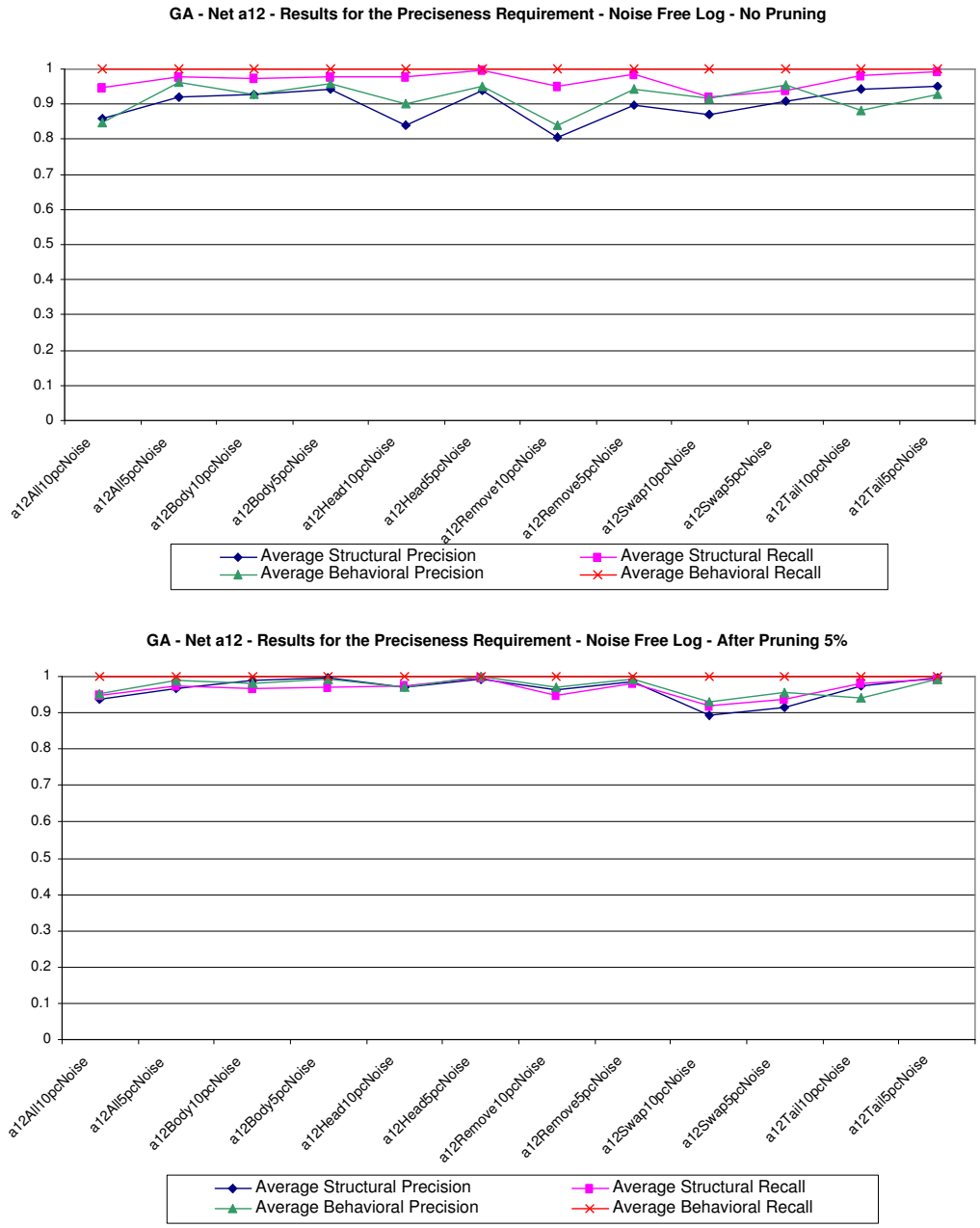


Figure 6.4: The results show the same metrics as explained for Figure 6.2, but these metrics are calculated base on a noise-free log of a12. Note that, contrary to the results in Figure 6.2, all mined models (before and after pruning) have an average behavioral recall that is equal to 1. This means that, with respect to the noise-free log, all the behavior allowed by the original model is also captured by the mined models.

Figure 6.5: Same metrics as in Figure 6.3, but this time the values are calculated based on a noise-free log of a12. The values for the average proper completion fitness point out that the models correctly captured at least 50% (see a12Body10pcNoise) of the behavior in the log. This indicates that the fitness indeed benefits the individuals that correctly model the most frequent behavior in the log. Besides, note that many more mined models are complete and precise after they have undergone arc post-pruning.

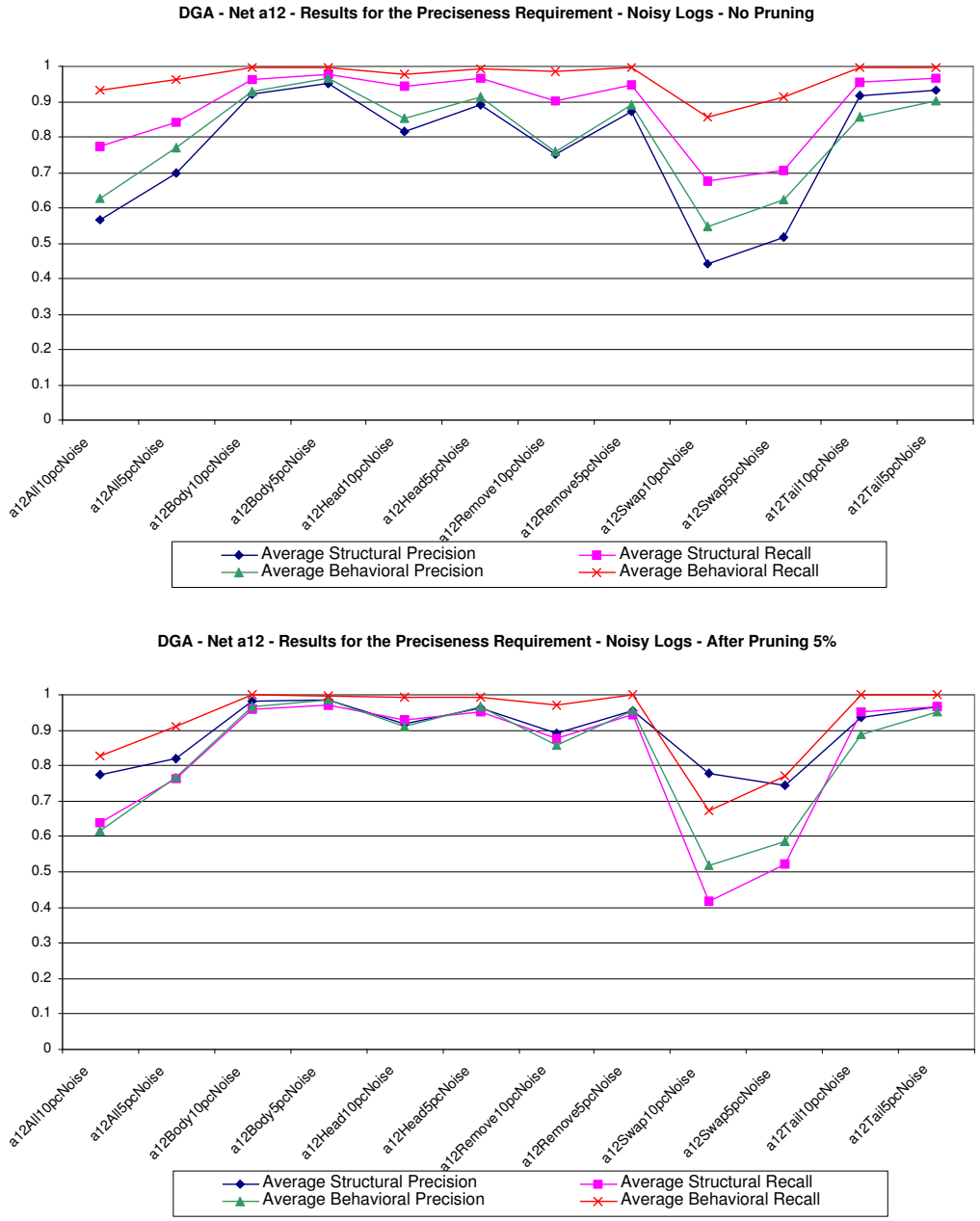
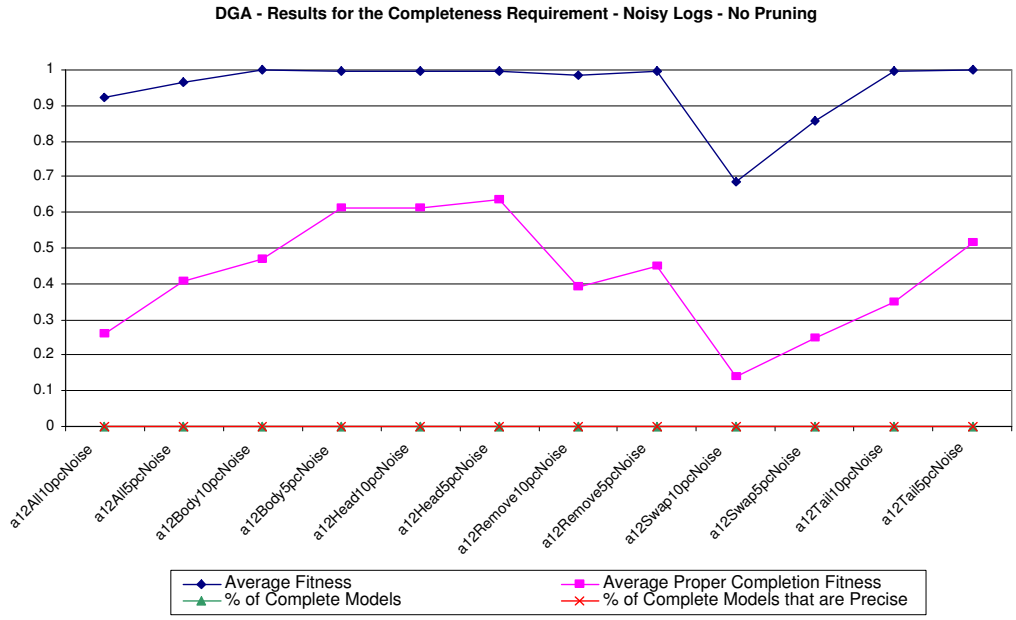


Figure 6.6: Average values for the behavioral and structural precision/recall metrics of the models mined by the DGA for noisy logs of the net a12. The top (bottom) graph shows the results for the mined models before (after) arc post-pruning. The results indicate that the DGA is more sensitive to the noise type *swap tasks* (note that the logs for a12Swap10pcNoise and a12Swap5pcNoise have the lowest average values for the metrics behavioral precision and recall) and, consequently, also the noise type *mix all*. Overall, the DGA is much more sensitive to noise than the GA (compare the results here with the ones in Figure 6.2).



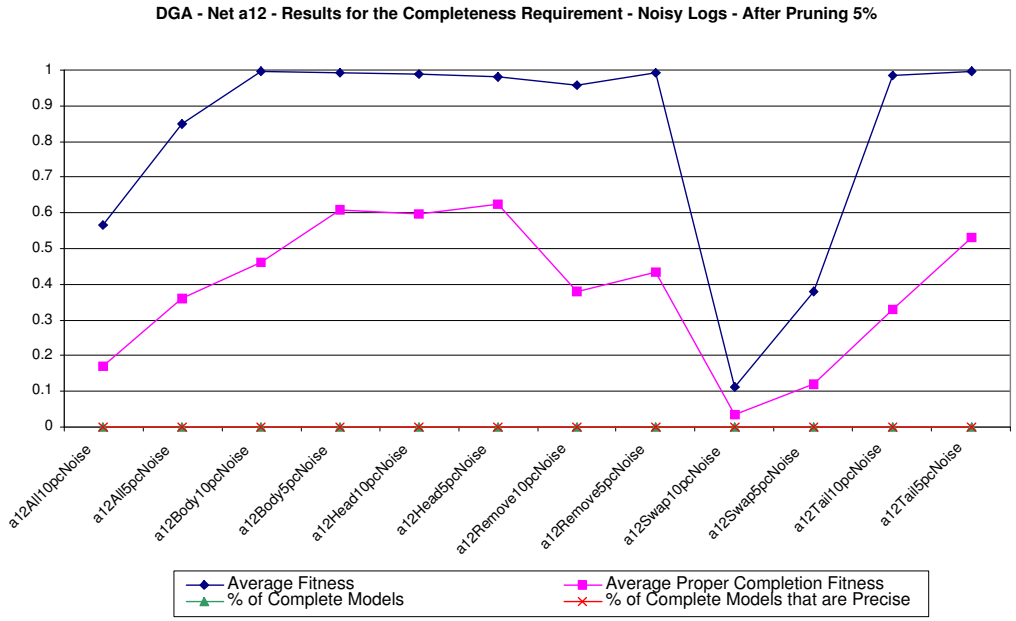


Figure 6.7: Results of the DGA algorithm for the noisy logs of the net a12: completeness metrics. The results suggest that the DGA would need more iterations to mine models that correctly capture the most frequent behavior in the log. Note that all the mined models can complete, on average, less than 60% of the traces in the logs.

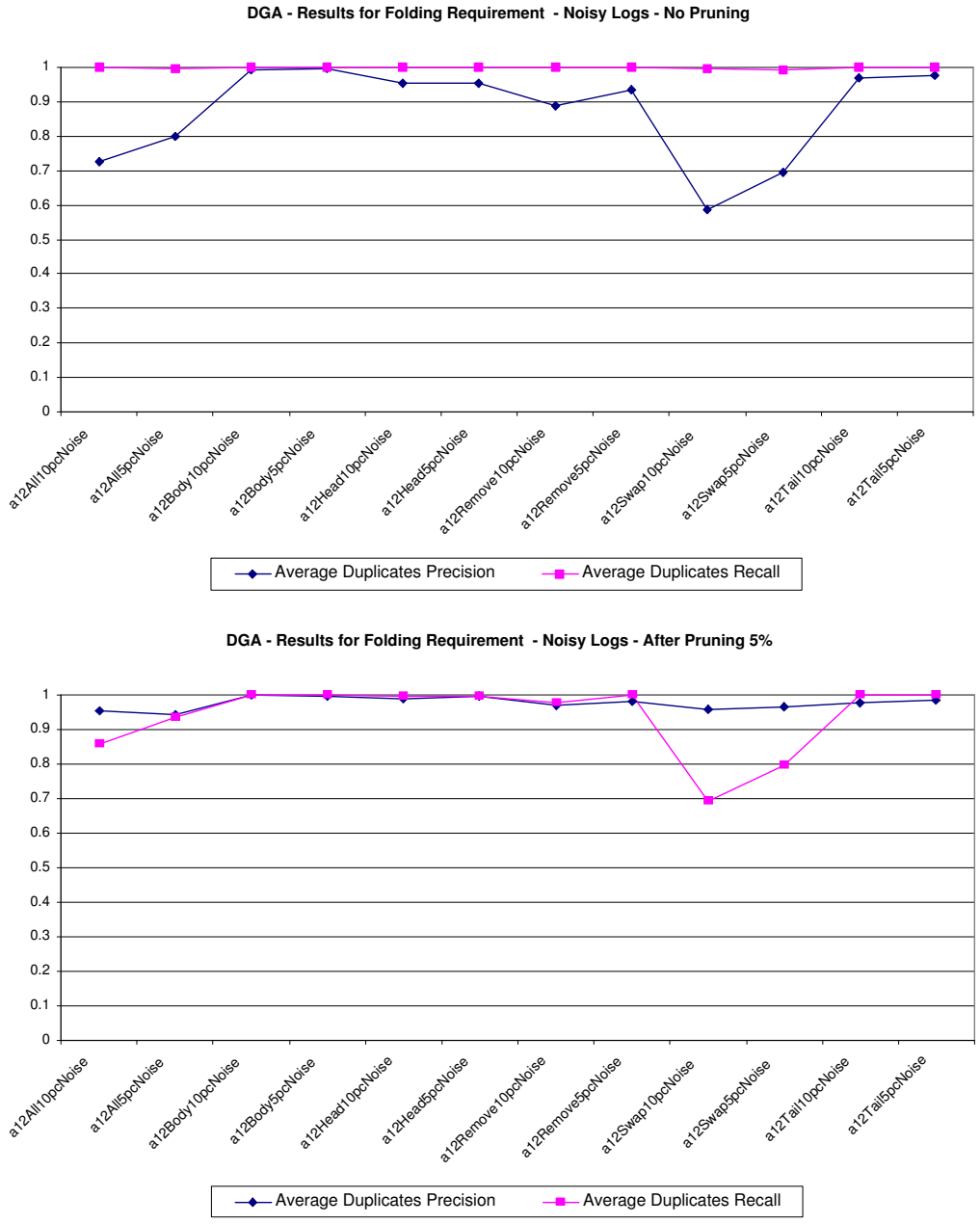


Figure 6.8: Results of the DGA algorithm for the noisy logs of the net a12: folding metrics. The metrics were calculated based on the noisy logs used during the mining. The top (bottom) graph shows the results for the mined (pruned) models. Except for the noise type *missing body*, all other noise types led to the mining of models that more unfolded (i.e. contain more duplicates) than the original one (note that behavioral precision is less than behavioral recall). This situation is improved after the mined models are pruned. In fact, the models for the noise types *swap tasks* and *mix all* become too folded (behavioral recall less than behavioral precision) after the arc post-pruning.

Figure 6.9: Results for the same metrics as in Figure 6.6, but the values are calculated based on a noise-free log of a12. Except for the noise types *swap tasks* and *mix all*, the quality of the mined models improved after they have undergone arc post-pruning. However, these pruned models are still less precise then the ones returned by the GA (cf. Figure 6.4).

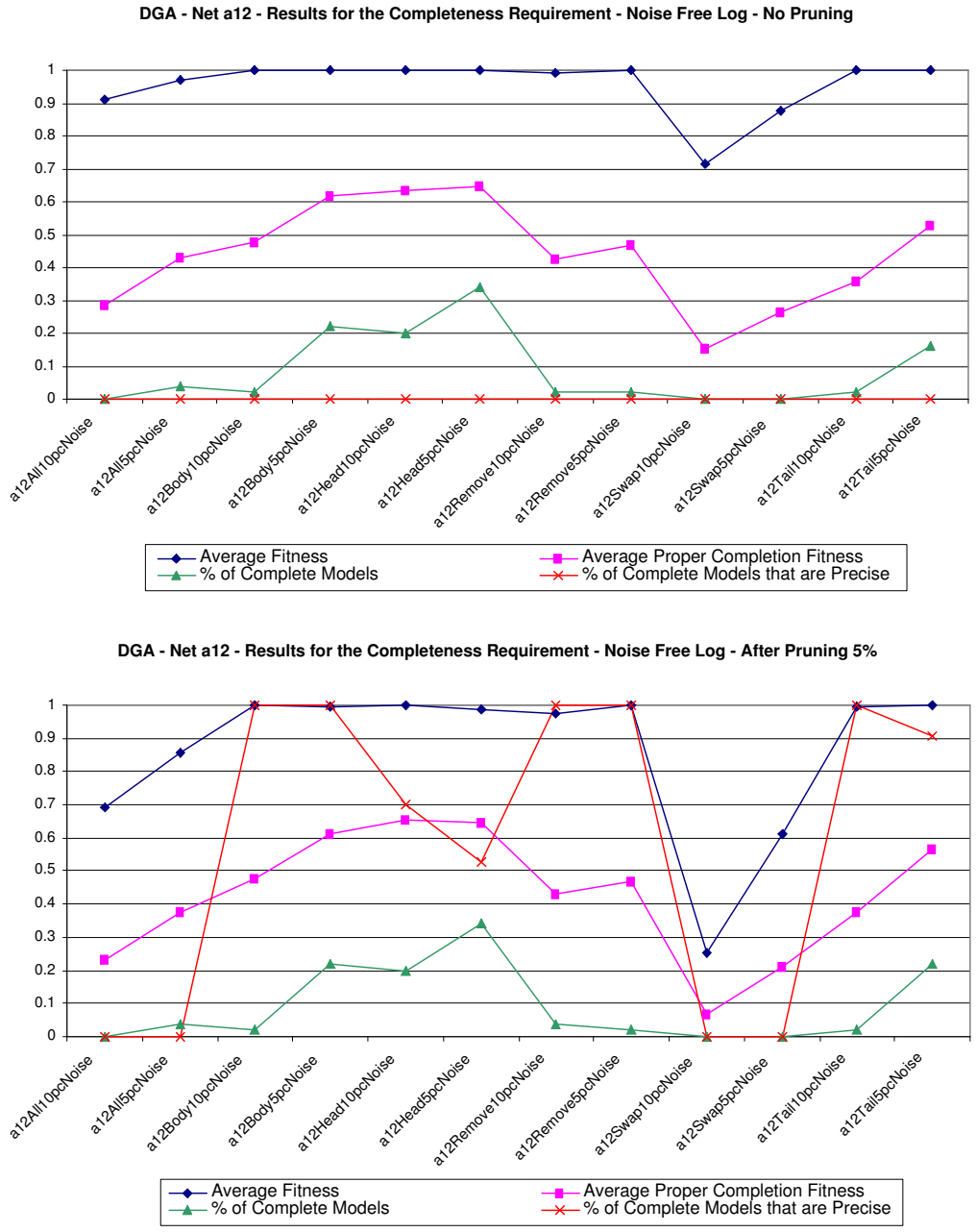


Figure 6.10: Same metrics as in Figure 6.7, but the values are computed based on a noise-free log of a12. Although the average proper completion is inferior to 0.5 in many cases, the complete models mined by the DGA are also precise in many situations. For instance, see the results for a12Remove10pcNoise and a12Remove5pcNoise.

Figure 6.11: Same remarks as for Figure 6.8.

one, and (ii) the fitness measures of both the GA and DGA are designed to prioritize the mining of models that correctly portrait the *most frequent* behavior in the log. Thus, when low-frequent behavior is also captured in a mined model, the end user has the flexibility to keep it or remove it by performing arc post-pruning.

During the experiments, 6 different noise types were considered: *missing head*, *missing body*, *missing tail*, *swap tasks*, *remove task* and *mix all*. Additionally, the experiments were executed for both the GA and the DGA. The results show that the GA is much more robust to noise than the DGA. This is probably because the GA uses a smaller search space than the DGA. The results also show that both algorithms are more sensitive to the noise type *swap tasks* and, consequently, *mix all*. Finally, although the experiments are not exhaustive, they reinforce the point that the fitness guides the algorithms to individuals that are as complete and precise as possible.

The next chapter gives an overview about the plug-ins that were implemented in the ProM tool to support the GA, DGA, the analysis metrics and the post-pruning step that was explained in this chapter.

Chapter 7

Implementation

This chapter describes how we have implemented the concepts that were explained in chapters 4 to 6. Rather than constituting a manual, this chapter aims at providing information about the plug-ins that have been developed. The information presented should be sufficient to enable the reader to mine his/her own logs or to repeat some of the experiments that we have described in previous chapters. All the algorithms mentioned before, i.e. basic Genetic Algorithm (GA), Duplicates Genetic Algorithm (DGA), arc pruning, and analysis metrics were developed as plug-ins in the ProM (**Pro**cess **M**ining) framework [32]. ProM is an open-source framework that supports the development of mining plug-ins. Thus, this chapter introduces this framework and the plug-ins that have been developed. The main idea is to make the reader familiar with the interfaces one encounters when playing with these plug-ins in the ProM framework. In addition to the plug-ins in the ProM framework, we have also developed some log conversion plug-ins in the ProM$_{import}$ framework [4, 46]. The converted logs were used during the experiments and the case study reported in this thesis. ProM$_{import}$ is also an open-source framework and it suports the development of plug-ins that convert logs from different systems (like Staffware, Flower, Eastman etc) to the MXML (**M**ining E**x**tensible **M**arkup **L**anguage) format that the ProM framework accepts. More details about the plug-ins explained in this chapter can be found at the help of the ProM or ProM$_{import}$ framework.

The remainder of this chapter is organized as follows. First, in Section 7.1, the ProM framework and the MXML format are introduced. Second, in sections 7.2 and 7.3, the mining plug-ins for the GA and the DGA are briefly described. Third, in Section 7.4, the plug-in to perform the arc post-pruning step is shown. Fourth, in Section 7.5, the analysis plug-ins and other plug-ins related to the experiments setup (e.g., the plug-ins to reduce the log size) are presented. Fifth, the ProM$_{import}$ plug-ins used to create the logs for the

experiments are explained in Section 7.6. Finally, Section 7.7 provides a short summary of this chapter.

7.1 ProM framework

ProM is an open-source framework [32] that is available at www.processmining.org. Since its creation, this framework has evolved from a tool to support the mining of process models from event logs, to a more general tool that also allows for the analysis of process models and conversions between process models in different notations [78]. Figure 7.1 shows a screenshot of the main interface of ProM. The menu *File* is often used as the starting point. Here, logs and models can be loaded into the framework and, consequently, used by the different plug-ins. The main concept behind the framework is that the plug-ins do not have to worry about loading the log files or other things that are not specific for a particular plug-in. As long as the log files are in MXML format, the ProM framework will handle them. Additionally, all algorithms are developed as plug-ins that specify its input (or accepted) objects and its output (or provided) objects. Based on the input/output of every plug-in, and the available objects at a certain moment in time, the framework automatically figures out which plug-ins to enable/disable. In other words, the collaboration between different plug-ins is automatically provided by the framework. The ProM framework is implemented in Java.

Figure 7.2 provides an overview of ProM's architecture. As illustrated, five types of plug-ins are supported: *mining*, *import*, *export*, *analysis* and *conversion* plug-ins. The *mining* plug-ins perform the actual mining based on logs in the MXML format (see Subsection 7.1.1 for more details about this format). The format of the output model depends on the mining plug-in. For instance, the *Alpha algorithm* plug-in provides Petri nets as output, while the *Multi-phase* plug-in also provides Event-driven Process Chains (EPC). In the context of this thesis, the GA and DGA are implemented as mining plug-ins that provide "Heuristic Nets" (HNs) as output. Causal matrices are called heuristic nets in ProM and are depicted graphically (i.e., a graph representation rather than a matrix representation). The *import* plug-ins allow for the loading of models. While importing, these models can be linked to an already opened log. The imported models can come from an external tool, like ARIS PPM or YAWL, or they may have been exported from ProM before. In our case, we have developed a plug-in to import and link causal matrices (or heuristic nets) to logs. The *export* plug-ins allow to save the results obtained during mining and analysis, as well as to perform operations over the logs and export the modifications. While describing the experimen-

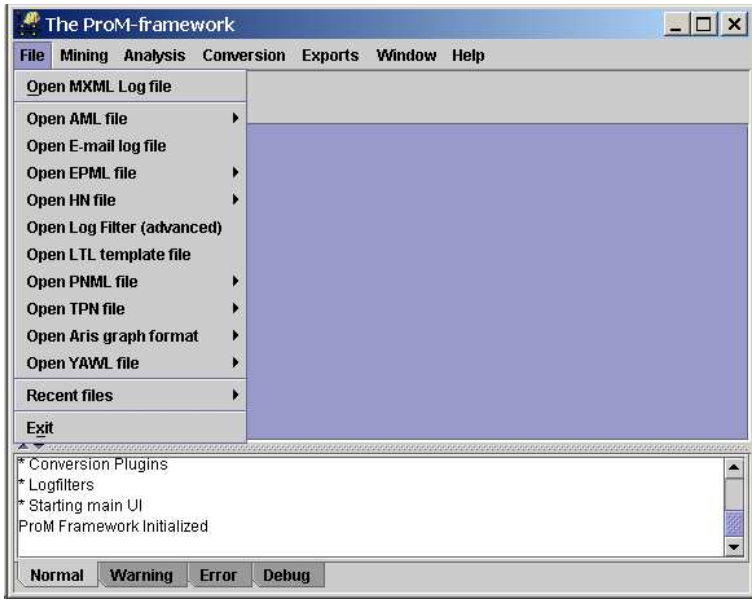


Figure 7.1: Screenshot of the main interface of the ProM framework.

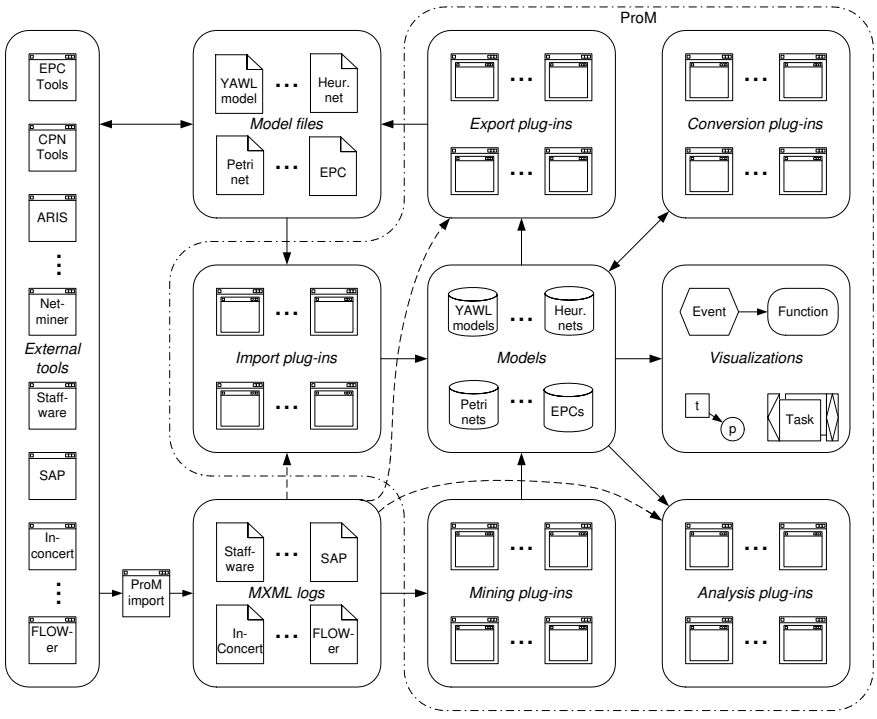


Figure 7.2: Overview of the architecture of the ProM framework.

tal setup in chapters 4, 5 and 6, we mentioned that we performed grouping of traces in the log to reduce its physical size. Grouping is implemented as an export plug-in in the ProM framework. The *analysis* plug-ins support the analysis of models and logs. For instance, if the model is a Petri net, it can be analyzed using place/transition invariants or state-space analysis. If the log should satisfy a certain property, e.g. a property like the four-eyes principle, the *LTL* (Linear Temporal Logic) plug-in can be used to check for this. Furthermore, one can replay a log for a certain (imported) model by using the *Conformance Checker* plug-in. In our context, both the analysis metrics and the arc post-pruning step are implemented as analysis plug-ins. The *conversion* plug-ins allow to convert between different models. In our case, we have implemented a plug-in to convert heuristic nets to Petri nets. An overview of all of the plug-ins supported by the ProM framework can be found at the help of this tool. In total there are 90 plug-ins [1]: 15 mining plug-ins, 10 import plug-ins, 22 export plug-ins, 10 conversion plug-ins, 24 analysis plug-ins and 9 log filters. Here we only describe the plug-ins that have been developed in the context of this thesis and mention some of the other plug-ins to give an impression of the features that are already supported by ProM and its plug-ins. More information about the ProM framework is available at www.processmining.org.

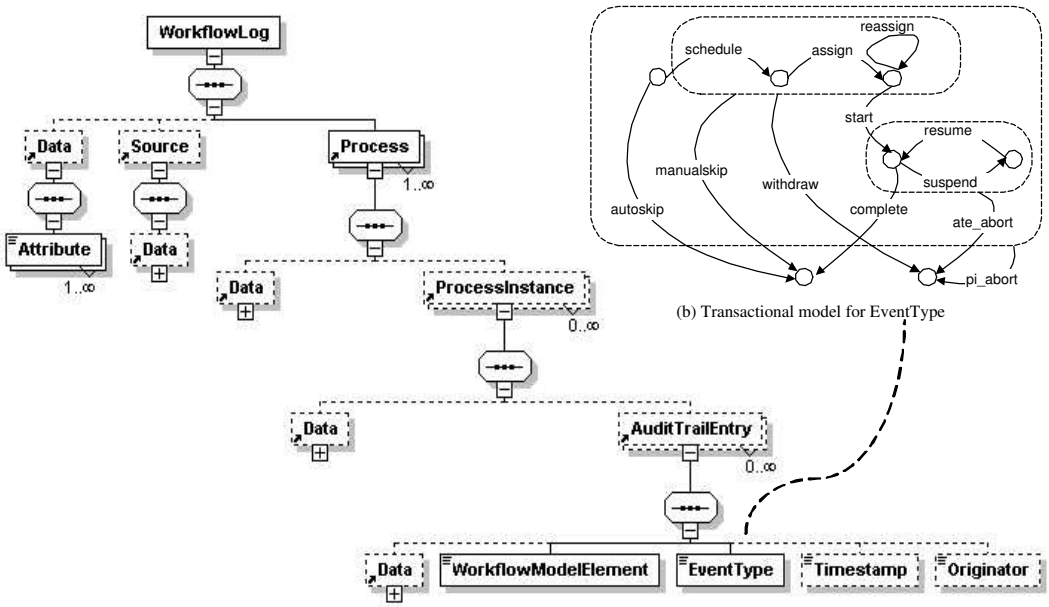
7.1.1 Mining XML format

The Mining XML[2] format (MXML) started as an initiative to share a common input format among different mining tools [14]. This way, event logs could be shared among different mining tools. The schema for the MXML format (depicted in Figure 7.3) is available at http://www.processmining.org/-WorkflowLog.xsd.

As can be seen in Figure 7.3, an event log (element *WorkflowLog*) contains the execution of one or more processes (element *Process*), and optional information about the source program that generated the log (element *Source*) and additional data elements (element *Data*). Every process (element *Process*) has zero or more cases or process instances (element *ProcessInstance)*. Similarly, every process instance has zero or more tasks (element *AuditTrailEntry*). Every task or audit trail entry (ATE) should at least have a name (element *WorkflowModelElement*) and an event type (element EventType). The event type determines the state of the tasks. There are 13 supported event types: schedule, assign, reassign, start, resume, sus-

[1] This information regards the version 3.1 of ProM.

[2] More information about the Extensible Markup Language (XML) can be found in [3].



(a) Process log XML format

(b) Transactional model for EventType

Figure 7.3: The visual description of the schema for the Mining XML (MXML) format.

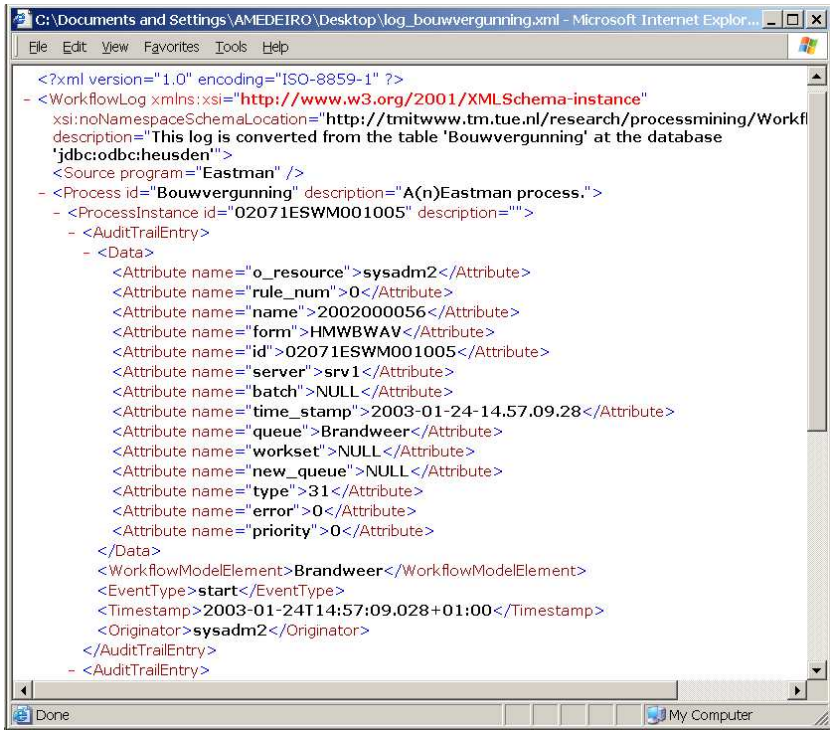


Figure 7.4: Excerpt of a log in the MXML format.

pend, autoskip, manualskip, withdraw, complete, ate_abort, pi_abort and unknown. The other task elements are optional. The *Timestamp* element supports the logging of time for the task. The *Originator* element records the person/system that performed the task. The *Data* element allows for the logging of additional information. Figure 7.4 shows an excerpt of a log in the MXML format. More details about the MXML format can be found in [32].

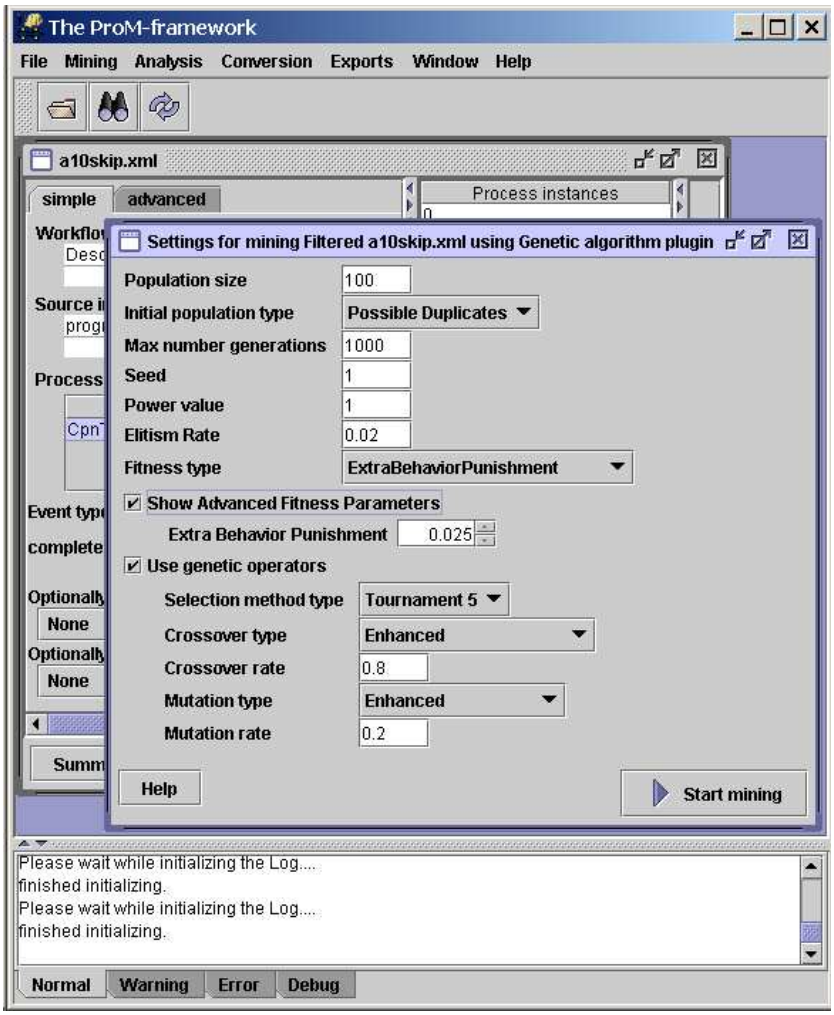


Figure 7.5: Main interface of the GA plug-in.

7.2 Genetic Algorithm Plug-in

The *Genetic algorithm* mining plug-in implements the GA explained in Chapter 4. The basic GA is suitable to mine process models that do not contain duplicate tasks. The GA plug-in receives as input an event log. Thus, before the reader can use this plug-in in ProM, the reader should first open a log. The interface of the GA plug-in is shown in Figure 7.5. A complete explanation about all of its configuration parameters can be found at the help of this plug-in. The configuration shown in Figure 7.5 is the same one used for one run of Scenario IV in Section 4.5.2. Once the configuration is set, a click on the button "Start mining" initiates the GA mining process. When the GA plug-in stops executing (either because it reached its stop criteria or because the user pressed the button "Cancel"), the last generated population is returned. Figure 7.6 shows the returned population after running the GA plug-in for 152 iterations[3]. As can be seen, the individuals are *decreasingly* ordered according to their fitness values. In other words, the best mined individual is returned first, and so on. Individuals can be selected by double-clicking one of the "Population" entries. Additionally, a returned individual (or heuristic net) can be visualized with or without showing the semantics of its split/join points (see checkbox "Display split/join semantics").

Although all the configuration parameters are explained in the help of the GA plug-in, we highlight here a few of them. The first one refers to the fitness types. In Chapter 4, we have explained the fitness F (cf. Definition 23) that is called "ExtraBehaviorPunishment" in the GA plug-in. However, the GA plug-in also supports four other experimental fitness types: "ProperCompletion", "StopSemantics", "ContinuousSemantics", and "ImprovedContinuousSemantics". These fitness types were created while searching for a good fitness measures to the basic GA. Our experience shows that the fitness measure F is the best one. However the user may be interested in, for instance, experimenting with fitness types that have a stop semantics parsing[4] (like the "ProperCompletion" and "StopSemantics"). Additionally, we also support other types of selection methods, crossover types and mutation types. Again, these other types were created while calibrating the basic GA. As an example, we have a crossover type ("Local One Point") that always swaps all input and output sets of the crossover point. Note that this crossover type is more coarse grained than the crossover type explained in Subsection 4.3.1 (which works at the subsets of the input/output of the crossover point). Again, the help of the GA plug-in in the ProM framework explains

[3]It starts at generation 0.

[4]In this stop semantics, the parsing of a trace (or process instance) stops whenever a task should be parsed but its corresponding activity (in the individual) is not enabled.

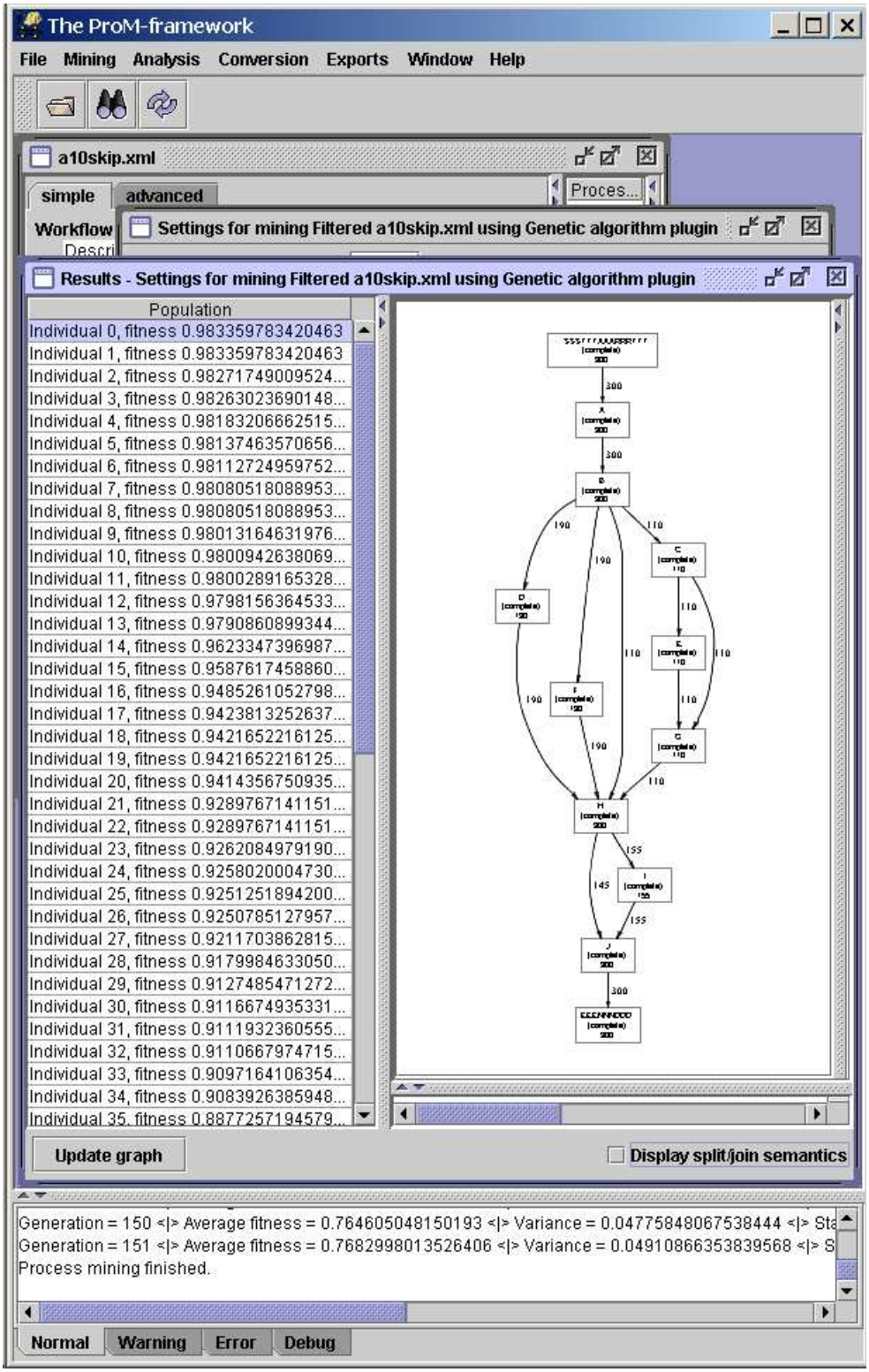


Figure 7.6: Interface for the results of one run of the GA plug-in.

all the (extra) supported types.

7.3 Duplicates Genetic Algorithm Plug-in

The *Duplicate Tasks GA* plug-in implements the DGA described in Chapter 5. The DGA can mine process models in which the duplicates of a task do not share input/output elements. In other words, the DGA can mine models in which the duplicates can be locally detected. The interface of the DGA plug-in is shown in Figure 7.7. As can be seen, this interface is very similar to the interface of the GA plug-in (cf. Section 7.2). Actually, the differences reside in the supported "Initial population type", "Fitness type" and "Crossover Type". These differences reflect the extensions made to the basic GA while defining the DGA. A complete explanation about all of the DGA's configuration parameters can be found at the help of this plug-in. The configuration shown in Figure 7.5 is the same one used for one run of the experiments described in Section 5.5.2.

Similar to the GA plug-in, the DGA plug-in also supports different variants for the initial population, fitness and crossover. These different variants are all explained in the help of the DGA plug-in. However, here we highlight *an initial population type that may reduce the search space of the DGA*. In Chapter 5, we explained that the maximum amount of duplicates that any individual in the population can have is based on the *follows* relations that can be extracted from the log (cf. Section 5.4.1). This is indeed the heuristic used by the initial population type "Follows Heuristics (Duplicates+Arcs)". However, sometimes a heuristics based on the *causal* relations (cf. relation "$\rightarrow_L$" in Definition 14) maybe be sufficient to set the maximum amount of duplicates per task. The principle is the same as for the heuristics based on the follows relation, but using the causal relation instead of the follows relation. To illustrate the search space reduction, consider the log in Table 5.1. A heuristics based on the *follows* relation sets that every individual has at most two duplicates of the tasks "Travel by Car", "Travel by Train" and "Conference Starts". The other tasks do not have duplicates. On the other hand, a heuristics based on the *causal* relation sets the same, but the task "Conference Starts" is also not duplicated. So, the DGA does not "waste time" searching for individuals with two duplicates for task "Conference Starts". In this case, building an initial population based on the causal relation would suffice because the target model (see Figure 5.1) indeed does not have more than one task "Conference Starts". Thus, for the situation in which the user assumes that the causal relations may be sufficient to detect the maximum amount of duplicates per task, the initial population type "Causal Heuristics

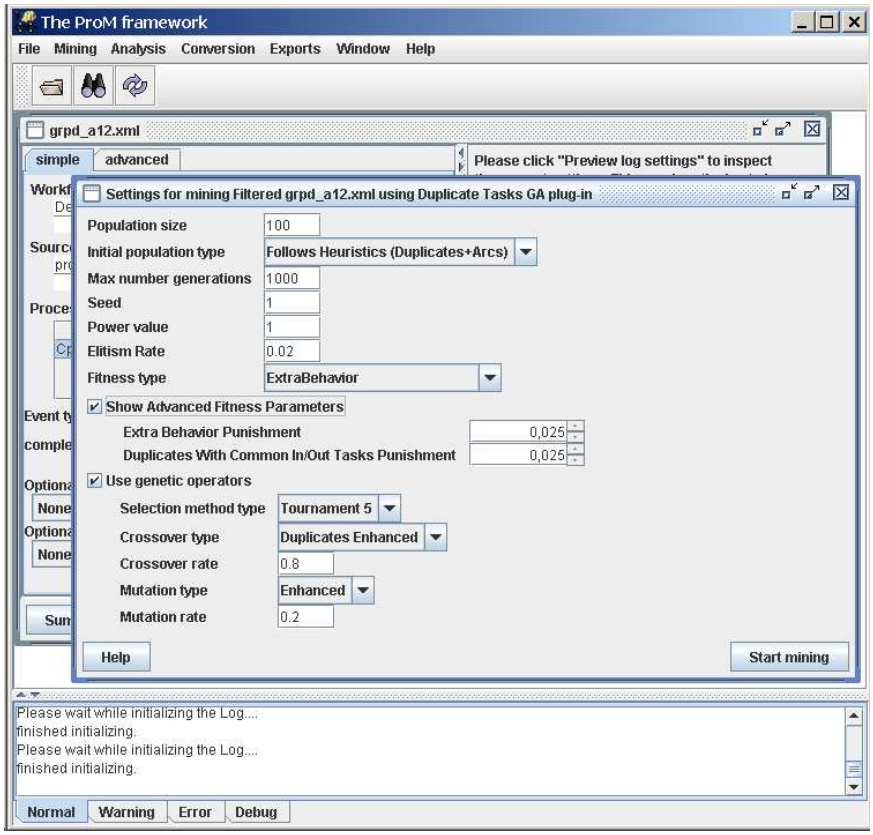


Figure 7.7: Main Interface of the DGA plug-in.

(Duplicates+Arcs)" is also provided in the DGA plug-in.

7.4 Arc Pruning plug-in

The post-processing step that prunes arcs from a model is implemented as the analysis plug-in "Prune Arcs". Figure 7.8 shows a screenshot of this plug-in in action. The left-side model is the unpruned one. The right-side model is the result of pruning arcs that are used 5% or fewer times than the most used arc. In this case, all arcs that are used 15 or fewer times were removed from the model because the highest arc usage is 300 (see arc between "ArtificialStartTask2" and "ArtificialStartTask"). Note that the *threshold* explained in Chapter 6 correspond to the parameter "Arc pruning in %". Furthermore, the plug-in takes the fitness type into account during the pruning. So, if the reader use the fitness "ProperCompletion" with threshold of 0%, only the arcs that were used to replay traces that proper complete are

kept.

7.5 Other plug-ins

Additionally to the GA, DGA and arc pruning plug-ins, we have implemented a set of auxiliary plug-ins to set up and analyse the experiments. The plug-ins regarding log preparation are explained in Subsection 7.5.1. The plug-in to convert mined individuals (heuristic nets) to Petri nets is described in Subsection 7.5.2. The plug-ins for the analysis metrics are described in Subsection 7.5.3.

7.5.1 Log Related Plug-ins

There are five log related plug-ins: *Add Artificial Start Task*, *Add Artificial End Task*, *Add Noise*, *Group Log (same sequence)* and *Group Log (same follows relation)*. The first three plug-ins are *log filters* that change the traces in the log. These plug-ins are illustrated in Figure 7.9. The last two plug-ins are *export* plug-ins that reduce the physical log size based on some similarity criterion. These plug-ins are illustrated in Figure 7.10.

The *Add Artificial Start Task* log filter allows to include a dummy task at the *start* of all log traces. In a similar way, the *Add Artificial End Task* log filter allows to include a dummy task at the *end* of log traces. These plug-ins have been developed because the implementations of the GA and DGA algorithms assume that the target model has a single start task and a single end task. Furthermore, these single start and end tasks must be XOR-join/split tasks. Thus, by adding two start tasks and two end tasks at every trace of an external log, we make sure that these constraints are satisfied. Note that these constraints are only implementation specific and do not affect the obtained results. The *Add Noise* log filter is used to include noise in a log. The plug-in provides an interface to select among noise types, the seed to be used during the noise inclusion, and the noise percentage.

The *Group Log (same sequence)* and *Group Log (same follows relation)* export plug-ins are used to reduce the physical size of the log. Recall that both the GA and DGA work by replaying the log traces in every unique individual in the population. Besides, these mining algorithms only take into account the sequence of the task in a trace, i.e. they do not consider who performed the task (originator element in MXML), or the data attributes of a task. Thus, to save on computational time, we have developed these two export plug-ins. The main idea is that they group the traces in a log based on some similarity criterion and register in the exported log the frequency

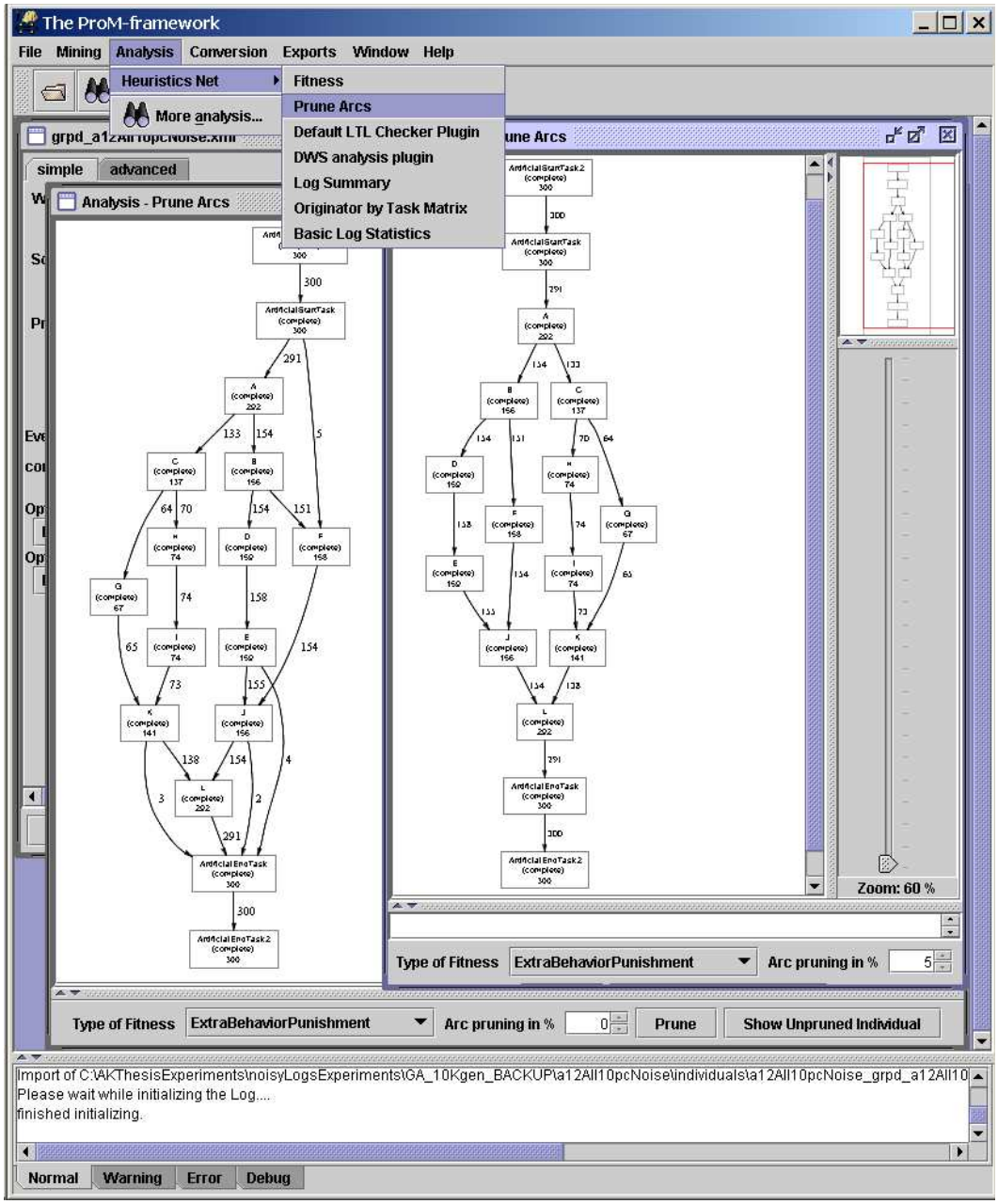


Figure 7.8: Arc pruning plug-in.

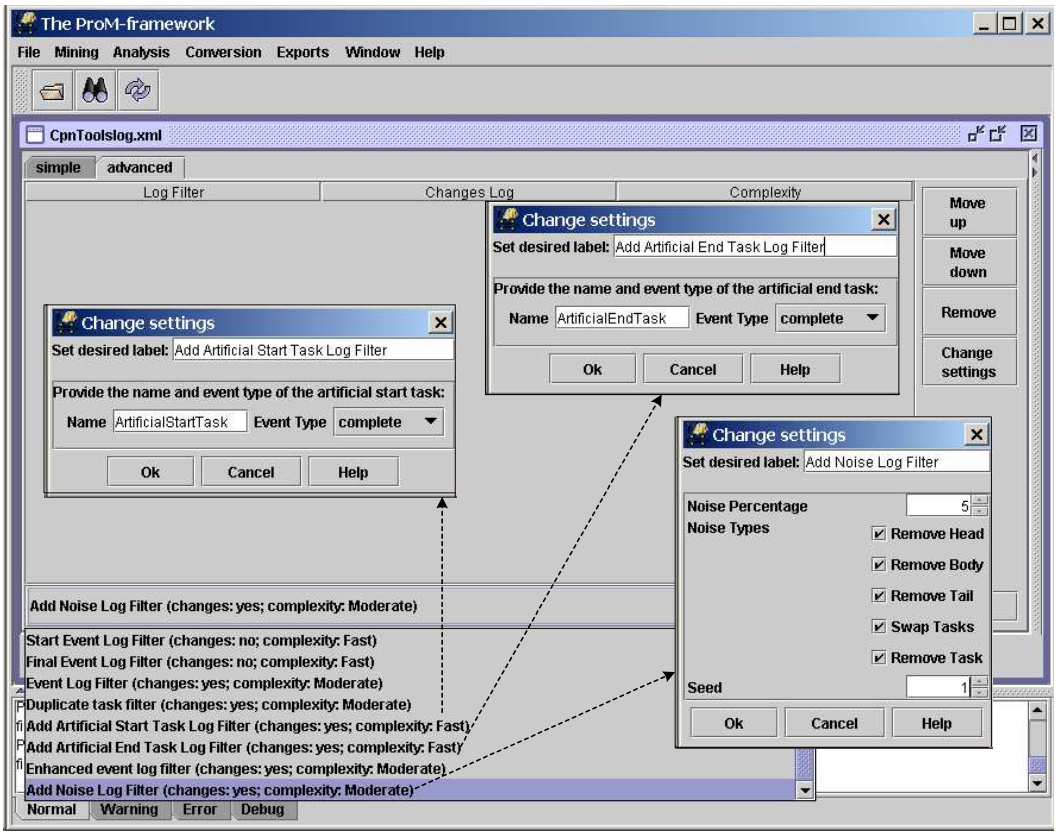


Figure 7.9: Log filter plug-ins: *Add Artificial Start Task*, *Add Artificial End Task* and *Add Noise*.

Figure 7.10: Export plug-ins to reduce the log size: *Group Log (same sequence)* and *Group Log (same follows relation)*.

that the grouped traces happen in the original log. The frequency is stored as a *Data* attribute of a *ProcessInstance* element in the MXML format (cf. Subsection 7.1.1). The name of this data attribute is "numSimilarInstances". Figure 7.11 shows the excerpt of a log in which traces are grouped. The *Group Log (same sequence)* export plug-in groups the traces that have the *same sequence of tasks with respect to the MXML elements WorkflowModelElement and EventType*. The *Group Log (same follows relation)* export plug-in groups the traces that have the *same set of follows relation with respect to the MXML elements WorkflowModelElement and EventType*, keeping the longest trace at the exported log. For instance, assume that a log contains the four traces (i) "$A_{complete}$, $B_{complete}$, $C_{complete}$, $B_{complete}$, $C_{complete}$, $D_{complete}$", (ii) "$A_{complete}$, $B_{complete}$, $C_{complete}$, $B_{complete}$, $C_{complete}$, $B_{complete}$, $C_{complete}$, $B_{complete}$, $C_{complete}$, $D_{complete}$", (iii) "$A_{complete}$, $F_{complete}$, $G_{complete}$, $H_{complete}$, $D_{complete}$" and (iv) "$A_{complete}$, $B_{complete}$, $C_{complete}$, $B_{complete}$, $C_{complete}$, $D_{complete}$". If this log is exported using the plug-in *Group Log (same sequence)*, the traces (i) and (iv) are going to be grouped because they express the same sequence of events. So, the resulting log would contain 3 unique traces and 4 traces in total (since one of the unique traces - "$A_{complete}$, $B_{complete}$, $C_{complete}$, $B_{complete}$, $C_{complete}$, $D_{complete}$" - has a counter 2 associated to it). However, if this same log is exported using the plug-in *Group Log (same follows relation)*, the traces (i), (ii) and (iv) are going to be grouped because the same set of follows relations can be inferred from these traces. In this case, the resulting log would contain 2 unique traces and 4 traces in total.

7.5.2 Model Related Plug-ins

Three model related plug-ins have been implemented: *Open HN File*, *Export HN File* and *Heuristic net to Petri net*. The first two plug-ins respectively provide the loading and saving of heuristic nets (or causal matrices). Thus, for instance, the actual mining and the analysis of the models can be performed at different moments in time. Figure 7.1 shows the menu option for the *Open HN File* import plug-in. The *HN File* export plug-in is illustrated in Figure 7.12. The *Heuristic net to Petri net* is a conversion plug-in. It maps heuristic nets to Petri nets. This way mined models can benefit from the other plug-ins in ProM that receive as input Petri nets. The *Heuristic net to Petri net* conversion plug-in is illustrated in Figure 7.13. This figure shows the menu option, a mined heuristic net (at the top right) and the mapped Petri net for this heuristic net (at the bottom).

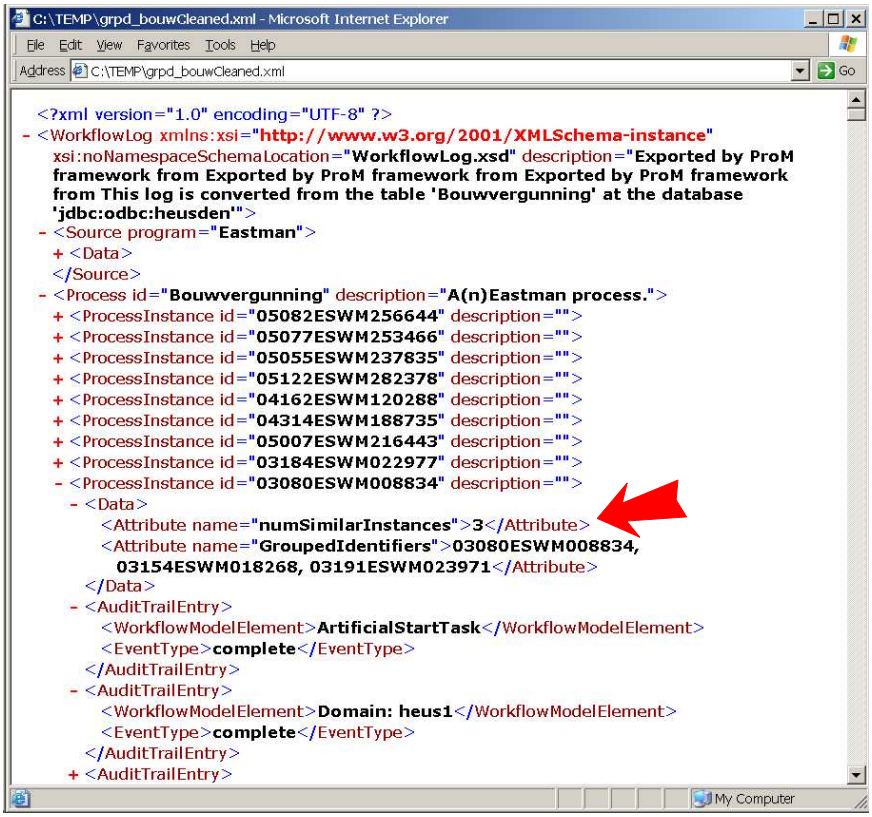


Figure 7.11: Excerpt of a grouped log. The indicated process instance results from the grouping of 3 process instances in the original log.

7.5.3 Analysis Related Plug-ins

The analysis metrics described in chapters 4 and 5 are implemented as analysis plug-ins in ProM. The metric for the *completeness requirement* is available as the analysis plug-in *Fitness*. This plug-in receives as input a heuristic net already linked to a log. The interface of the *Fitness* analysis plug-in (see Figure 7.14) provides the selection of different fitness types. The partial fitness $PF_{complete}$ (see Definition 21) corresponds to the fitness type *ImprovedContinuousSemantics*. Figure 7.14 shows the results of applying the fitness *ProperCompletion* to a heuristic net. The fitness proper completion gives the percentage of log traces that can be parsed by a heuristic net without leading to missing tokens or without leaving tokens. The heuristic net in Figure 7.14 can correctly parse 90% of log traces to which it is linked. The *Behavioral, Structural and Duplicates Precision/Recall* metrics are illustrated in Figure 7.15 (see bottom left). All these metrics have an interface to provide the original model (the "Base net") and the mined model (the "Mined

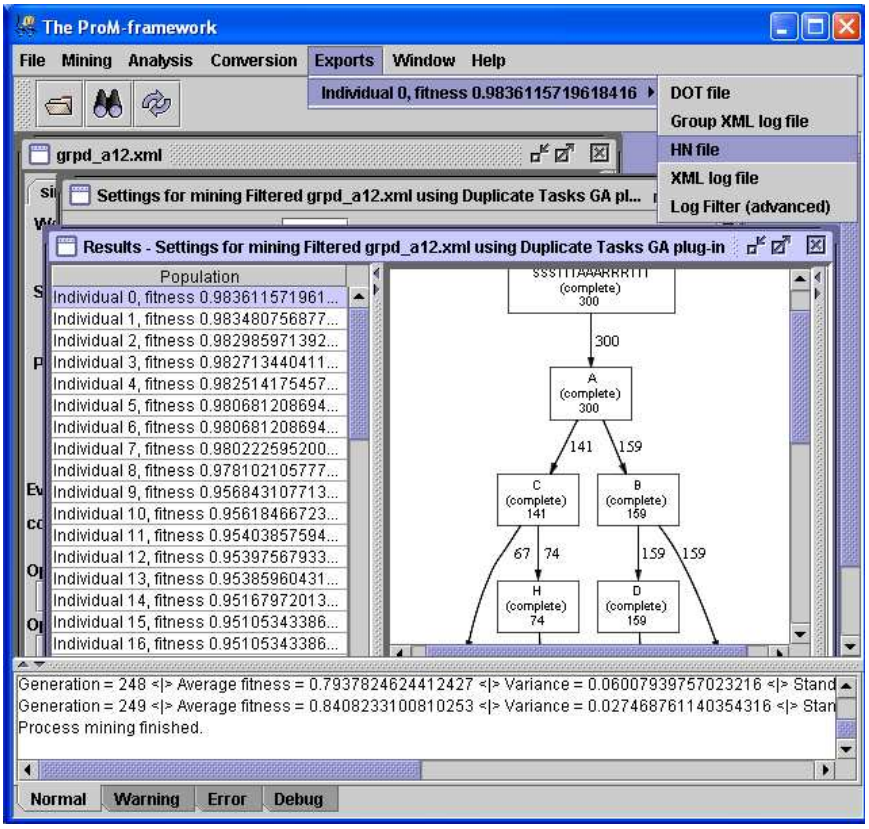


Figure 7.12: Export plug-in: *HN File*.

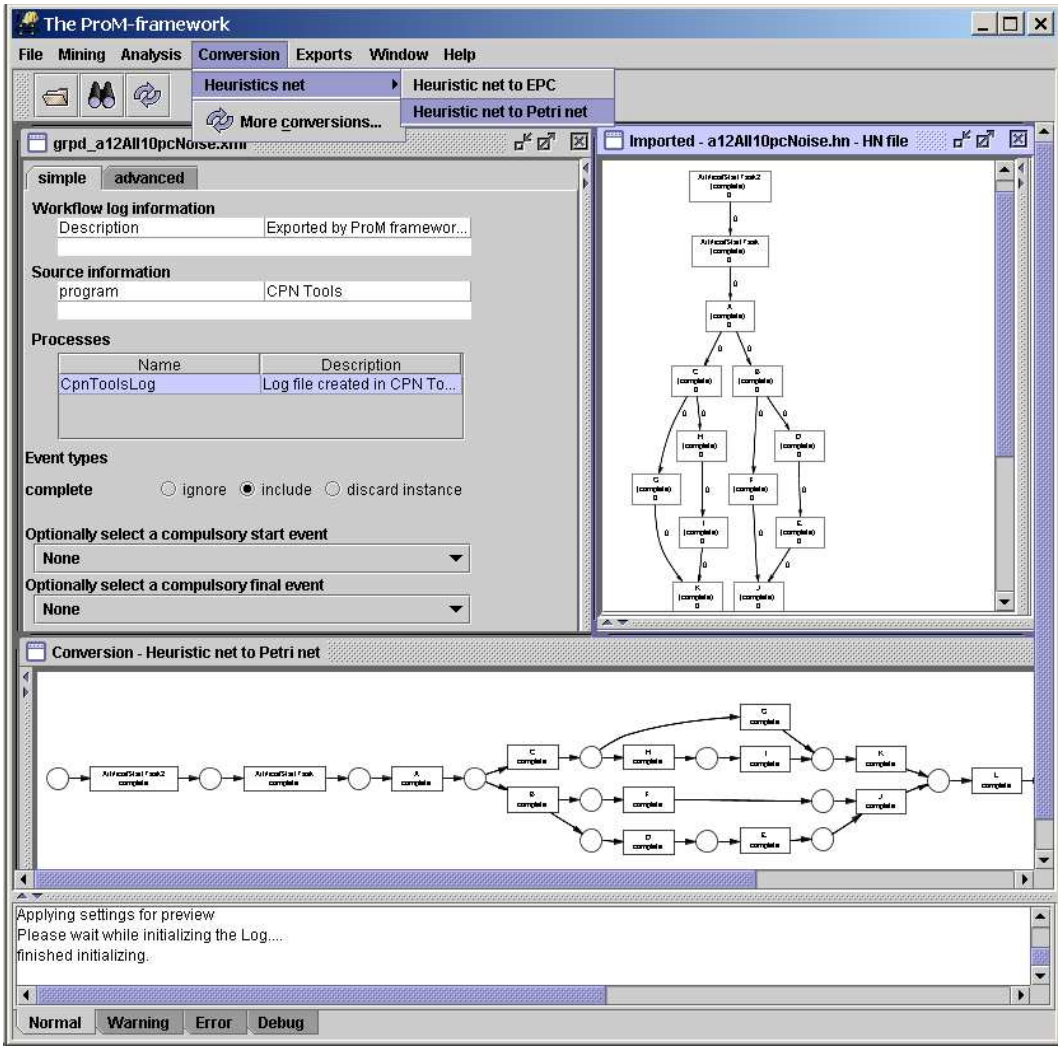


Figure 7.13: Conversion plug-in: *Heuristic net to Petri net*.

net"). Additionally, the behavioral precision/recall metrics require the two selected nets to the linked to the same log. Figure 7.14 illustrates the behavioral precision (Bp = 0.729) and the behavioral recall (Br = 0.988) for two individuals.

7.6 ProM$_{import}$ Plug-ins

ProM$_{import}$ [4, 46] is an open-source framework that makes it easier to convert event logs from different formats to the MXML format that ProM can read. In this section we explain the two ProM$_{import}$ plug-ins that we have implemented to generate the logs used in our experiments. The two ProM$_{import}$ plug-ins are: *CPN Tools* and *Eastman*. The *CPN Tools* ProM$_{import}$ plug-in was used to generate the noise-free logs mentioned in chapters 4 and 5, as well as the logs utilized during the single-blind experiments (see Section 8.2). The *Eastman* plug-in was used to convert the logs for the case study explained in Section 8.3. All these plug-ins can be downloaded together with the ProM$_{import}$ framework at www.processmining.org. The following subsections provide more details about these plug-ins.

7.6.1 CPN Tools

CPN Tools is a software package that supports the modelling, execution and analysis of Coloured Petri nets (CP-nets) [55, 56]. Additionally, there is a fair amount of CPN models that can be used as input to test mining algorithms. Thus, we decided to extend CPN Tools to support the creation of MXML logs. The main idea is to create random MXML logs by simulating CP-nets in CPN Tools. The extension is fairly simple. The first part of the extension consisted of implementing the ML functions to support the logging from a CP-net. The second part consisted of implementing the *CPN Tools* plug-in in the ProM$_{import}$ framework to bundle the logged files into a single MXML file. In short, two steps are necessary to create MXML logs using CPN Tools:

1. Modify a CP-net by invoking a set of ML functions that will create logs for every case executed by the CP-net. This step involves modifying the declarations of the CP-net and the input/output/action transition inscriptions.
2. Use the CPN Tools plug-in in the ProM$_{import}$ framework to group the logs for the individual cases into a single MXML log.

The synthetic logs used in our experiments were created by performing the two steps above. Figure 7.16 shows an annotated CPN net for the net choice.

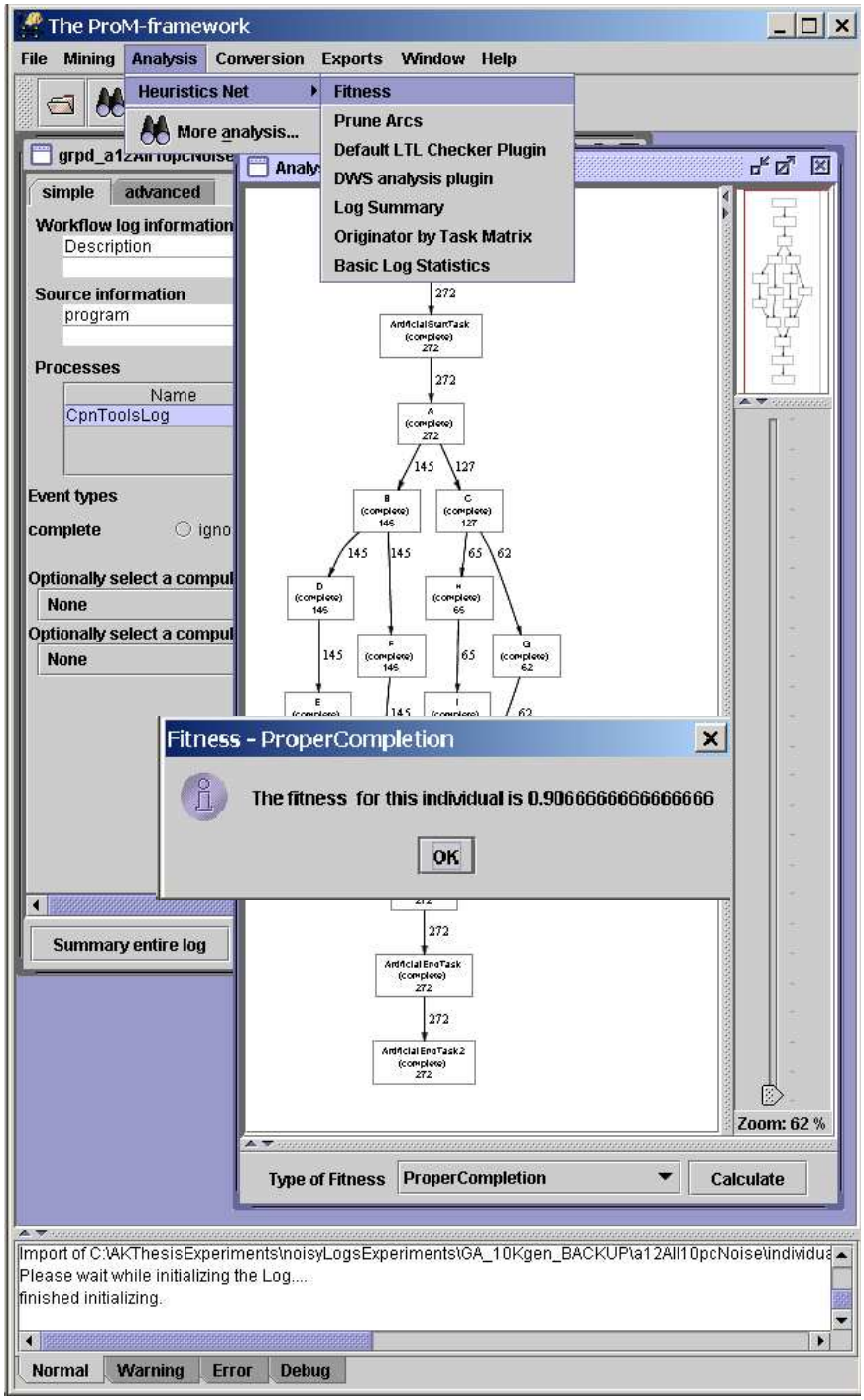


Figure 7.14: Analysis plug-in to check for the completeness requirement: *Fitness*.

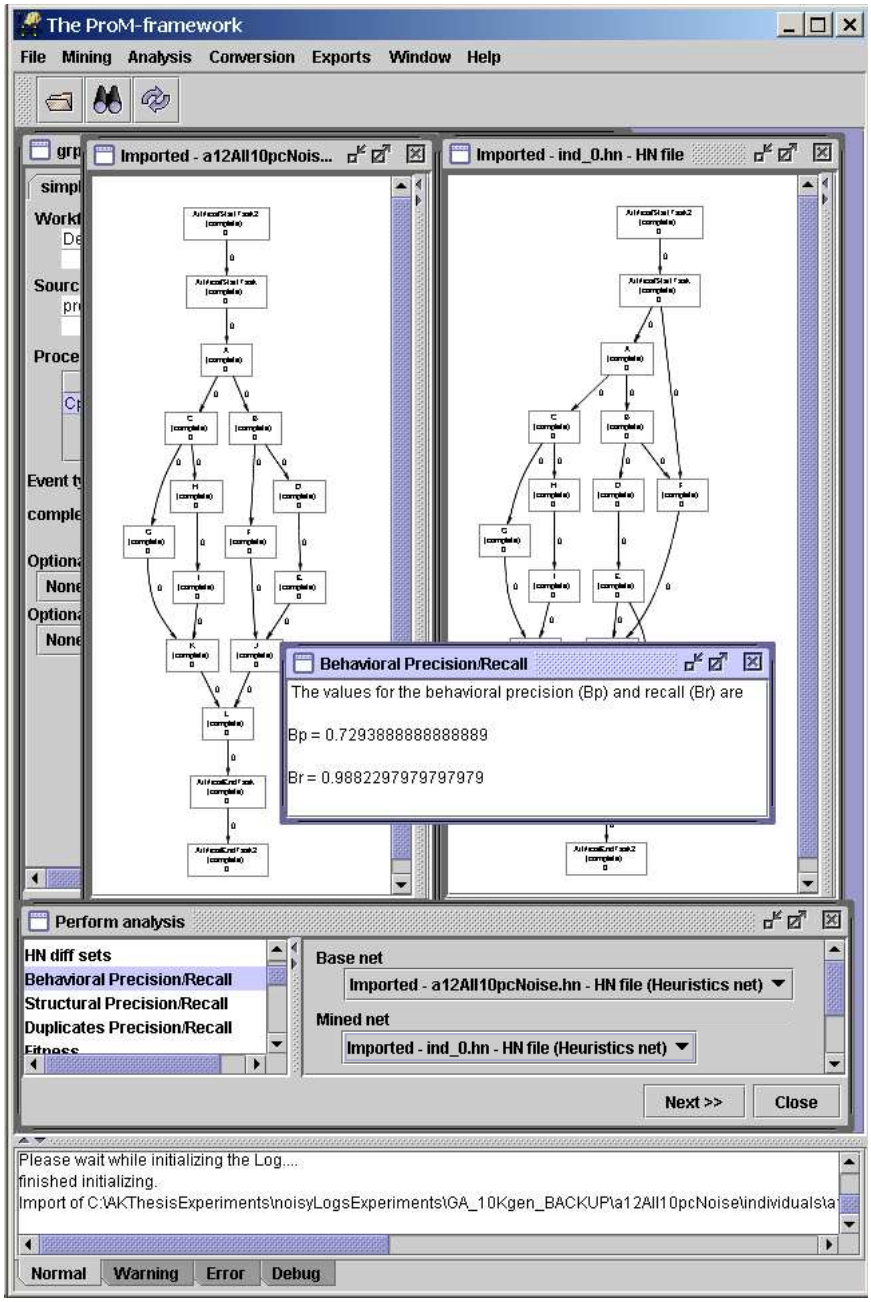


Figure 7.15: Analysis plug-in to check for the preciseness and folding requirements: *Behavioral, Structural* and *Duplicates Precision/Recall*.

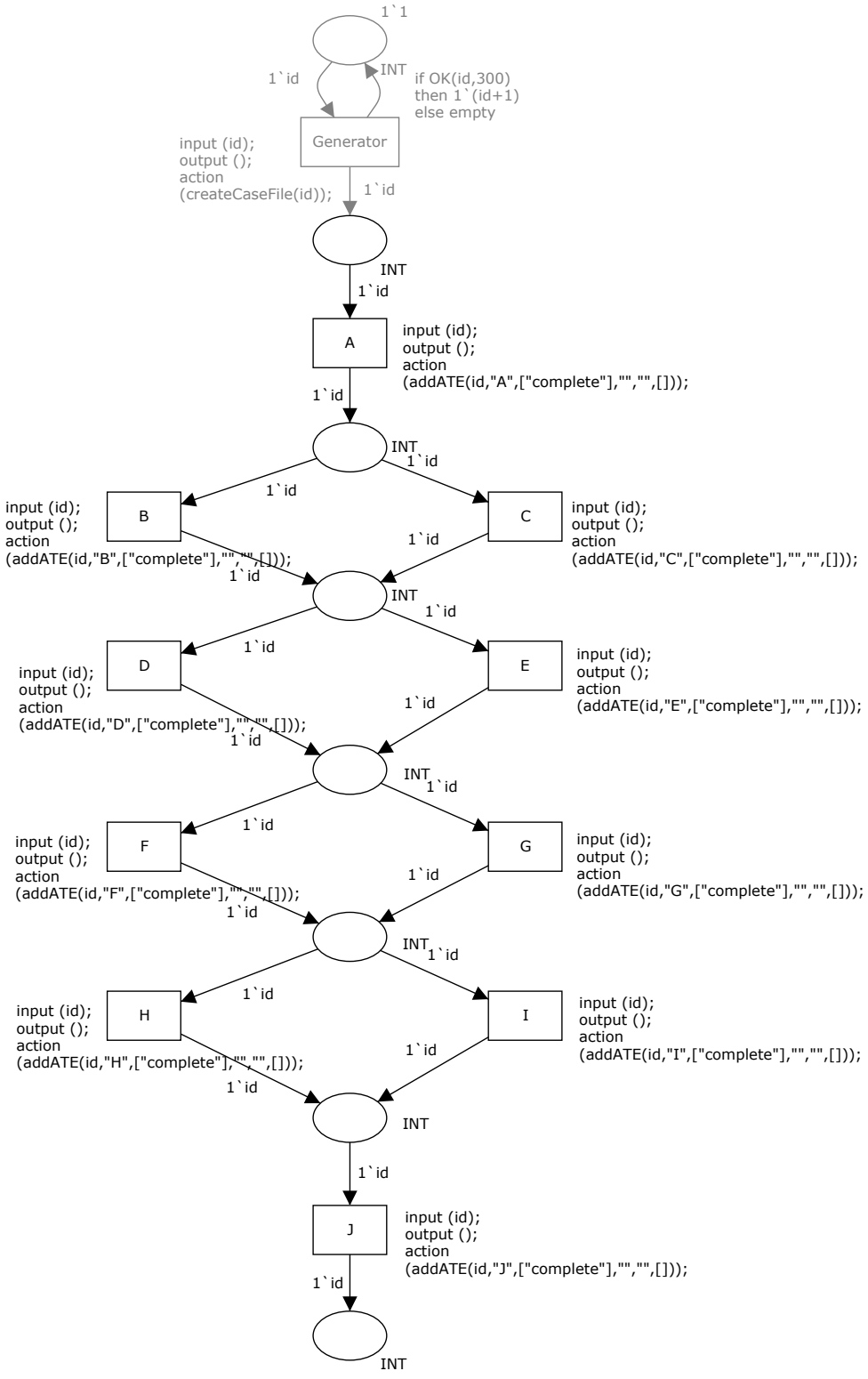


Figure 7.16: Annotated CP-net for net choice.

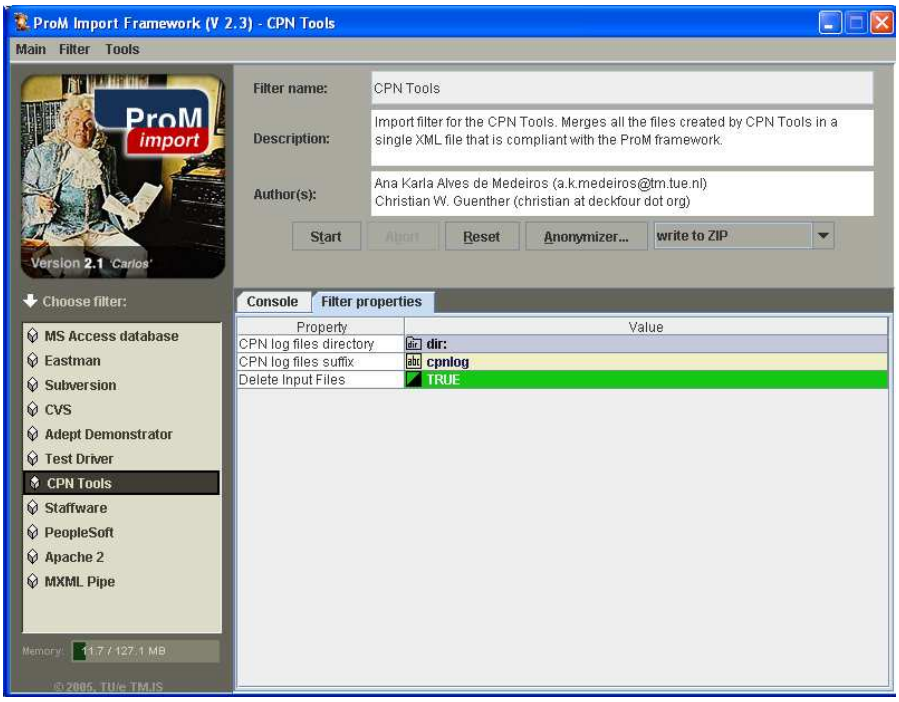


Figure 7.17: Prom$_{import}$ plug-in: *CPN Tools*.

The function *createCaseFile* creates a file for every case. The functions *addATE* logs the execution a given task (or Audit Trail Entry) to a case file. In Appendix B we show the annotated nets for all models used during the experiments. The *CPN Tools* ProM$_{import}$ plug-in is illustrated in Figure 7.17. It receives as input the directory to which the logs for every case were written ("CPN log files directory"), the termination of these files ("CPN log files suffix") and if the case files should be deleted after the aggregation ("Delete input files"). The output is a MXML file that contains all the cases. More details about how to use CPN Tools to generate synthetic logs can be found in [27] and in "ProM CPN library" tutorial provided at www.processmining.org.

7.6.2 Eastman

Eastman[5] [74] is the workflow system that created the logs used during the case study (cf. Section 8.3). The cases are logged in an MS-Access database.

[5]The Eastman workflow system is nowadays called *Global 360* (www.global360.com). However, we have decided to keep the original name of this workflow system in the ProM$_{import}$ plug-in.

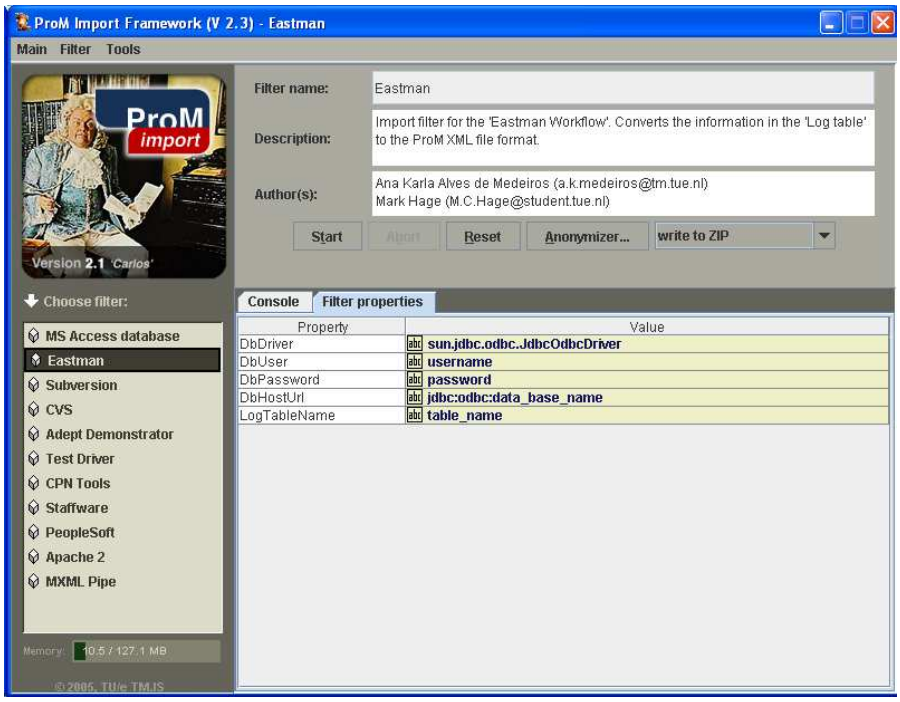


Figure 7.18: Prom$_{import}$ plug-in: *Eastman*.

The *Eastman* ProM$_{import}$ plug-in is illustrated in Figure 7.18. It receives as input the database ODBC driver ("DbDriver"), the database username ("DbUser") and password ("DbPassword"), the database URL ("DbHostUrl") and the name of the table with the logging information to be converted ("LogTableName"). The output is also a MXML log file.

7.7 Summary

This chapter introduced the plug-ins we have developed to realize the algorithms and analysis metrics explained in chapters 4 to 6. In total, 17 plug-ins were implemented: 15 in the ProM framework and 2 in the ProM$_{import}$ framework. However, both ProM and ProM$_{import}$ offer a much wider variety of plug-ins.

As an illustration, the ProM framework has a plug-in that allows for the verification of EPCs [36]. Thus, before deploying a model (or showing it for the managers), the reader can verify its correctness. The models can be imported in the format *EPML* (EPC Markup Language), *AML* (ARIS Markup Language) or the ARIS PPM graph format [59] and verified with the plug-in "Check correctness of EPC". Furthermore, the ProM framework provides

other diagnosis tools based on logs and a variety of models. The analysis plug-in "LTL Checker" [10] uses linear temporal logic to verify properties in the log. For instance, for a log of an ATM machine, the reader can check if a task to ask and check for a password is always executed before a task that provides the money. The LTL formulae can refer to any of the elements defined in the MXML format (including the data attributes). Another useful plug-ins are the "Conformance Checker" [68] and the "Performance Analysis with Petri Nets". The "Conformance Checker" replays the log in the model and identifies the points of discrepancies, the most frequent paths, etc. The "Performance Analysis with Petri Nets" also works based on a model and a log. This plug-in calculates basic management information about the process like the average completion time of processes, the waiting time between tasks etc. In total, the ProM framework offers more than 90 plug-ins. Some plug-ins work based on Petri nets, other on EPCs or Heuristic nets, but this is not a problem since the ProM tool provides conversion plug-ins from one format to another.

The ProM$_{import}$ framework also provides additional plug-ins (more than 15 in total!). For instance, the *Flower* plug-in, suitable to convert logs from the case-handling tool Flower [19, 20], and the *Staffware* plug-in, which converts logs from the workflow system Staffware [75].

Both ProM and ProM$_{import}$ are open-source tools developed in Java. They are all available at www.processmining.org. We encourage the reader to try their plug-ins and to collaborate with us.

In next chapter we present additional evaluation of our approach. We describe blind-experiments and a case study.

Chapter 8

Evaluation

This chapter describes the evaluation of our genetic mining approach. The genetic mining approach presented in this thesis was evaluated using three kinds of logs: *synthetic logs from known models*, *synthetic logs from unknown models* and *real-life logs*. Because both the setup and the results for the experiments with known models were already given in chapters 4 to 6, here we only explain why these models were selected and how their synthetic logs were created. This is done in Section 8.1. The experiments with *synthetic logs from unknown models* are reported in Section 8.2. The results for the *case study* (experiments with *real-life logs*) are given in Section 8.3. Finally, Section 8.4 provides a short summary of this chapter.

8.1 Experiments with Known Models

The models used to test our approach should contain the constructs used to review the related work (cf. Chapter 2). These constructs are: sequences, choices, parallelism, short loops, structured/arbritary loops and non-local non-free-choice, invisible tasks and duplicate tasks. Additionally, the models should have different numbers of tasks. In total, we have modelled 39 nets. Table 8.1 gives an overview of the constructs present in each net. The nets themselves are in Appendix B. Note that the 18 nets whose names start with "herbst..."[1] were directly copied from Herbst's PhD thesis [50]. We have selected these nets for two reasons: (i) Herbst's approach can also handle duplicate tasks, so it is interesting to get an idea about how well our approach performs on these models, and (ii) *these nets have not been defined by us.*

[1]The name indicates the net's respective figure number in Herbst's PhD thesis. For instance, the net in "Figure 3.4" in his thesis has the name "herbstFig3p4".

Net	Figure	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Structured Loop	Arbitrary Loop	Non-Local NFC	Invisible Tasks	Duplicates in Sequence	Duplicates in Parallel
a10skip	B.1	✓	✓	✓						✓		
a12	B.3	✓	✓	✓								
a5	B.5	✓	✓	✓	✓							
a6nfc	B.7	✓	✓	✓						✓		
a7	B.9	✓	✓	✓								
a8	B.11	✓	✓	✓								
al1	B.13	✓	✓	✓				✓				
al2	B.15	✓	✓	✓				✓				
betaSimplified	B.17	✓	✓						✓	✓	✓	
bn1	B.19	✓	✓									
bn2	B.21	✓	✓				✓			✓		
bn3	B.23	✓	✓				✓			✓		
choice	B.25	✓	✓									
driversLicense	B.27	✓	✓						✓			
flightCar	B.29	✓	✓	✓								✓
herbstFig3p4	B.31	✓	✓	✓			✓					
herbstFig5p1AND	B.33	✓		✓							✓	
herbstFig5p1OR	B.35	✓	✓								✓	
herbstFig5p19	B.37	✓	✓	✓						✓	✓	
herbstFig6p10	B.39	✓	✓	✓			✓			✓	✓	
herbstFig6p18	B.41	✓	✓		✓	✓	✓			✓		
herbstFig6p25	B.43	✓	✓			✓	✓			✓	✓	
herbstFig6p31	B.45	✓	✓								✓	
herbstFig6p33	B.47	✓	✓								✓	
herbstFig6p34	B.49	✓	✓	✓			✓			✓	✓	
herbstFig6p36	B.51	✓	✓						✓			
herbstFig6p37	B.53	✓		✓								
herbstFig6p38	B.55	✓		✓								✓
herbstFig6p39 nc	B.57	✓		✓						✓		✓
herbstFig6p41	B.59	✓	✓	✓								
herbstFig6p42 nc	B.61	✓	✓	✓						✓	✓	
herbstFig6p45	B.63	✓		✓								
herbstFig6p9	B.65	✓	✓							✓	✓	
l1l	B.67	✓	✓		✓							
l1lSkip	B.69	✓	✓	✓	✓					✓		
l2l	B.71	✓	✓			✓						
l2lOptional	B.73	✓	✓			✓						
l2lSkip	B.75	✓	✓			✓				✓		

Continued on next page

Net	Figure	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Structured Loop	Arbitrary Loop	Non-Local NFC	Invisible Tasks	Duplicates in Sequence	Duplicates in Parallel
parallel5	B.77	✓		✓								

Table 8.1: Nets for the experiments with known models.

From the 18 models copied form Herbst's thesis (cf. Table 8.1), 12 have duplicate tasks. From these, 2 (herbstFig6p39 and herbstFig6p42) are non-compliant (nc) with the kind of models our DGA aims at. This is the case because, for both models, some of the duplicates share input/output elements. For instance, consider the heuristic net representation of the net herbstFig6p39 in Figure 8.1. Note that the two duplicates of "A" share "join" as an output element. Yet we included both models in our experiments to check if the DGA would be able to mine also some non-compliant nets.

The remaining 21 nets were designed by us. The main motivation to create each of these nets was to check for:

- *Different kinds of short loops*: Tasks in a loop can be optional or required to be executed at least once. For instance, both the nets l1l and l1lSkip model length-one loop situations (cf. the respective figures 8.2 and 8.3) for the activities "B" and "C". However, l1lSkip requires the activities "B" and "C" to be executed at least once, while both activities can be completely skipped in l1l. In a similar way, the nets l2l, l2lOptional and l2lSkip model different types of length-two loop constructs.
- *Interleaving situations*: As seen in chapters 4 and 5, the fitness of our genetic approach benefits individuals that correctly parse the most frequent behavior in the log. In situations in which many interleaving between the tasks are possible, this characteristic of the fitness measure may generate side-effects to the mined model because it will always benefit the individuals that show the *most frequent* interleaving situations in the log. Thus, the nets a12, a8, choice and parallel5 were created.
- *Non-local NFC*: Herbst's models did not contain non-local NFC constructs. Actually, the net herbstFig6p36 uses duplicate tasks instead of

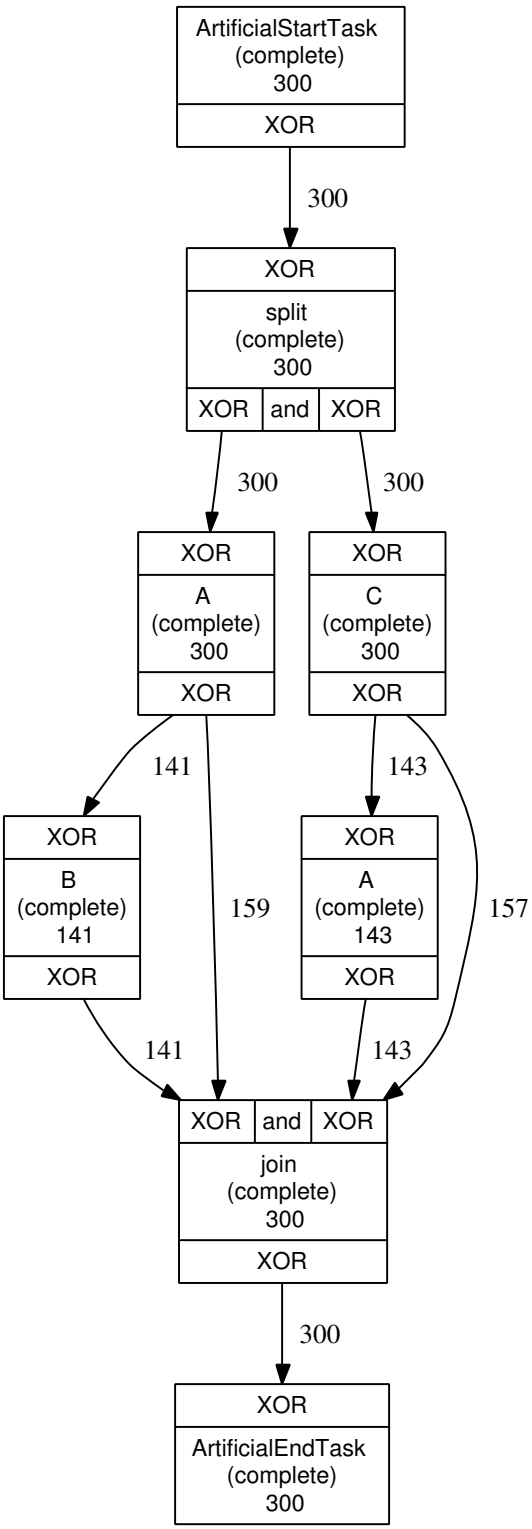


Figure 8.1: Heuristic net for herbstFig6p39. This model contains duplicate tasks that violate the assumption made in Chapter 5, i.e. both "A" tasks share a common "join".

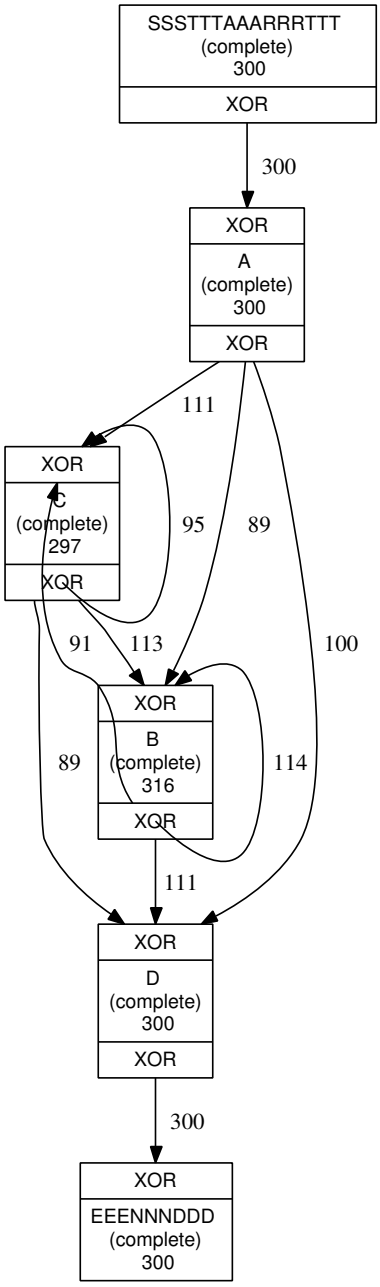


Figure 8.2: Heuristic net for l1l.

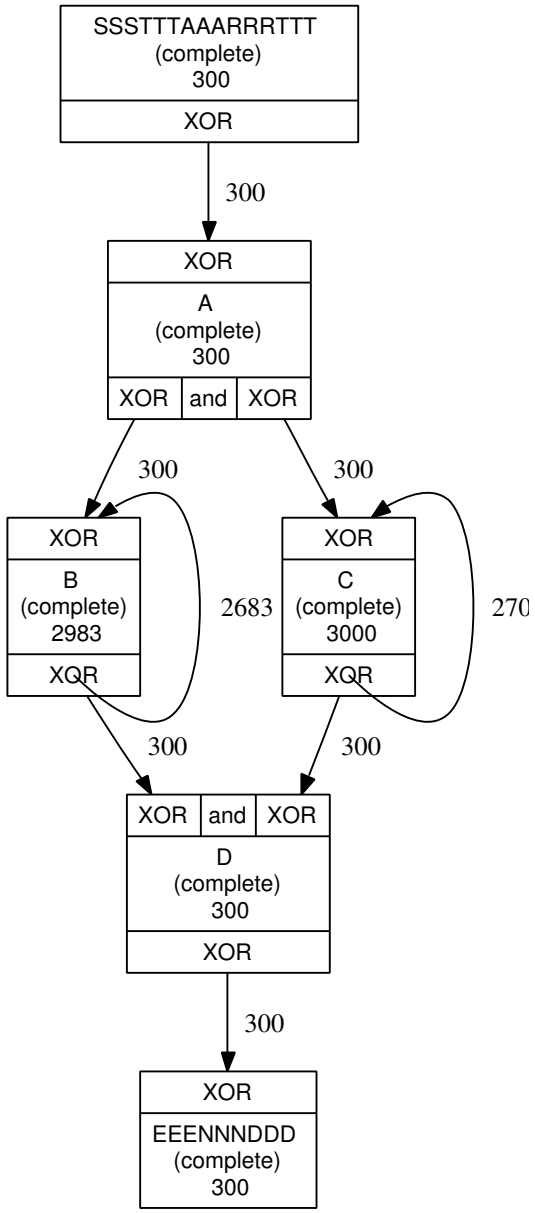


Figure 8.3: Heuristic net for l1lSkip.

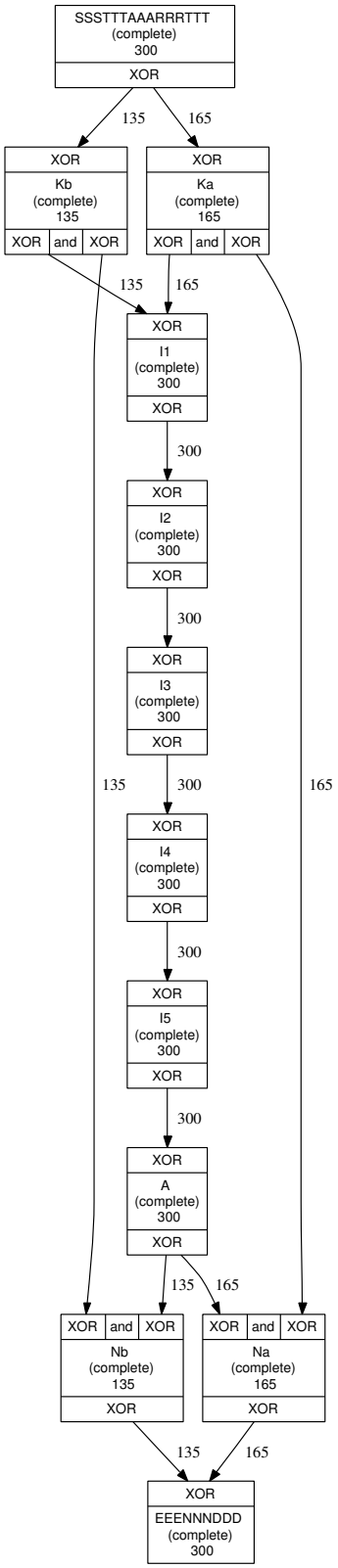


Figure 8.4: Heuristic net for herbstFig6p36.

non-local NFC. However, because our DGA does not capture this kind of nets (the duplicates share input/output elements), we have modelled it (cf. Figure 8.4) with a non-local NFC construct. Additionally, the nets driverLicense, betaSimplified and a6nfc also contain non-local NFC constructs. Note that the net betaSimplified also has duplicate tasks.

- *Non-block structured constructs*: All of Herbst's models are block structured. So, the nets a10Skip, a5, a7, al1 and al2 have been created. The first three nets have branches that intersect. The nets al1 and al2 contain arbitrary loops [13].
- *Different net sizes*: The original nets have between 7 and 44 tasks. We needed nets with different size to check how sensitive our genetic mining approach is to the size of the models. For instance, the nets for the short loops ("l1l..." and "l2l...") have no more than 10 tasks, while the nets "bn1..." have about 40 tasks.

As shown in Table 8.1, some of the mentioned nets would fit in more than one of the previous items. For instance, all nets with non-local non-free-choice are also non-block structured. However, we have opted for highlighting the most important reason why these models have been selected.

Once the nets have been chosen, they have been modelled in CPN Tools [2] to create the logs via simulation [27]. More details about this process are given in Subsection 7.6.1.

As a final remark about the models in Table 8.1, we emphasize that Herbst's approach can mine all models that start with "herbst...". The experiments in Chapter 5 show that the DGA can also mine many of these models. However, Herbst's approach is much faster (the mined models are returned in seconds or minutes [50], while the DGA takes minutes or hours). Besides, Herbst's approach, although block-structured, does not require the duplicates to have disjoint input/output elements. Thus, for such models, Herbst's approach seems to be more suitable. However, the advantage of our genetic approach over Herbst's one is that ours can mine also non-block-structured nets and it captures (non-local) non-free-choice.

More details about the setup and the results for the experiments with known models are provided in sections 4.5, 5.5 and 6.2.

8.2 Single-Blind Experiments

As explained in the previous section, most of the models without duplicates used to test the basic Genetic Algorithm (GA) (cf. Chapter 4) were created by us (cf. Table 8.1). Only five models (herbstFig3p4, herbstFig6p36, herb-

stFig6p37, herbstFig6p41 and herbstFig6p45) were not our design. Thus, to make sure that the models we have created were not too biased, we have conducted a *single-blind experiment* to test the GA even further. In a single-blind experiment, the original models are unknown while running the experiments and selecting a mined model for a given log. The aim of this experiment was to check how complete and precise the models returned by the GA were.

The original models were extracted from group assignments from students of the course "Business Process Modelling"[2]. The group assignment has two-phases. In the first phase, the students have to create workflow models from fictive situations that resemble real-life situations. In the second phase, the students have to apply redesign rules in [65] to improve the performance of their first-phase models. The redesigned models are usually closer to models that one typically encounters in real-life. Probably because they were more thoughtfully considered while applying the redesign rules. In total, there were 20 groups. Thus, a third party selected one redesigned model from each group and created one noise-free log for every model. Like for the experiments with the known models (cf. Section 8.1), the third party also used CPN Tools to simulate the selected models and, therefore, create the logs. In total, 20 logs were provided (one log for every selected model of a group). As for the experiments with the known models (cf. Subsection 4.5.2), every log had 300 traces.

Since our previous experience with the GA pointed out that the more the GA iterates, the better the results, we have opted for running the experiments with a smaller population size (only 10 individuals), but for more generations (5000 at most). Additionally, for each of the 20 logs, 10 runs were executed (i.e. 10 experiments per log). The configuration of the remaining parameters for each run was just like for Scenario IV in Subsection 4.5.2. Once the 10 runs for a log were finished, the best mined model was selected. This selected model can correctly parse more traces in the log than the other 9 mined models. In case of ties, one of the mined models was randomly chosen. After the mined models were selected, the original models were made public to us, so that we could proceed with the analysis of the mined models.

The original models have all constructs but non-free-choice and duplicate tasks. The tool[3] the students used to design these models does not support duplicates. Non-free-choice constructs are supported by this tool, but the selected models happened not to have this kind of construct. Table 8.2 shows the main constructs used in every model. Note that, although the models do

[2]This course is held at the Information Systems subdepartment. This subdepartment belongs to the Technology Management department of the Eindhoven University of Technology.

[3]The modelling tool Protos [63] is used for this course.

not have non-free-choice or duplicates, they do exhibit imbalances between the AND-split/join points (cf. last column in the Table 8.2).

Net	Figure	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Structured Loop	Arbitrary Loop	Invisible Tasks	Unbalanced AND-split/join
g2	C.1	✓	✓	✓		✓	✓		✓	
g3	C.2	✓	✓	✓			✓	✓	✓	
g4	C.4	✓	✓	✓		✓				✓
g5	C.6	✓	✓	✓				✓	✓	
g6	C.7	✓	✓	✓		✓			✓	
g7	C.8	✓	✓	✓			✓		✓	
g8	C.9	✓	✓	✓		✓	✓		✓	✓
g9	C.11	✓	✓	✓		✓	✓		✓	
g10	C.13	✓	✓	✓				✓	✓	
g12	C.15	✓	✓	✓		✓		✓	✓	
g13	C.16	✓	✓	✓	✓	✓			✓	✓
g14	C.18	✓	✓	✓			✓		✓	✓
g15	C.20	✓	✓		✓	✓			✓	
g19	C.22	✓	✓	✓		✓			✓	✓
g20	C.24	✓	✓		✓	✓		✓	✓	
g21	C.25	✓	✓					✓	✓	
g22	C.26	✓	✓	✓			✓		✓	✓
g23	C.28	✓	✓	✓		✓				✓
g24	C.30	✓	✓	✓				✓	✓	✓
g25	C.32	✓	✓	✓	✓				✓	

Table 8.2: Nets for the single-blind experiments.

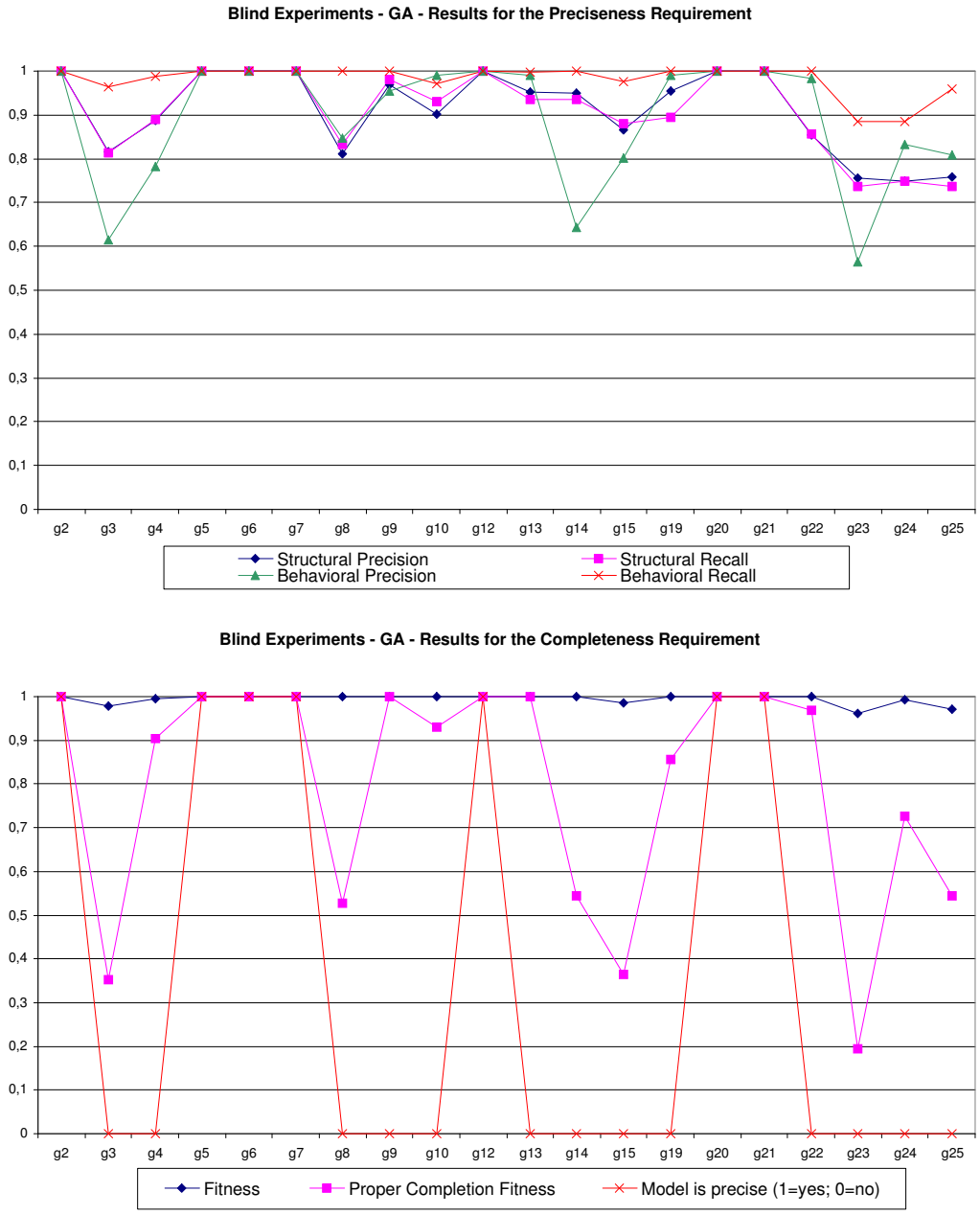


Figure 8.5: Analysis metrics for the mined models for the single-blind experiments.

Figure 8.5 shows the results for the analysis metrics. In short, the single-blind experiments revealed that the GA has problems to mine models whenever many interleaving situations between the tasks are possible. This result is not really surprising because of the nature of the GA's fitness measure. Recall that the fitness measure always benefits the individuals that better portrait the *most frequent behavior* in the log. From the 20 models in Table 8.2, the GA had more problems to mine models with: (i) parallel constructs with more than two branches, especially when these branches have more than two tasks each; (ii) unbalance of AND-split/join points (note that this also derives from item (i) before because it involves parallel branches as well); and more than three length-two loops connected to a same starting point. More specifically, the results in Figure 8.5 show that:

- 85% (17/20) of the mined models can correctly parse more than 50% of the traces in the log (cf. bottom graph in Figure 8.5, measure "Proper Completion Fitness"). Furthermore, 60% (12/20) of the mined models correctly parse at least 90% of the traces in the log, and 77% (7/9) of the mined models that are complete are also precise (see measure "Model is precise" on page 175). This supports the hypothesis that the fitness of the GA indeed guides the search towards individuals (or models) that correctly capture the most frequent behavior in the log.
- All the seven models that turned out to be precise were also structurally identical to the original models. Note that the models g2, g5, g6, g7, g12, g20 and g21 have all structural precision and recall values that are equal to 1 (cf. top graph in Figure 8.5).
- 60% (12/20) of the mined models (see top graph in Figure 8.5) have behavioral precision and recall metrics close to 1. This shows that, even when the models are not as precise as the original ones (like for g8, g9, g13, g19 and g22), they do not allow for much more extra behavior than the original one with respect to the log.
- The mined models for g3, g14 and g23 are less precise. As illustrated in the top graph in Figure 8.5, all of these models have behavioral precision values that are inferior to 0.7. The original and mined models for g3, g14 and g23 are respectively shown in figures 8.6 to 8.13. Based on these models, we can conclude that:

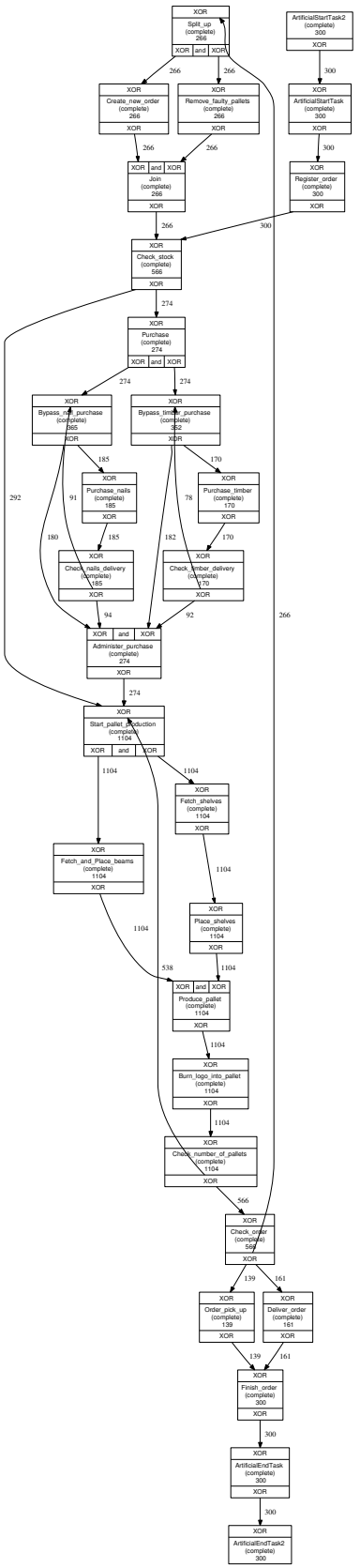


Figure 8.6: Original model for net g3. The parallel construct in which the two branches are length-three loops is difficult to mine.

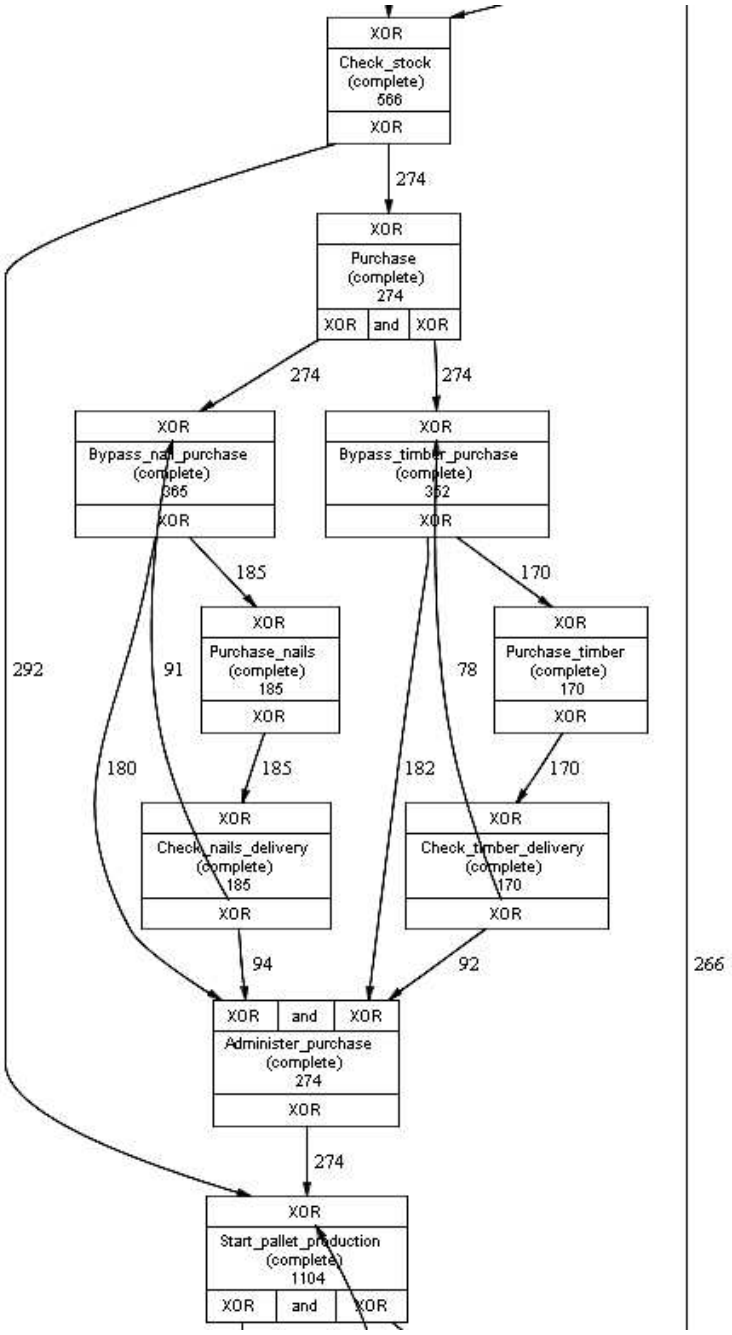


Figure 8.7: Zoomed view of the parallel construct with length-three loops in the original model for net g3. Note that this construct is not correctly captured in the mined model in Figure 8.8, but the remaining structure of this mined model is very similar to the original model in Figure 8.6, as it can be seen in the arc-pruned model in Figure 8.9. This model has been pruned so that only the arcs/activities that are used to replay the traces that proper complete are kept.

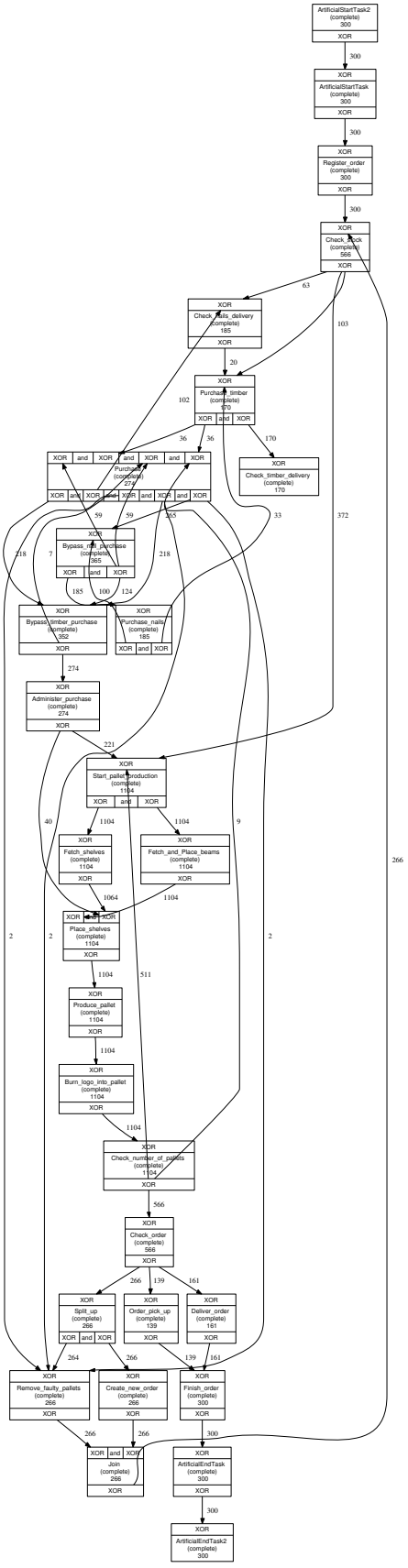


Figure 8.8: Mined model for net g3.

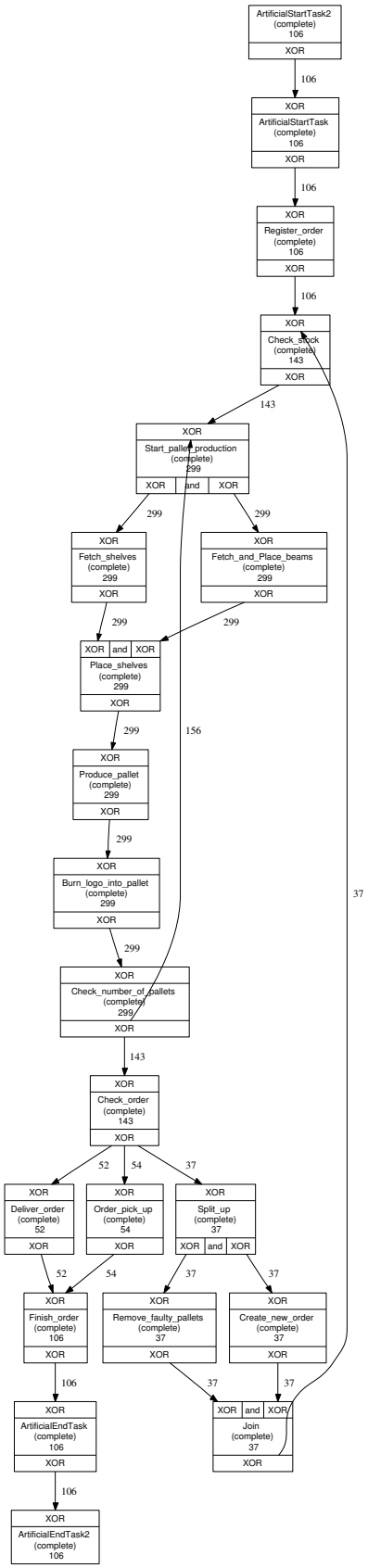


Figure 8.9: Mined model for net g3 after arc pruning.

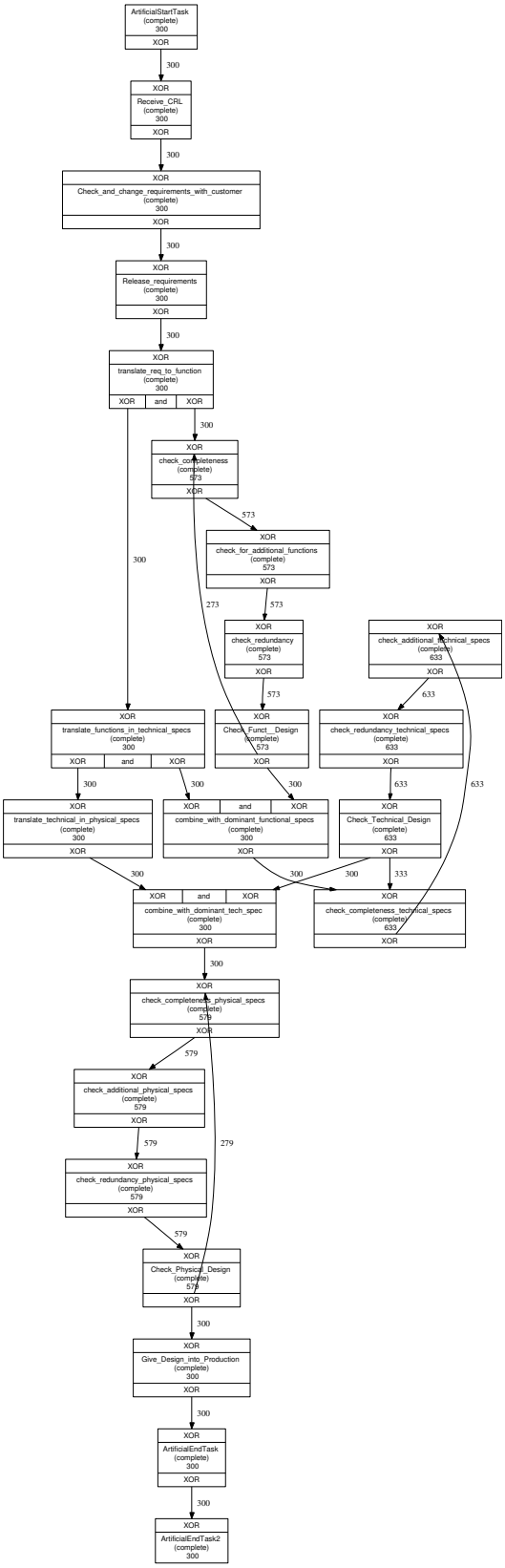


Figure 8.10: Original model for net **g14**. The unbalanced AND-split/join points are difficult to mine.

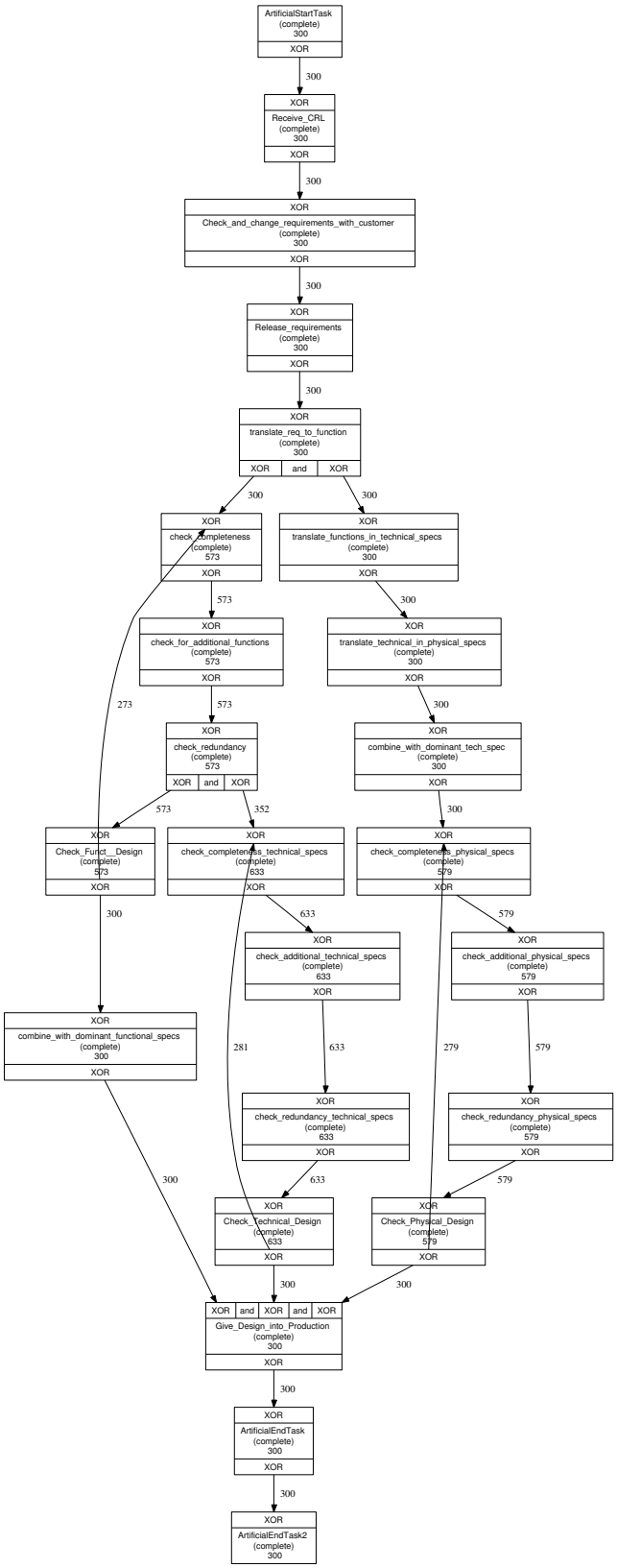


Figure 8.11: Mined model for net **g14**. Note that, in comparison with the original model in Figure 8.10, the loops are all correctly captured, but the AND-split/join points are not.

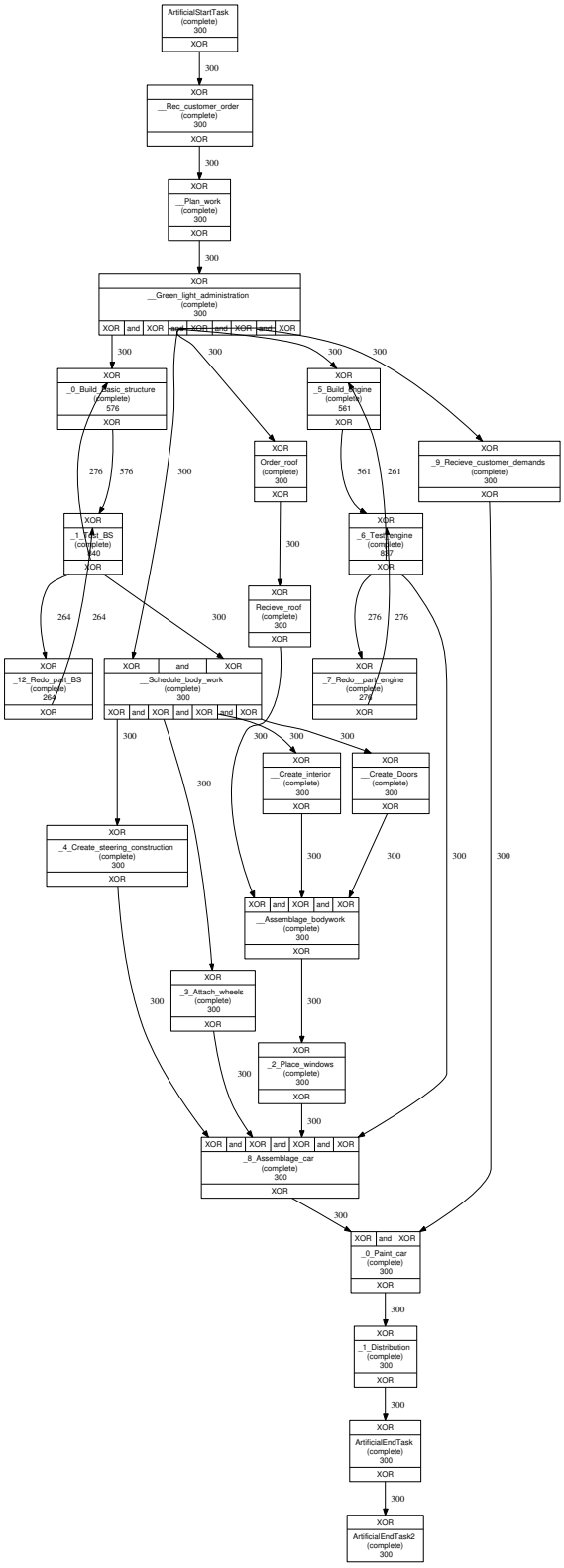


Figure 8.12: Original model for net g23. This model has four-parallel branches with unbalanced AND-split/join points.

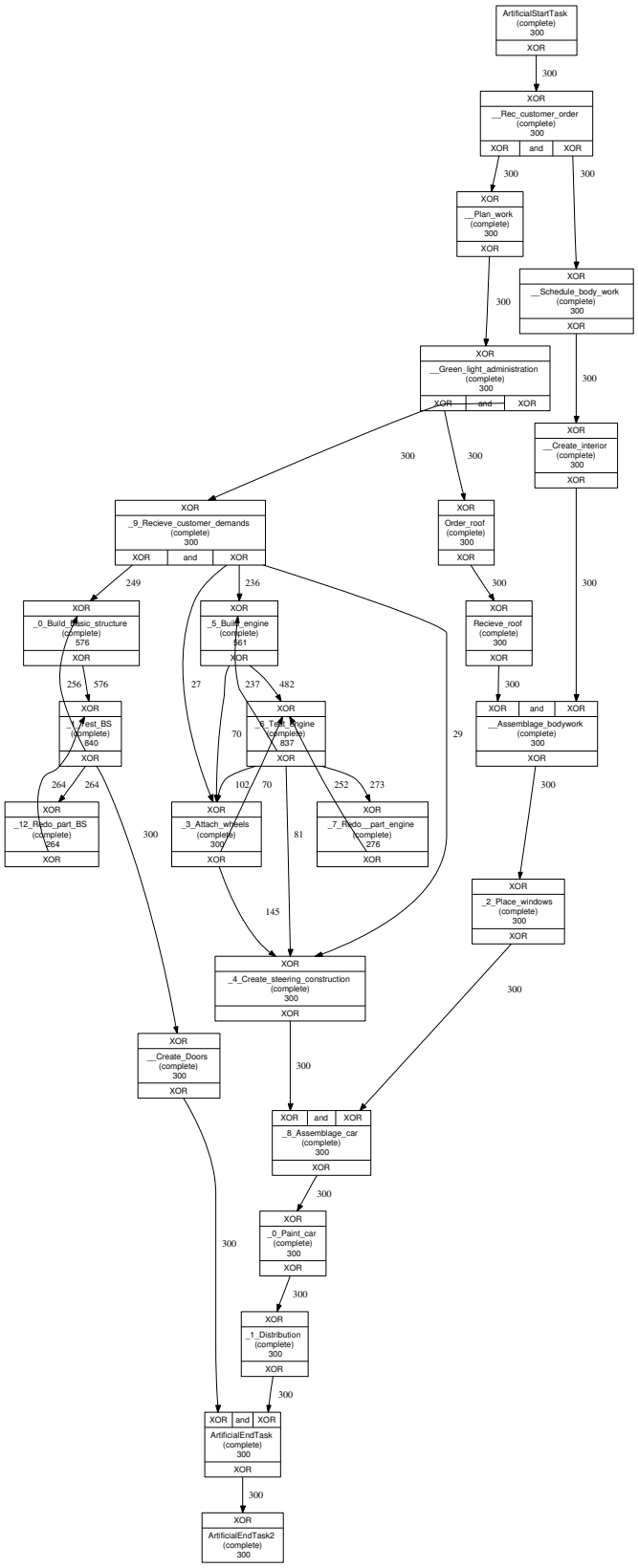


Figure 8.13: Mined model for net g23. Note that the four-parallel branches in the original model (cf. Figure 8.12) are not correctly captured, but the loop constructs are.

  - For the model g3 (cf. Figure 8.6), the parallel construct with loops (zoomed in Figure 8.7) is the main problem. This parallel construct starts with the AND-split activity "Purchase" and finishes with the AND-join activitiy "Administer_purchase". Note that this parallel construct has two branches with more than 2 tasks each. Besides, every branch is in itself a loop. This parallel construct was not correctly captured by the mined model (cf. Figure 8.8). However, the remaining structure of g3 was correctly captured in the mined model. To illustrate this, we show in Figure 8.9 the resulting model after the mined model has undergone arc pruning. This pruned model only shows the arcs and tasks that are used while replaying the traces that can be correctly parsed. Note that the pruned model indeed correctly captures the remaining structure of g3 (cf. original model in Figure 8.6) that does not include the parallel construct between the activities "Purchase" and "Administer_purchase".
  - For the model g14 (cf. Figure 8.10), the main problem is the unbalance between the AND-split and the AND-join activities. Note that the loop constructs in g14 were correctly captured in the mined model (cf. Figure 8.11), but the original unbalanced match between the AND-split and AND-join points was not.
  - For the model g23 (cf. Figure 8.12), the combination of multiple concurrent branches and unbalanced AND-split/join points is the main problem. Like for g14, the loop structures were correctly captured (cf. mined model for g23 in Figure 8.13) and the parallelism plus unbalanced AND-split/join points was the main issue.

  The mined models for g3, g14 and g23 indicate that the higher the amount of possible interleavings, the lower the probability that the GA will mine a model that is complete and precise.

- Multiple parallel branches with unbalance of split/join points are more harmful for the GA than the single presence of multiple parallel branches. For instance, consider the models g25 (cf. Figure 8.14) and g23 (cf. Figure 8.12). Both models have four parallel branches. However, the values of the behavioral precision and recall for the model g25 are better than the ones for the model g23 (cf. top graph in Figure 8.5). The main difference between these two models is that the AND-split/join points in g25 are balanced.

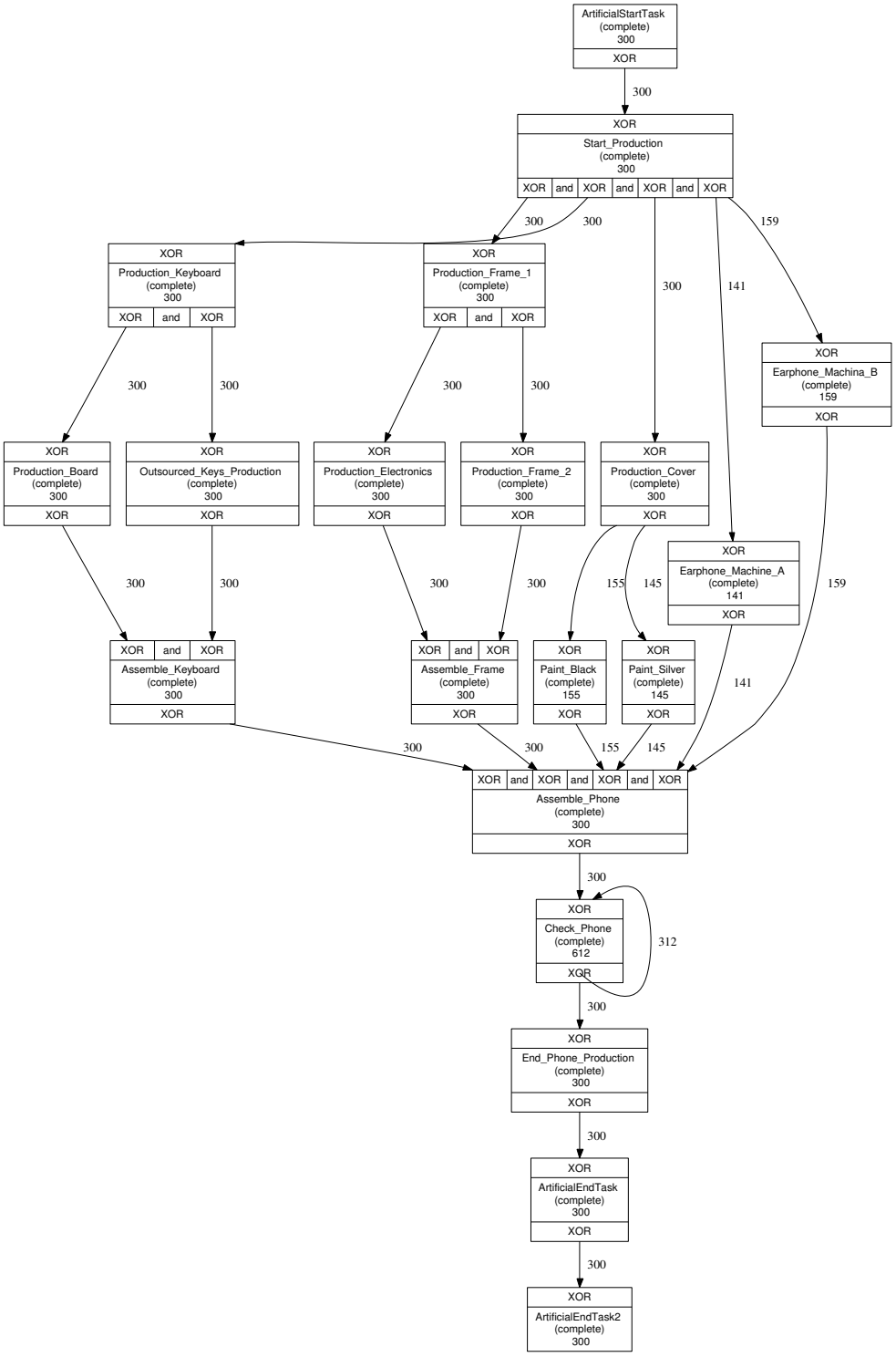


Figure 8.14: Original model for net g25. Like the net g23 (cf. Figure 8.12), this net also has four-parallel branches. However, unlike g23, all AND-split/join points in g25 are balanced.

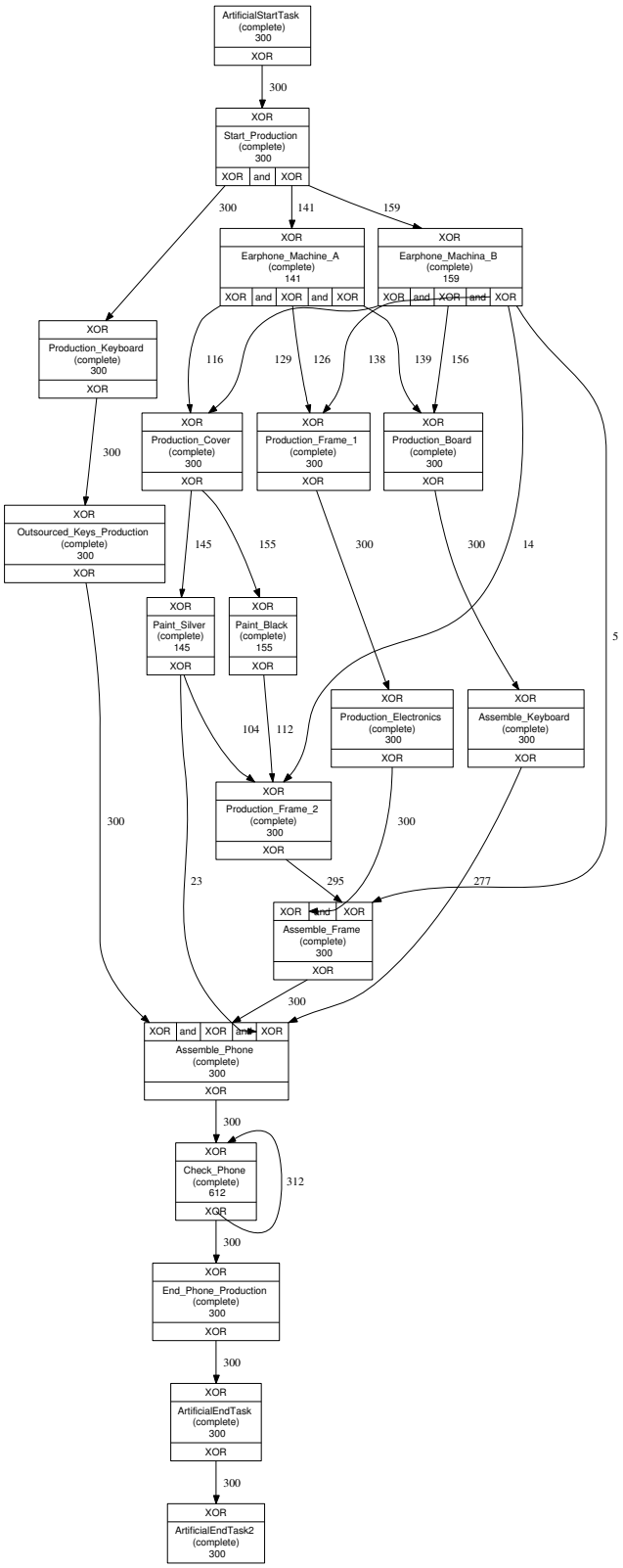


Figure 8.15: Mined model for net g25. Note that, although this model is different from the original one (cf. Figure 8.14), it better expresses the most frequent behavior in the log (cf. metrics in Figure 8.5) than the mined model for net g23 (cf. Figure 8.13).

- Six length-two loops connected to a same starting point hinder the GA mining more than three length-two loops connected to a same place. At least, this is the conclusion we can draw by comparing the results for the model `g8` and `g15`. Both models have length-two loops connected to a same initial point (cf. Figure 8.16 and 8.18). However, the results in Figure 8.5 show that mined model for `g8` can correctly parse more traces (proper completion fitness around 52%) than `g15` (proper completion fitness around 35%). Additionally, both the behavioral recall and precision of `g8` have higher values than for `g15`. By looking at the mined models for `g8` (cf. Figure 8.17) and `g15` (cf. Figure 8.19), one can notice that the three length-two loops connected to a same starting point are correctly mined for `g8`, but the six ones of `g15` are not. In fact, `g8` was not correctly mined because it contains unbalanced AND-split/join points in other parts of its structure. However, for `g15`, the lenght-two loops are the constructs that could not be correctly mined. Note that loops connected to a same starting point also lead to interleaving situations.

In our case, the blind experiments were important because (i) they highlighted the fact that we forgot to include nets with unbalanced AND-split/join constructs in the experiments with known models, and (ii) they reinforced the main conclusion drawn from the experiments in Chapter 4: *models that allow for many interleaving situations are more difficult to mine*. Clearly more exhaustive logs are needed for discovering correct models. In other words, there will always be some interleaving situations that are more frequent than others and often many interleavings will be missing.

All the original and mined models used during the blind experiments are included in Appendix C.

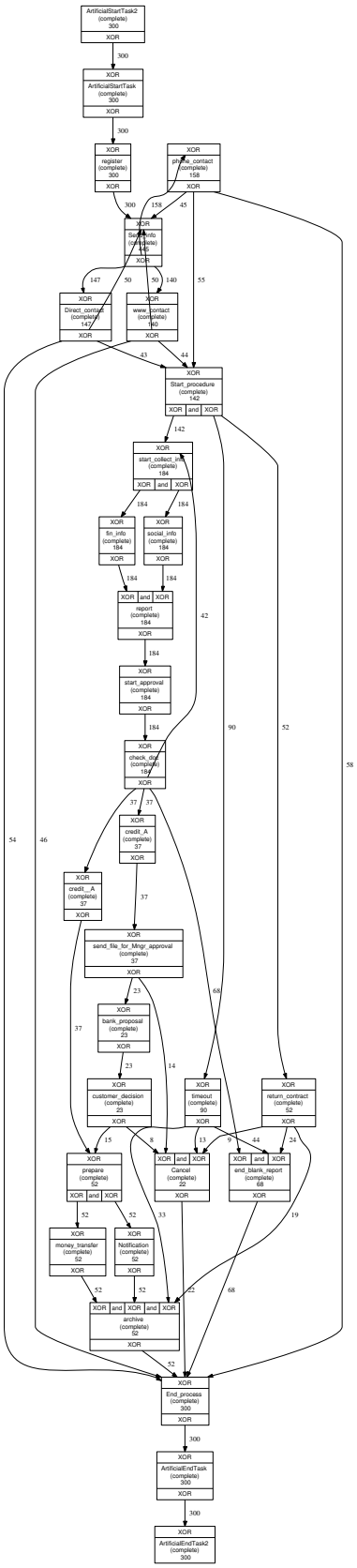


Figure 8.16: Original model for net g8. Note the three length-two loops after the activity "register".

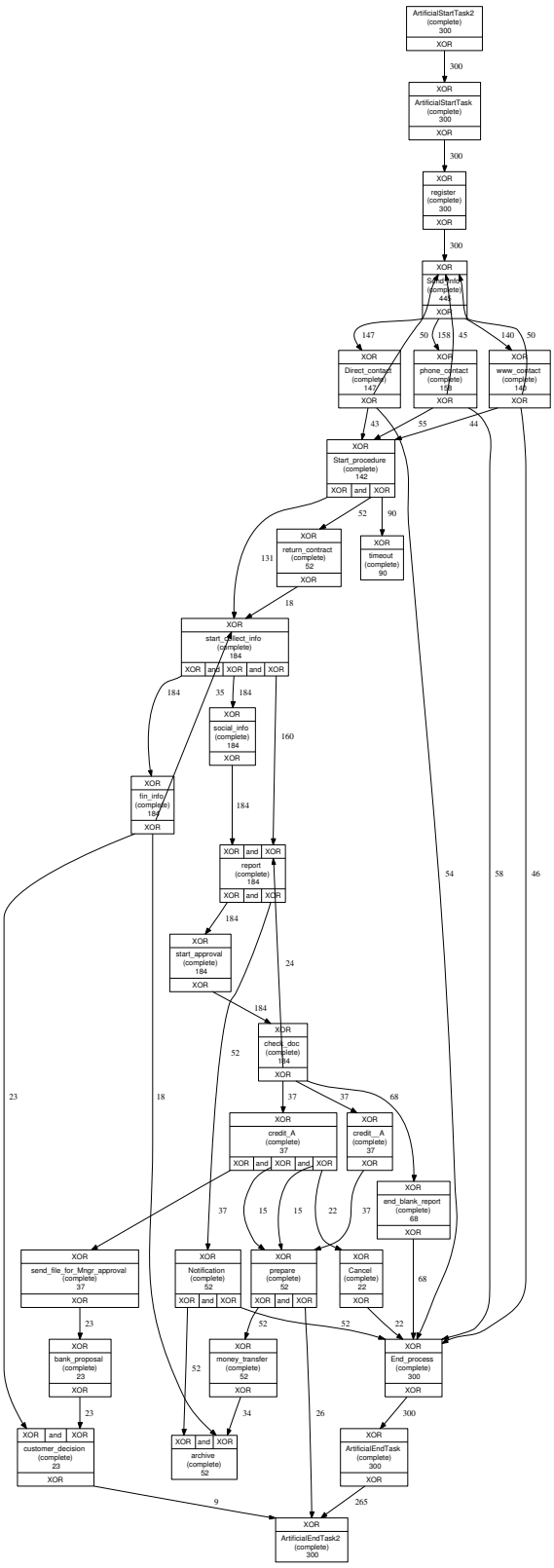


Figure 8.17: Mined model for net g8. Note that the three length-two loops after the activity "register" are correctly captured (cf. original model in Figure 8.16).

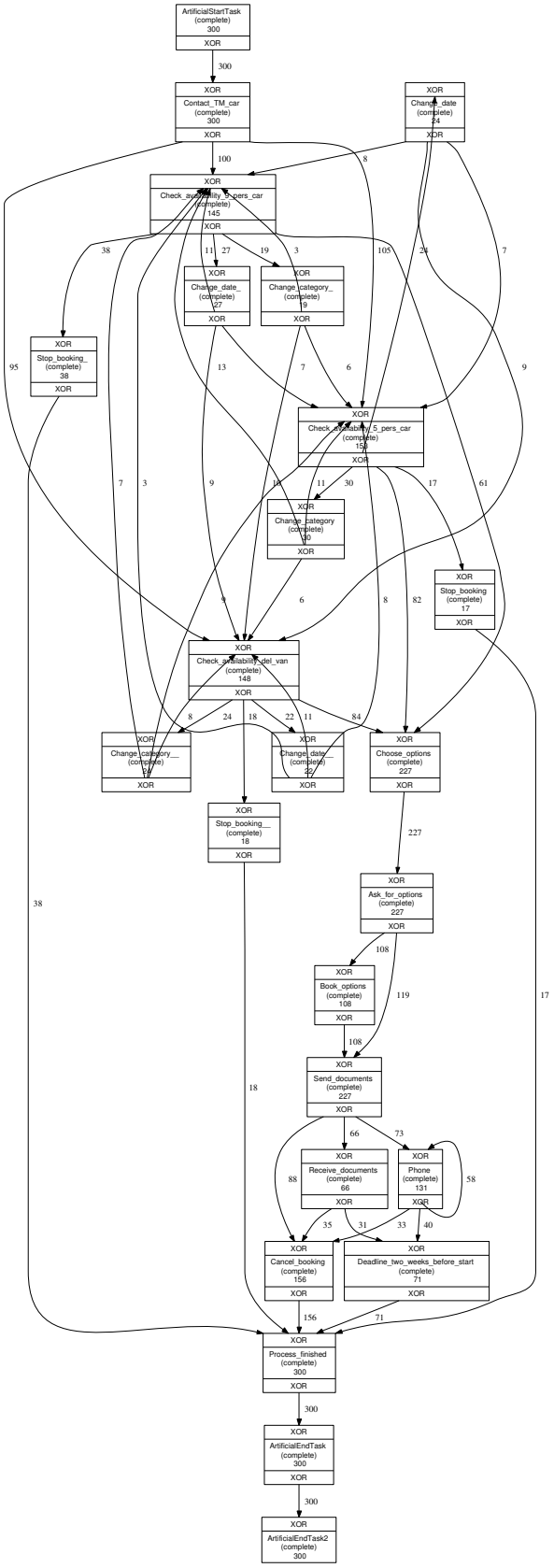


Figure 8.18: Original model for net g15. Note the six length-two loops after the activity "Contact_TM_car".

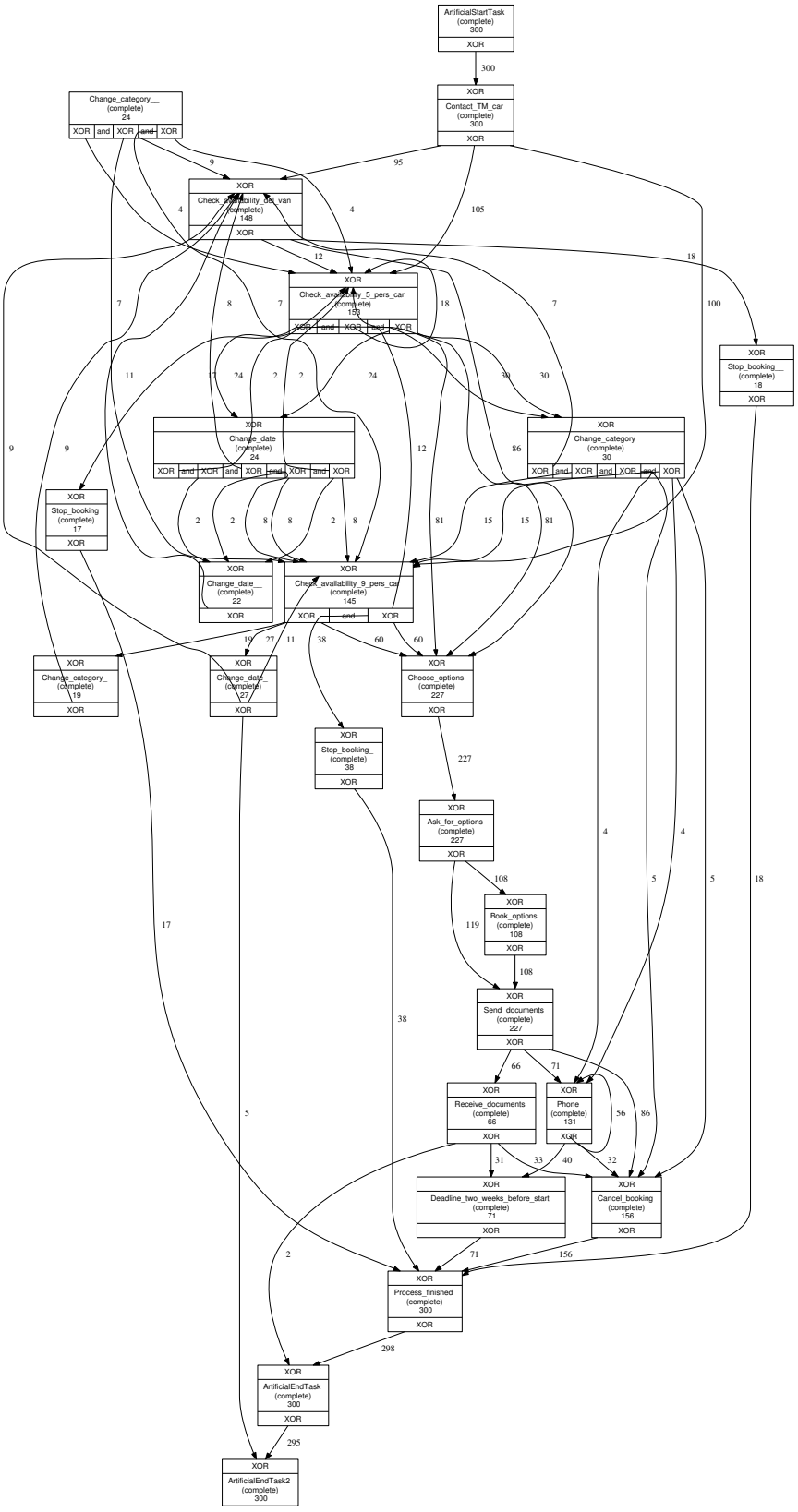


Figure 8.19: Mined model for net `g15`. Unlike the three length-two loops in `g8` (cf. Figure 8.17), the six length-two loops in `g15` (cf. original model in Figure 8.18) are not correctly captured in the mined model.

8.3 Case Study

The case study was conducted based on real-life logs from a municipality in The Netherlands. Here we obtained the logs of four processes: *Bezwaar*, *BezwaarWOZ*, *Afschriften* and *Bouwvergunning*. The first three processes deal with the handling of *complaints*, the last process (Bouwvergunning) has to do with getting a *building permit*. The workflow system used in the municipality is the "Eastman Software Workflow for NT"[4] [74]. For the four selected models, all constructs but non-free-choice and duplicate tasks are present. An overview of the control-flow patterns supported by the Eastman workflow is provided in [13].

The managers at the municipality were especially interested in using the process mining techniques for feedback analysis (or delta analysis). Among others, they would like to know (i) if there are deviations from the designed process, (ii) what the exact differences are, (iii) what the most frequent followed paths per process are and (iv) how the model that describes the current situation looks like. To check for discrepancies (points (i) to (iii)), we have used the *Conformance Checker plug-in* [67, 68] in the ProM framework (see Section 7.1). Since duplicates are not present in the prescribed models, we have used the *Genetic Algorithm plug-in* (see Chapter 4 and Section 7.2) to mine the logs (point (iv)). However, before any analysis could be performed, we needed to convert the provided logs to the MXML format (see Subsection 7.1.1) used by ProM, and we needed to clean the logs.

The *conversion* to the MXML format was done by using the *Eastman $ProM_{import}$ plug-in*. This import plug-in was explained in Subsection 7.6.2. The *data cleaning* consisted of two steps. In the *first step*, a projection was made of every case in the log of every process, so that only the tasks that were executed by the personnel of the municipality were considered. As an illustration, Figure 8.20 shows the model for the process Bezwaar. The tasks that have been crossed out are executed by external third parties and were removed from the logs of this process. The projection of the cases was done by using the log filter *Event Log Filter* of the ProM framework. The *second step* consisted of selecting the cases that were finished. The selection of such cases was made based on the information provided by managers of the municipality. For every process, they informed us about the *start* tasks and the *end* tasks. The filtering was done by using the log filters *Start Event Log Filter* and *Final Events Log Filter* that are also in the ProM framework. Furthermore, because the implementation of the Genetic Algorithm plug-in

[4]The current name of this workflow is Global360 (www.global360.com). However, for historic reasons, we have decided to keep the original name.

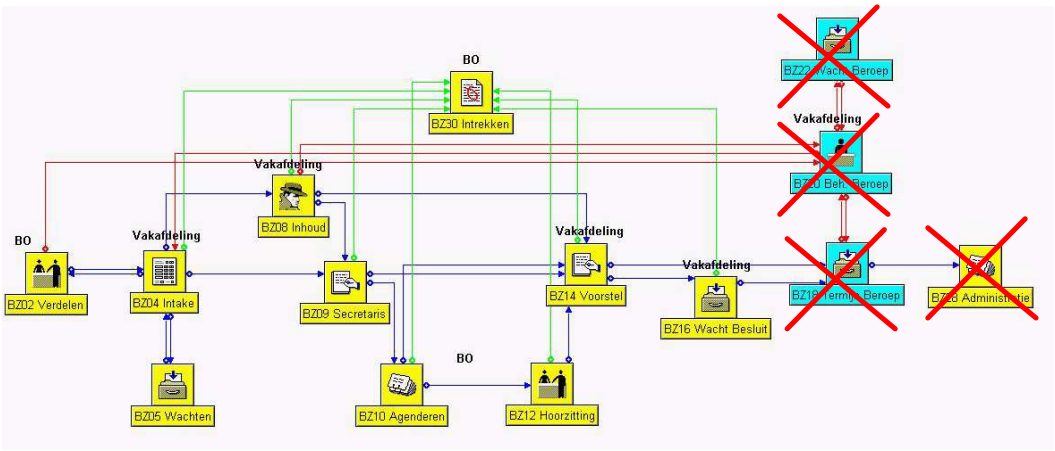


Figure 8.20: Prescribed model for the process *Bezwaar*. The crossed tasks were filtered out of the log during the data cleaning. The remaining tasks have an XOR-split/join semantics.

requires the target model to have a single start/end task that has XOR-split/join semantics, artificial start/end tasks were added at the start/end of every case. The artificial tasks were inserted by using the ProM log filters *Add Artificial Start Task Log Filter* and *Add Artificial End Task Log Filter*. Table 8.3 shows the numbers of cases in the log before and after the cleaning. As can be seen, many of the cases for the processes Bezwaar, BezwaarWOZ and Bouwvergunning have not been completed by the time these logs were collected. Once the logs have been cleaned, we have proceeded with their actual analysis.

Process Model	# of cases in *raw* log	# of cases in *cleaned* log	# of *unique* cases in cleaned log
Afschriften	374	358	4
Bezwaar	130	35	22
BezwaarWOZ	1982	747	54
Bouwvergunning	2076	407	107

Table 8.3: Information about the logs before and after cleaning, for every process model in the municipality. # stands for "number". The cases were grouped by using the export plug-in *Group Log (same sequence)* (cf. Subsection 7.5.1).

The remainder of this section is organized as follows. Subsection 8.3.1 shows the results of the analysis for the log replay performed by the Confor-

mance Checker plug-in on the original models. Subsection 8.3.2 presents the models mined by the Genetic Algorithm plug-in. Subsection 8.3.2 contains the reflections of the responsible people in the municipality about the results for the log replay and re-discovered processes.

8.3.1 Log Replay

To check how compliant the cases in the cleaned logs were with the deployed (or original) models, we needed to import the original models and logs into ProM. The results after having played with the Conformance Checker plug-ins are summarized in Table 8.4. Note that all cases (100%) for the process Afschriften are compliant with the original (or deployed or prescribed) model, and most of the cases (80%) for the process Bouwvergunning do not differ from the prescribed behavior. However, many cases of the other two processes - Bezwaar and BezwaarWOZ - do not comply with the deployed models. Actually, the most frequent path (19% of the cases) for the process Bezwaar is not compliant with the original model. The discrepancy is due to the presence of a length-two loop between the tasks "BZ12 Voorstel" and "BZ16 Wacht Besluit"[5]. Note that the original model (cf. Figure 8.20) does have a connection (arc or dependency) from "BZ16 Wacht Besluit" to "BZ14 Voorstel". The most frequent paths per model are summarized in Table 8.5.

Process Model	Number of Cases	% Correctly Parsed Cases Original Model	Figure
Afschriften	358	100%	8.21
Bezwaar	35	51%	8.22 & 8.23
BezwaarWOZ	747	46%	8.24 & 8.25
Bouwvergunning	407	80%	8.26 & 8.27

Table 8.4: Percentage of cases that could be correctly parsed by the original process models. A case is correctly parsed when no tokens are missing or left behind during the parsing.

[5] "Voorstel" means "offer, proposal" in English, and "Wacht Besluit" means "Wait for Decision".

Process Model	Most frequent path for all cases
Afschriften	**88%** of the cases follow the **compliant** path: DH1→AG02→AG04→AG08
Bezwaar	**14%** of the cases follows the **non-compliant** path: DH1→BZ02→BZ04→BZ08→BZ09→BZ10 →BZ12→BZ14→BZ16→**BZ14→BZ16** **11%** of the cases follows the **compliant** path: DH1→BZ02→BZ04→BZ08→BZ30
BezwaarWOZ	**19%** of the cases follows the **compliant** path: DH1→OZ02→OZ06→OZ08→0Z12→OZ16 →OZ20→OZ24 However, **17%** of the cases follows the **non-compliant** path: DH1→OZ02→OZ06→**OZ12**→OZ16→OZ20 →OZ24
Bouwvergunning	**31%** of the cases follows the **compliant** path: DH1→BV02→BV04→BV06 Brandweer →BV06 Milieu→BV06 CCT→BV06 Toets ontv →BV08 →BV10→BV16→BV22→BV24

Table 8.5: The most frequent paths for every process model.

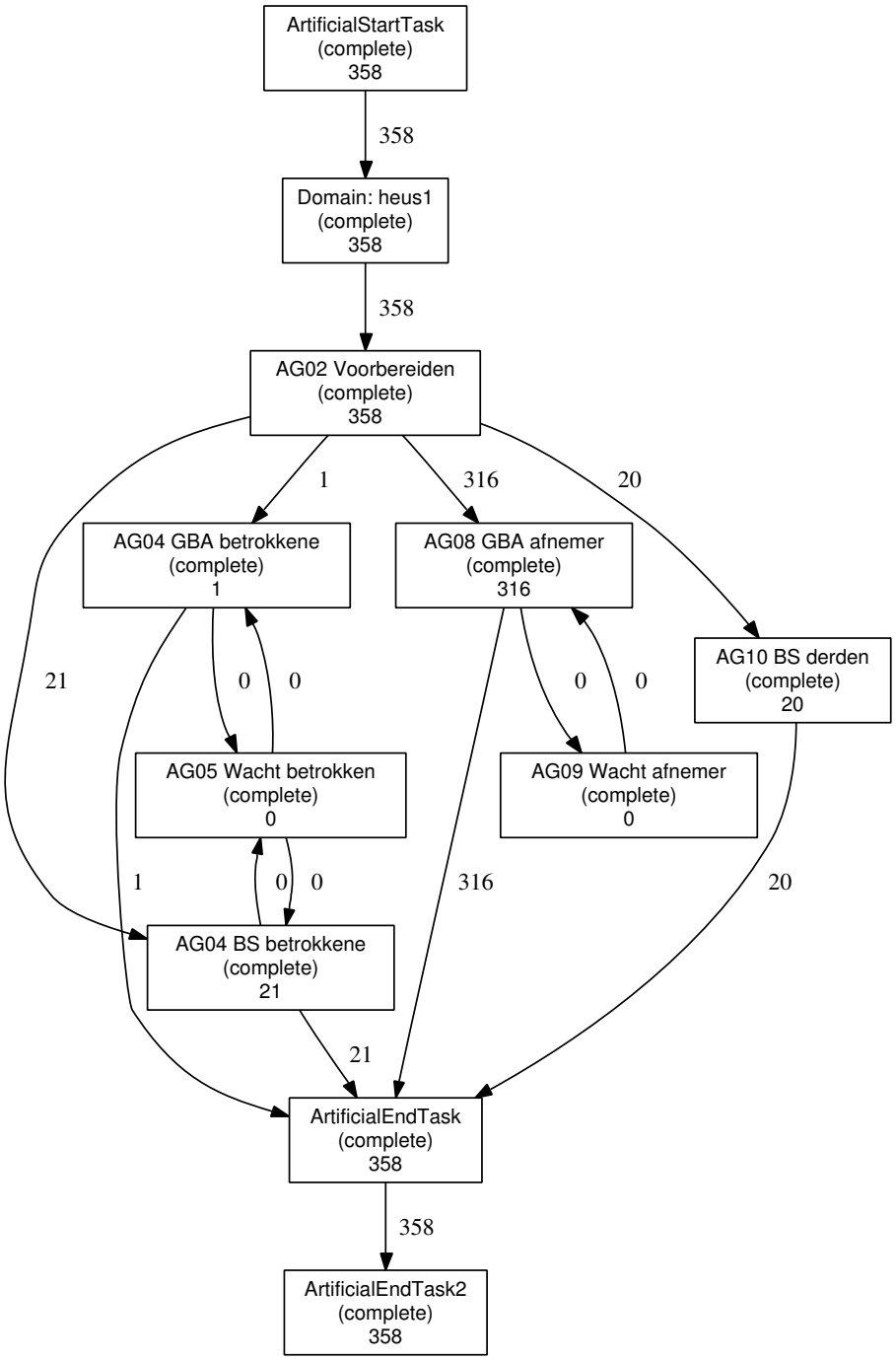


Figure 8.21: Original model for Afschriften. The numbers associated to the arcs indicate how often the arc has been used during the log replay. In a similar way, the numbers inside the activities (third textual line in an activity) indicate how often the activity has been used during the log replay. Furthermore, to improve the readability of the text in the models, we have omitted the semantics of the activities whenever they are all XOR-split/join points. This "omission policy" applies to all models presented in this chapter.

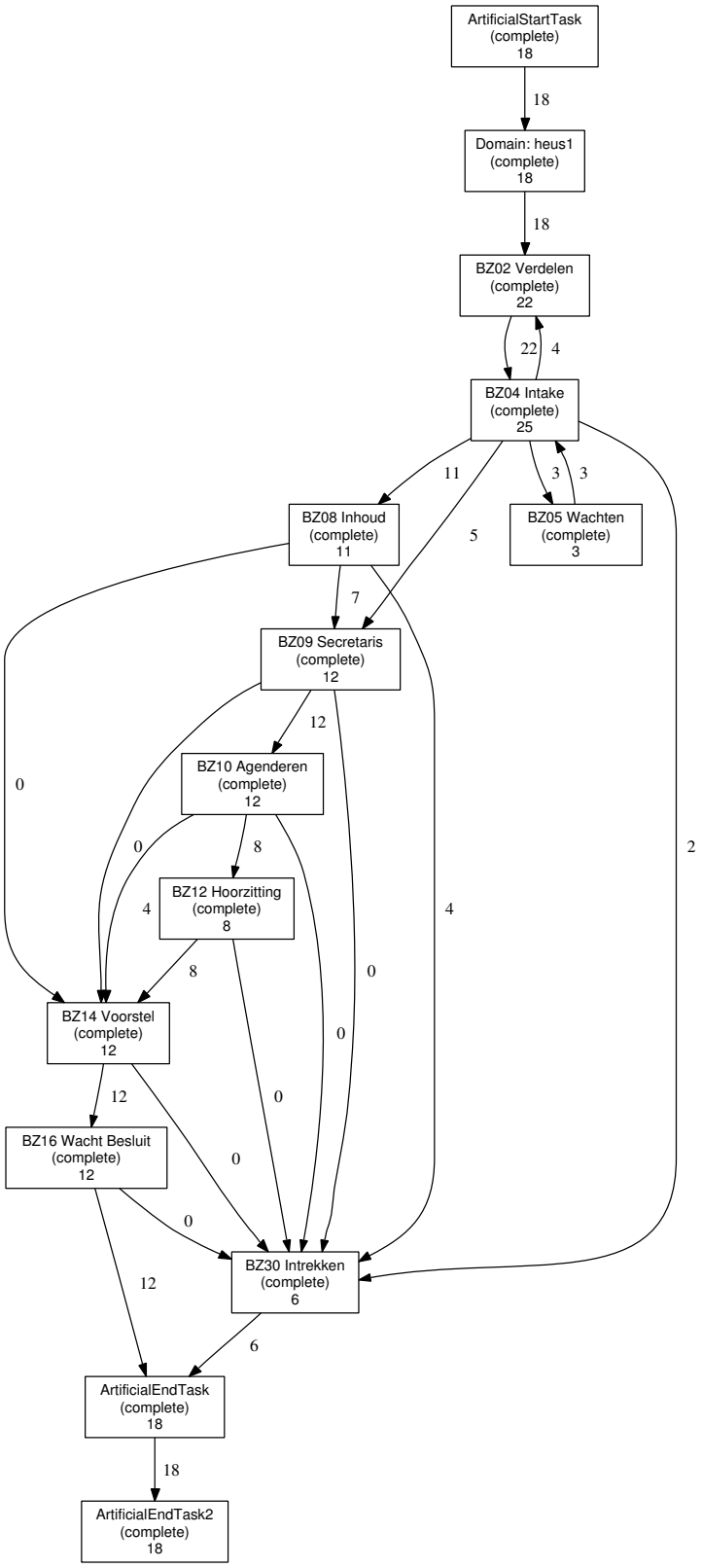


Figure 8.22: Original model for Bezwaar - Replay of compliant traces (18 out of 35).

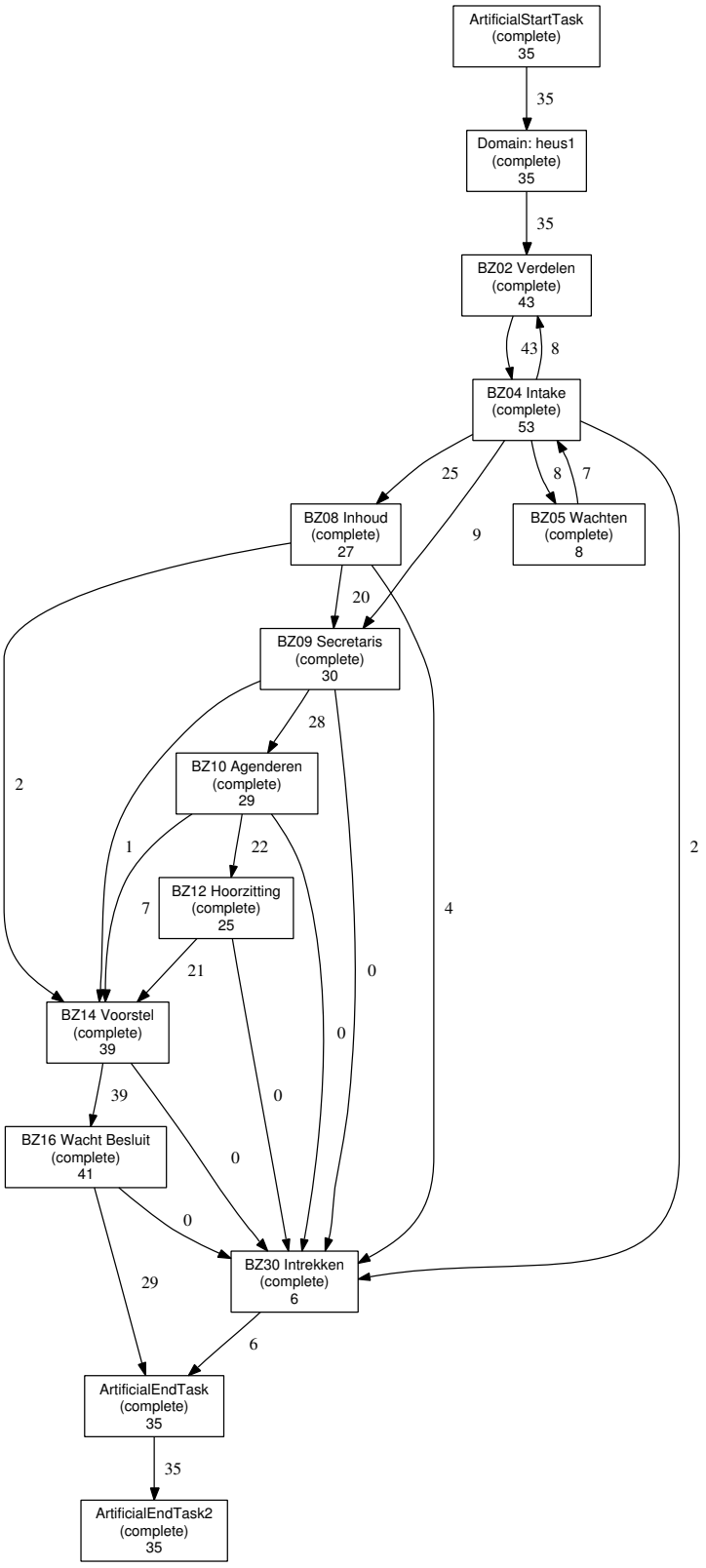


Figure 8.23: Original model for Bezwaar - Replay of all traces.

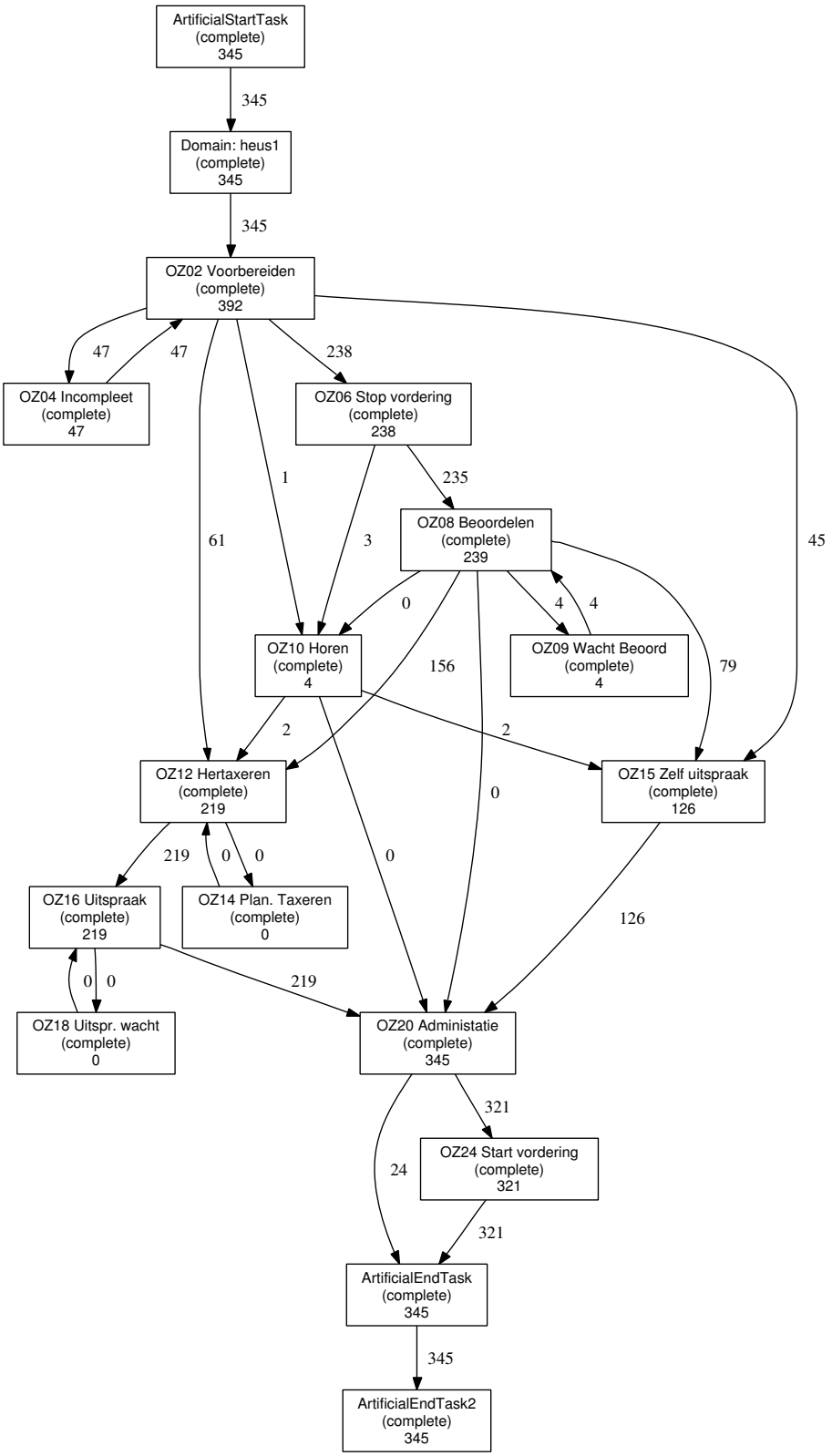


Figure 8.24: Original model for BezwaarWOZ - Replay of compliant traces (345 out of 747).

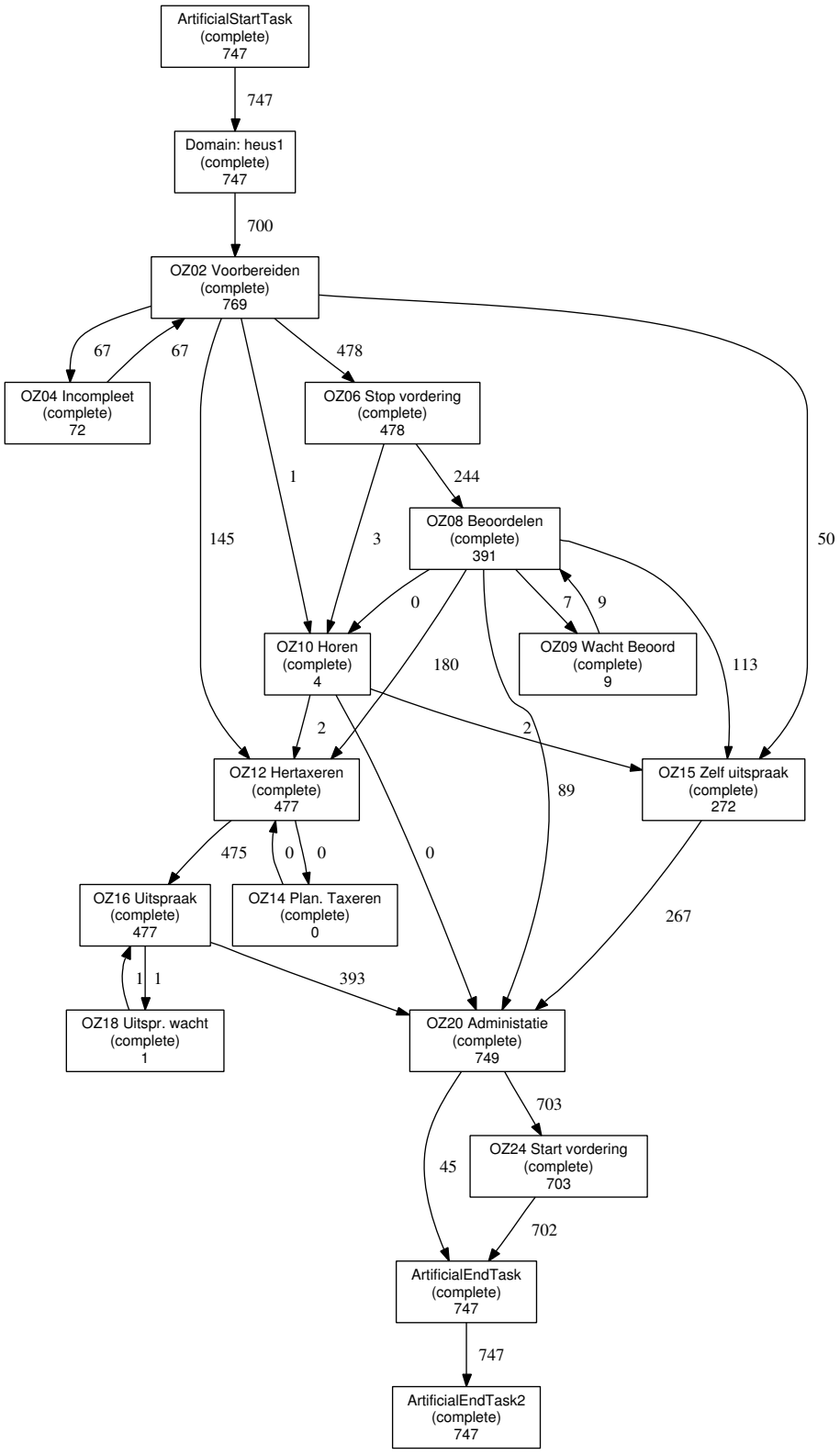


Figure 8.25: Original model for BezwaarWOZ - Replay of all traces.

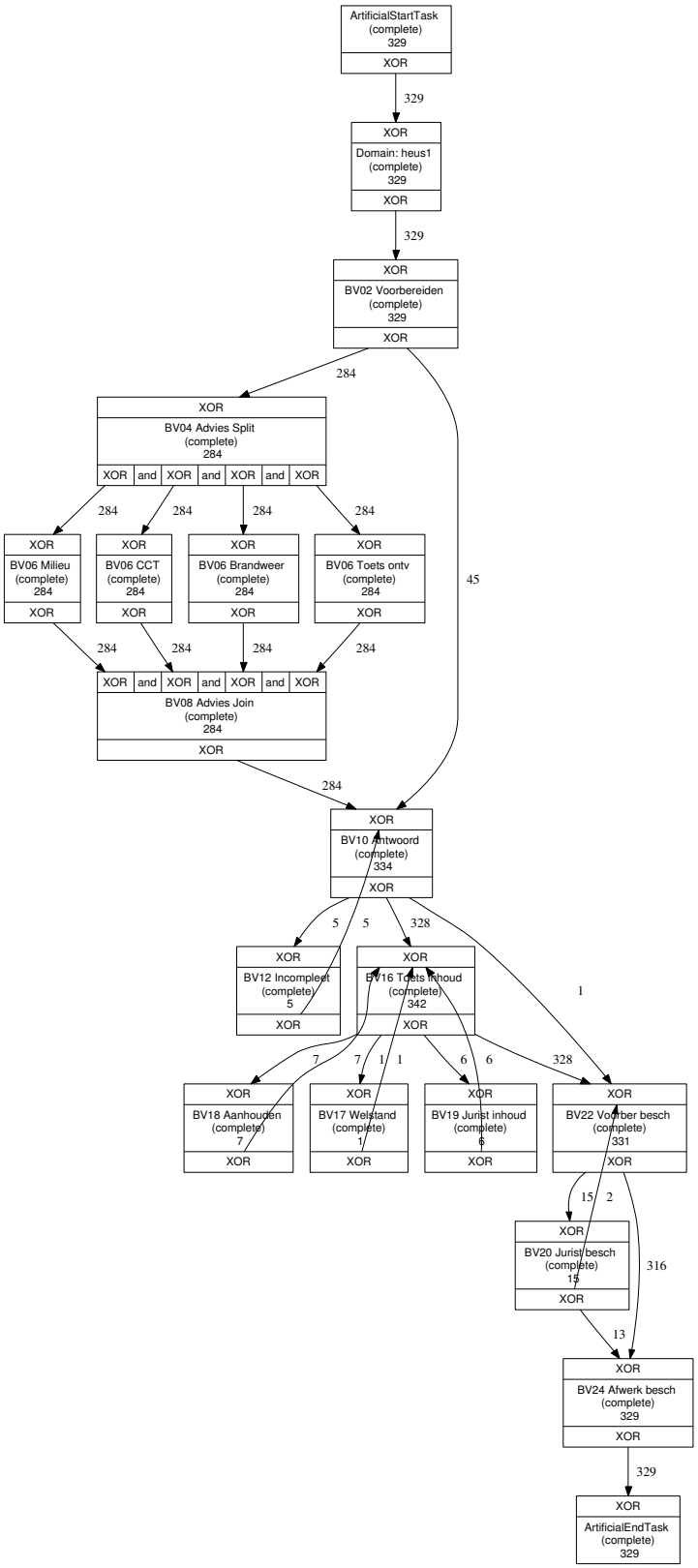


Figure 8.26: Original model for Bouwvergunning - Replay of compliant traces (329 out of 407).

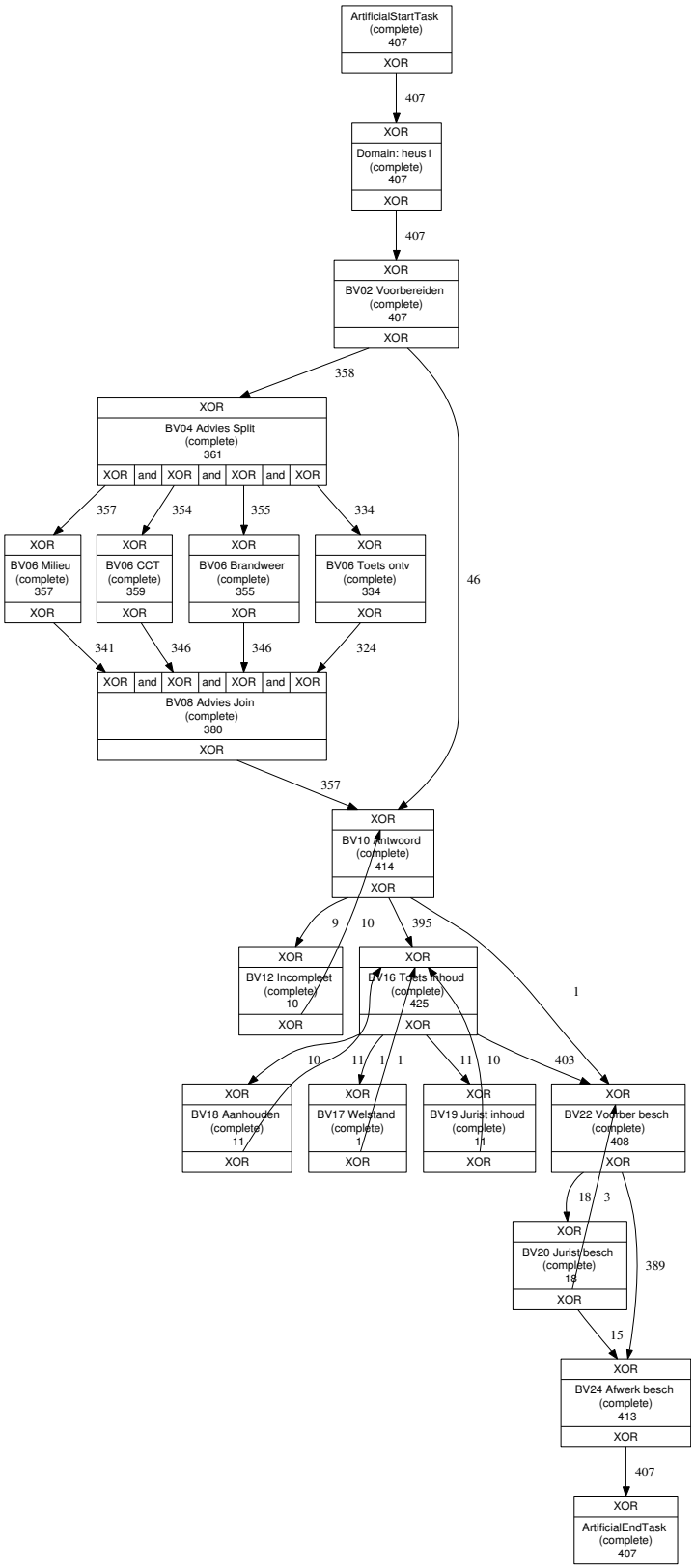


Figure 8.27: Original model for Bouwvergunning - Replay of all traces.

Figures 8.28 and 8.29 respectively show screenshots of the *model and log diagnostic perspective* of the Conformance Checker plug-in. These two perspectives provide detailed information about the problems encountered during the log replay. The *model perspective* diagnoses information about *token counter* (number of missing/left tokens), *failed tasks* (tasks that were not enabled), *remaining tasks* (tasks that remained enabled), *path coverage* (the tasks and arcs that were used during the log replay) and *passed edges* (how often every arc in the model was used during the log replay). The *log perspective* indicates the points of non-compliant behavior for every case in the log. As an illustration, consider the length-two loop situation of the most frequent case for the process Bezwaar. The screenshot in Figure 8.28 shows the diagnosed problems for the tasks "BZ14 Voorstel" and "BZ16 Wacht Besluit". The selected cases are the ones that *do not fit*, i.e., the non-compliant ones. Note that, for this selection, the task "BZ14 Voorstel" could be replayed without any problems for 10 out of the 17 non-compliant cases, it cannot be replayed once for 6 of the non-compliant cases, and for 1 non-compliant case, this task could not be correctly replayed twice (since 2 tokens were missing and this task has a single input place). A further look at the log perspective of these non-fitting traces (see Figure 8.29) reveals that this is indeed the case. Note that the arrow in Figure 8.29 points out that the task "Voorstel" happened three times for one of the non-compliant process instances. The first occurrence could be replayed without any problems and, therefore, the task is not highlighted. However, the second and third occurrences were not possible according to the model and, consequently, are highlighted in the log diagnosis view. All this illustrates that the Conformance Checker is a useful tool to inspect the match between the executed cases and the deployed models.

8.3.2 Re-discovery of Process Models

This section shows the results of applying the *Genetic Algorithm plug-in* to the four processes in the municipality. The aim is to get a concise picture (the mined models) of what is really happening in the log. Based on the log-replay results (cf. Subsection 8.3.1), we should expect the mined model for the process Afschriften to be very similar to the deployed model. Note that all the cases for this process are compliant with the deployed model (cf. Table 8.4). For the other three processes - Bezwaar, BezwaarWOZ and Bouwvergunning - we would expect the mined models to differ from the deployed models because many cases are not compliant with these models. More specifically, the mined models should capture the most frequent paths in the log.

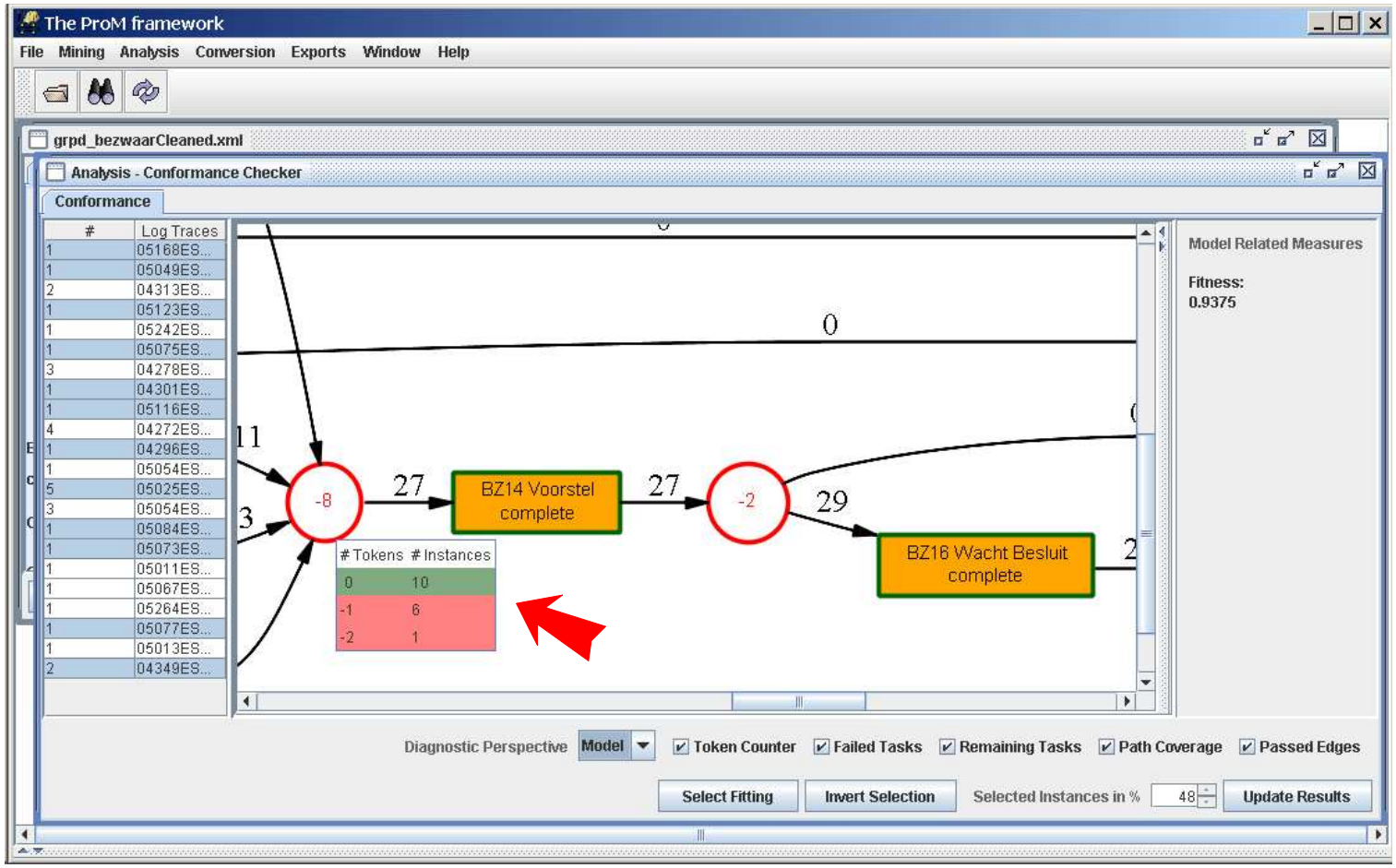


Figure 8.28: Conformance Checker plug-in - Screenshot of the "Model Diagnostic Perspective" for the process Bezwaar.

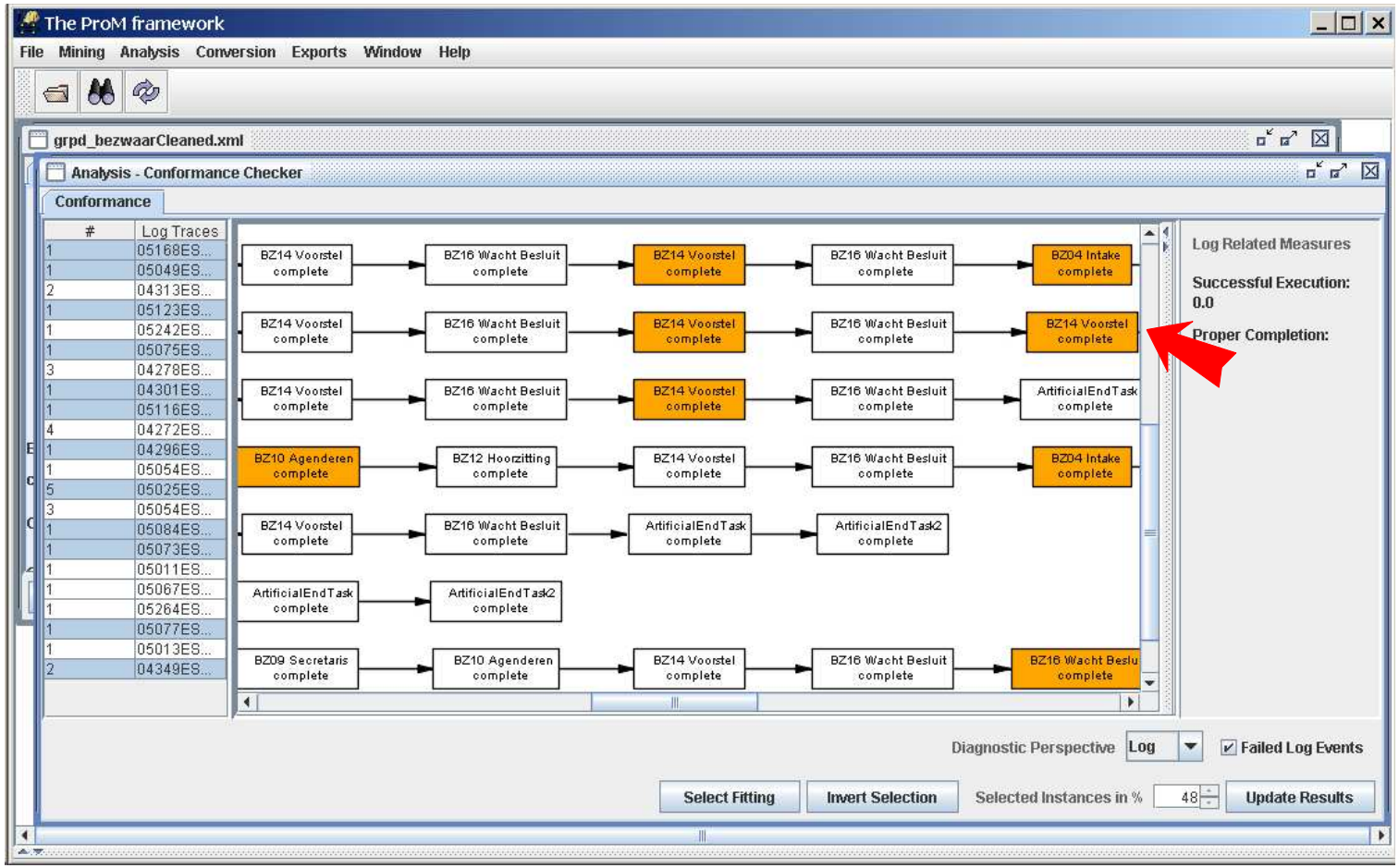


Figure 8.29: Conformance Checker plug-in - Screenshot of the "Log Diagnostic Perspective" for the process Bezwaar.

	% Correctly Parsed Traces Mined Model & Figure			
Process Model	*no pruning*	*1% pruning*	*5% pruning*	*10% pruning*
Afschriften	100% & 8.30	99% & 8.31	99% & 8.31	88% & 8.32
Bezwaar	100% & 8.33	100% & 8.33	62% & 8.34	51% & 8.35
BezwaarWOZ	98% & 8.36	95% & 8.37	91% & 8.38	68% & 8.39
Bouwvergunning	77% & 8.40	76% & 8.41	73% & 8.42	70% & 8.43

Table 8.6: Percentage of traces that could be correctly parsed by the mined process models before and after the arc pruning. A trace is correctly parsed when no tokens are missing or left behind during the parsing.

Process Model	S_P	S_R	B_P	B_R	$PF_{complete}$
Afschriften	1.0	0.65	1.0	0.92	1.0
Bezwaar	0.73	0.92	0.80	0.79	1.0
BezwaarWOZ	0.80	0.81	0.85	0.73	0.99
Bouwvergunning	0.62	0.59	0.76	0.85	0.97

Table 8.7: Analysis metrics for the mined models.

For every log, the experimental setup was the same as for the single-blind experiments (see Section 8.2, on page 173), but the maximum number of generations was 10,000. Again 10 runs were generated for each log. From these 10 runs, the best models were selected. The best model is the one that correctly parses most of the cases in the log. The results are summarized in Table 8.6 and Table 8.7. Table 8.6 shows the percentage of cases that can be correctly parsed by the selected mined models. Note that the reported results also include the pruned models of these mined models. Table 8.7 shows the values of the analysis metrics for the mined models. Note that, because the original models for Bezwaar, BezwaarWOZ and Bouwvergunning cannot correctly parse all the behavior in their logs (cf. Table 8.4), we expect the values for the behavioral/structural precision and recall metrics to point out the presence of discrepancies between these models and the mined ones. To facilitate the visual comparison between the original and mined models, we show their heuristic nets in figures 8.21 to 8.27 (for the original models) and 8.30 to 8.43 (for the mined models). Based on these figures and tables 8.4, 8.6 and 8.7, we can conclude that:

- The results in Table 8.6 reinforce that the fitness measure of our genetic approach indeed guides the search towards individuals that can correctly parse the most frequent behavior in the log. Note that, even after the mined models have undergone 10% arc pruning, they are still able to parse at least 51% (process Bezwaar) of the cases in the logs.

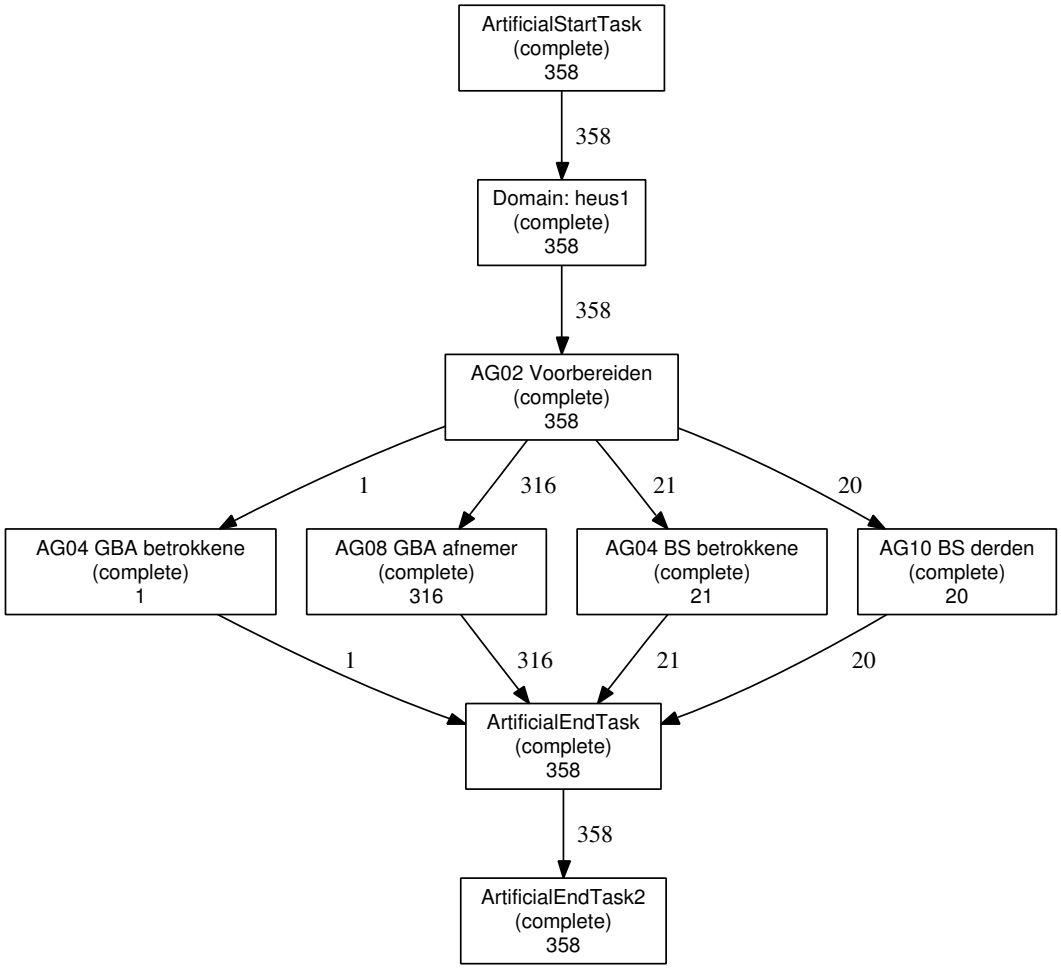


Figure 8.30: Mined model for Afschriften - Replay of compliant traces.

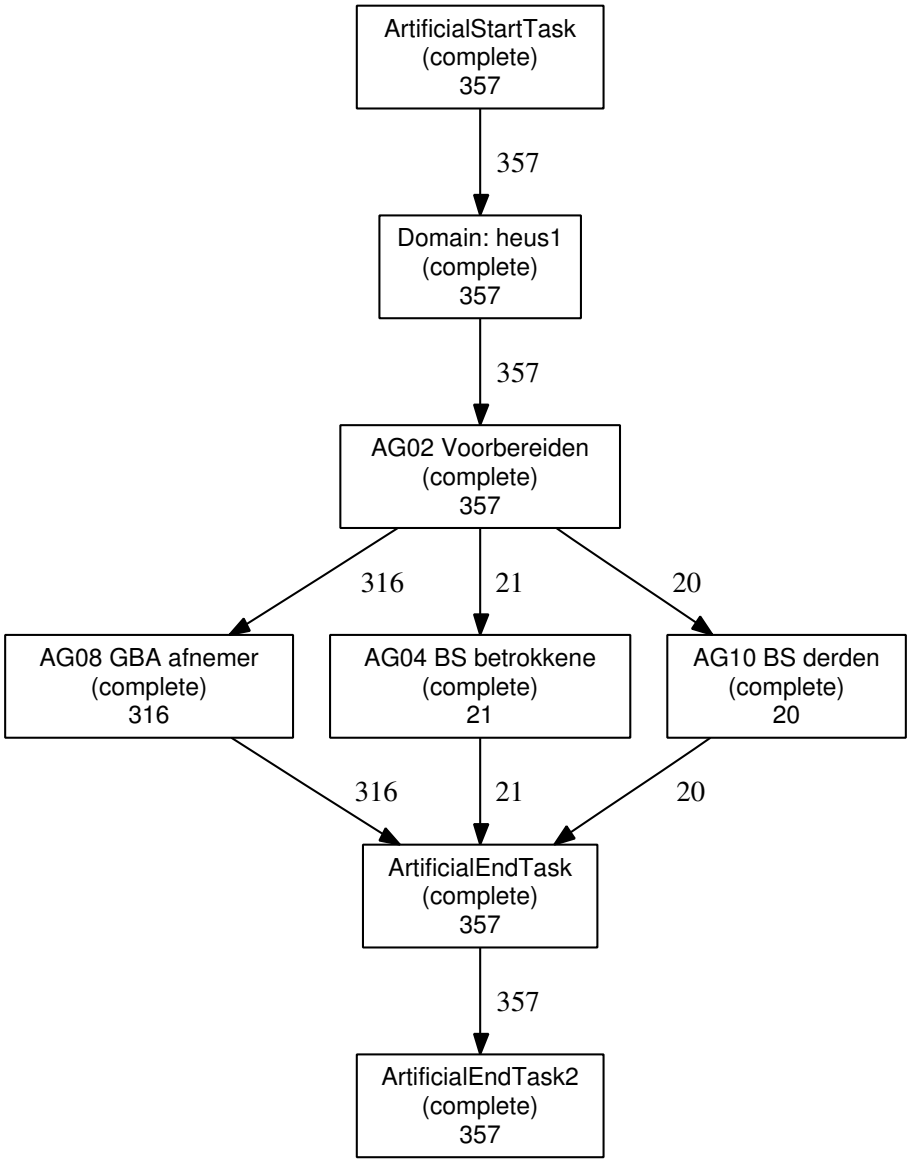


Figure 8.31: Afschriften - 1% or 5% arc pruning over the mined model in Figure 8.30.

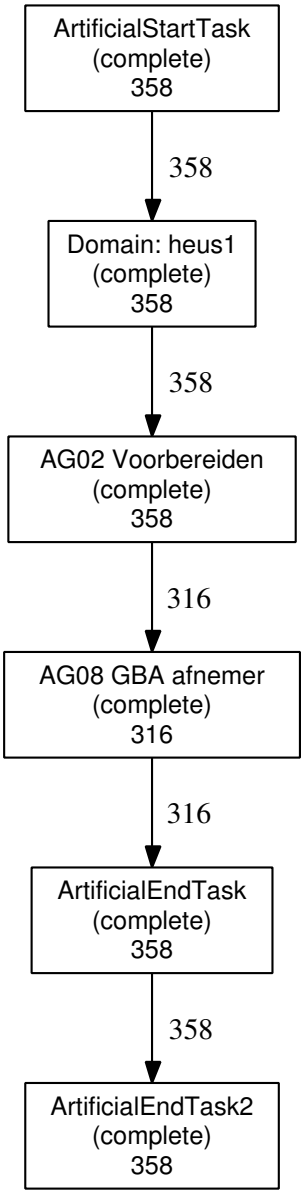


Figure 8.32: Afschriften - 10% arc pruning over the mined model in Figure 8.30.

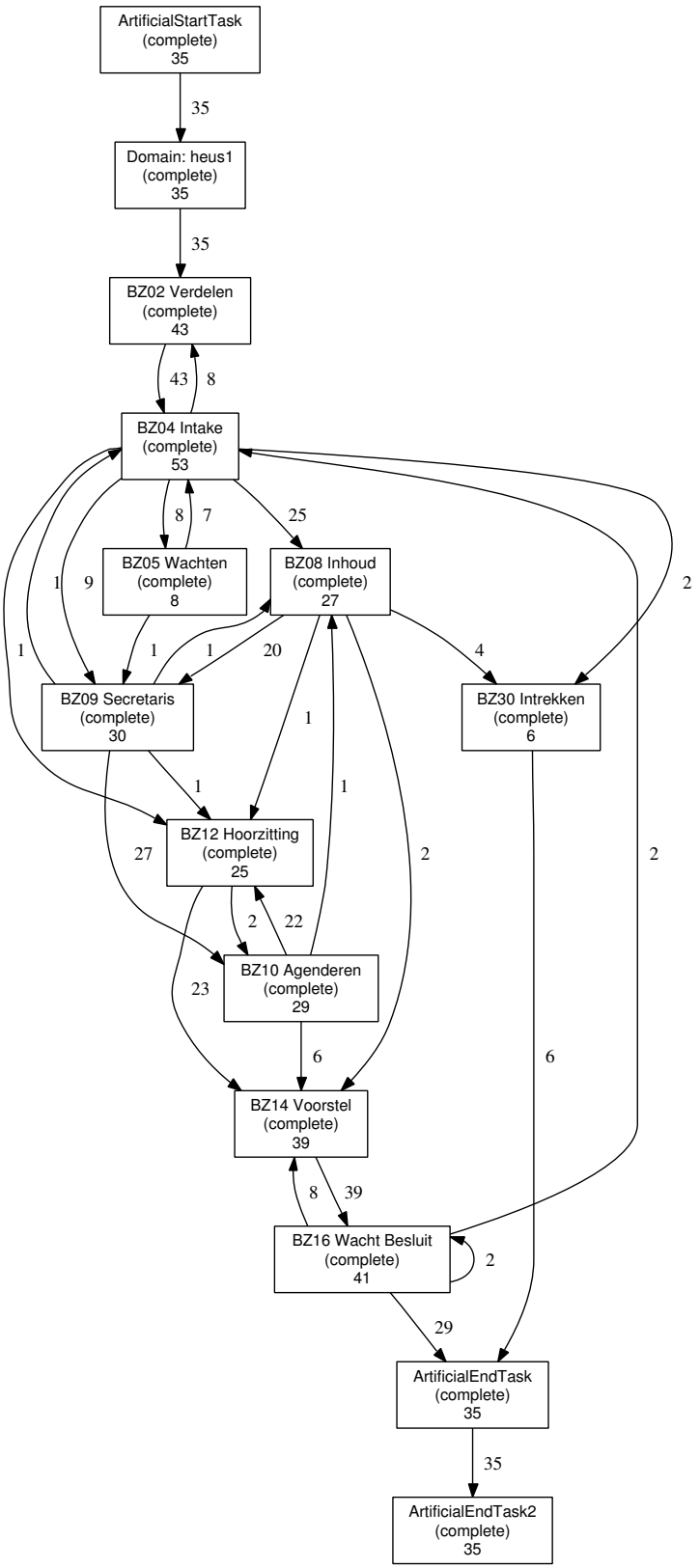


Figure 8.33: Mined model for Bezwaar - Replay of compliant traces.

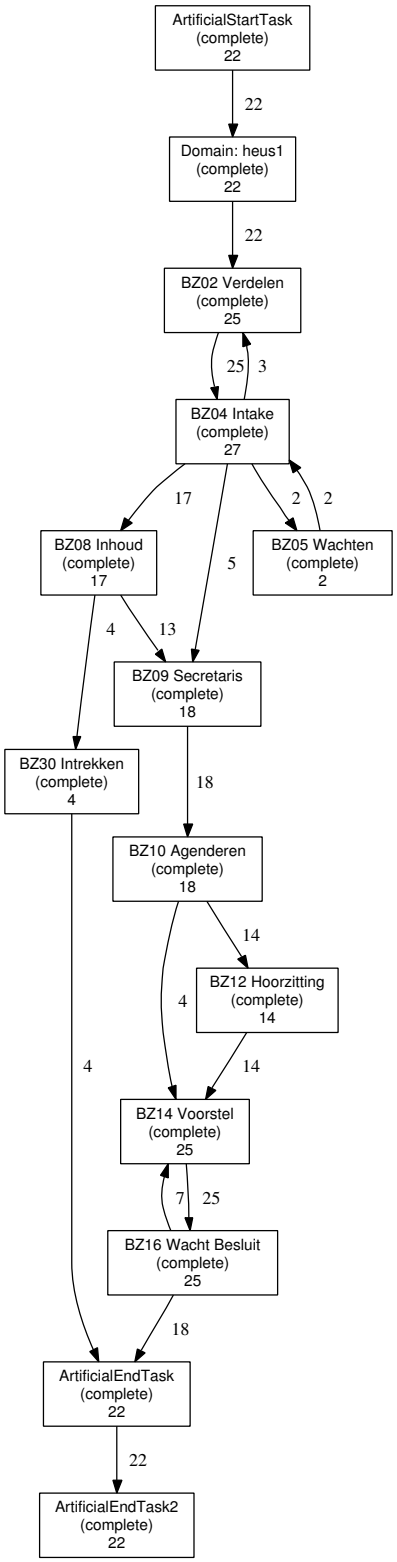


Figure 8.34: Bezwaar - 1% or 5% arc pruning over the mined model in Figure 8.33.

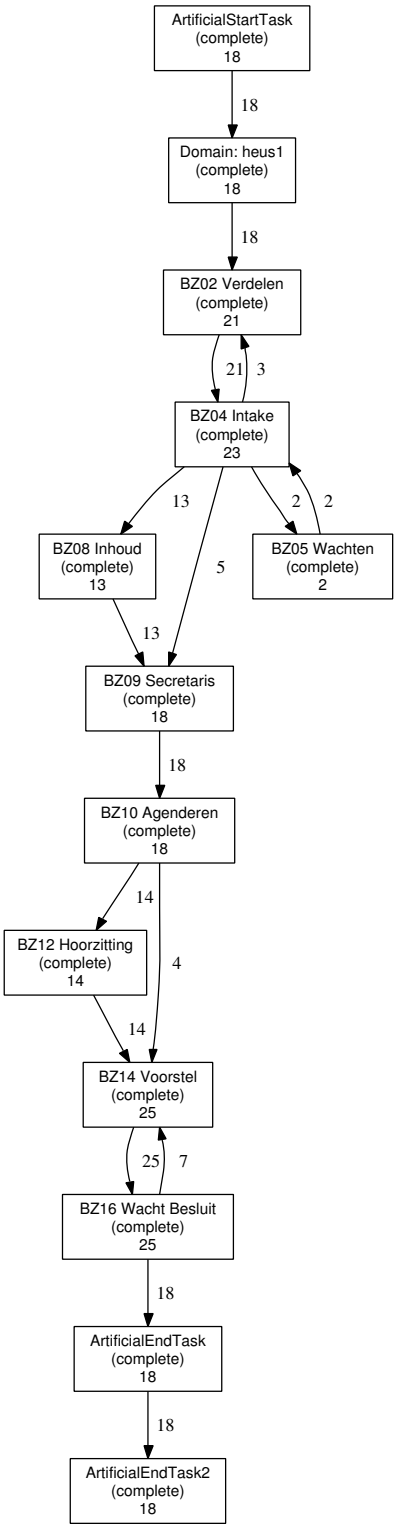


Figure 8.35: Bezwaar - 10% arc pruning over the mined model in Figure 8.33.

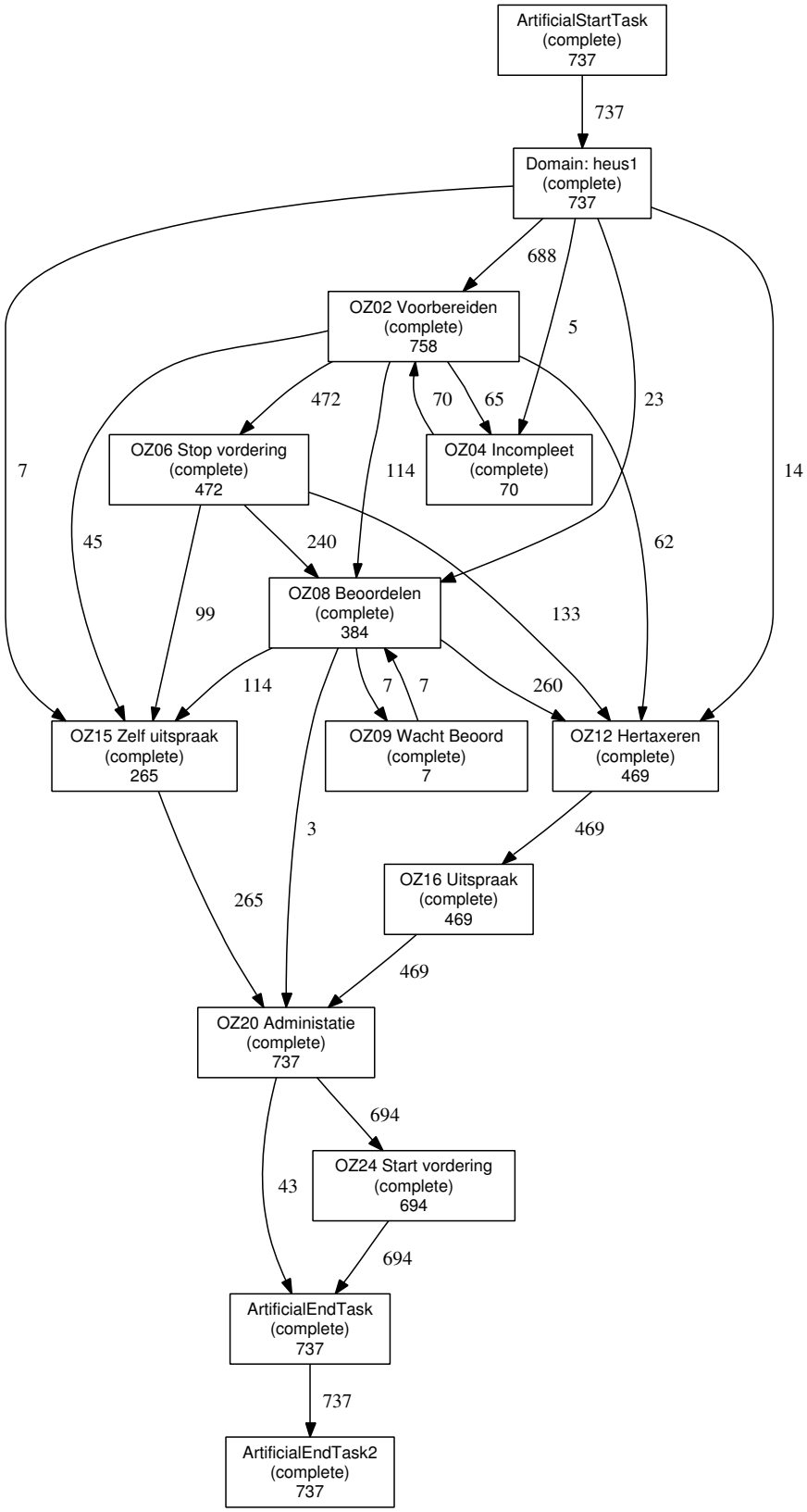


Figure 8.36: Mined model for BezwaarWOZ - Replay of compliant traces.

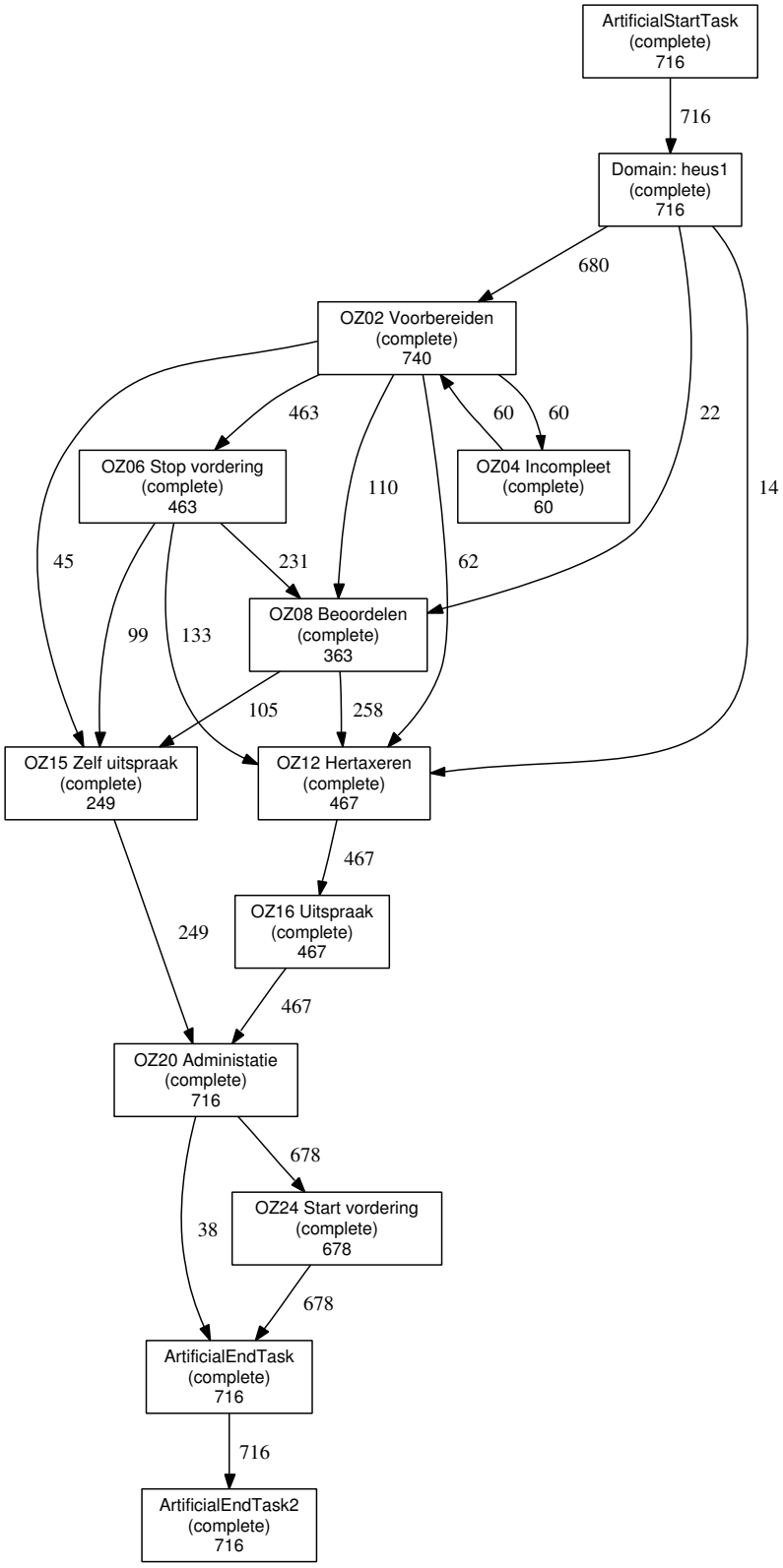


Figure 8.37: BezwaarWOZ - 1% arc pruning over the mined model in Figure 8.36.

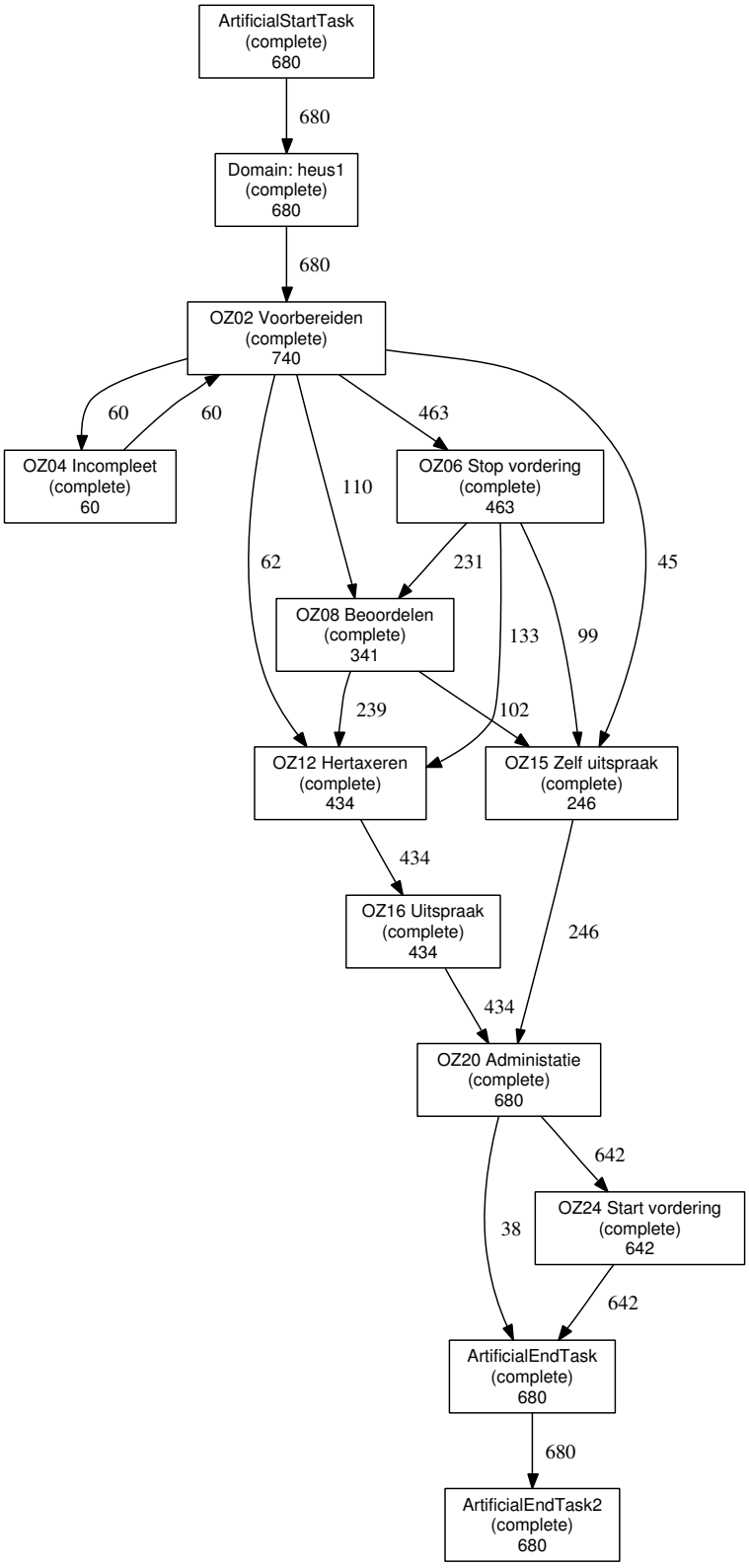


Figure 8.38: BezwaarWOZ - 5% arc pruning over the mined model in Figure 8.36.

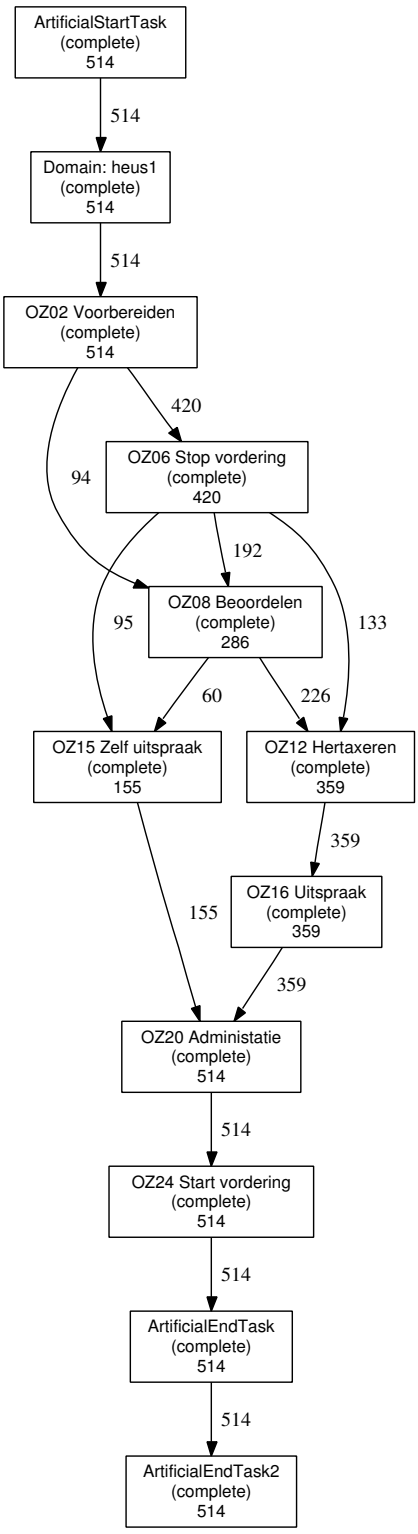


Figure 8.39: BezwaarWOZ - 10% arc pruning over the mined model in Figure 8.36.

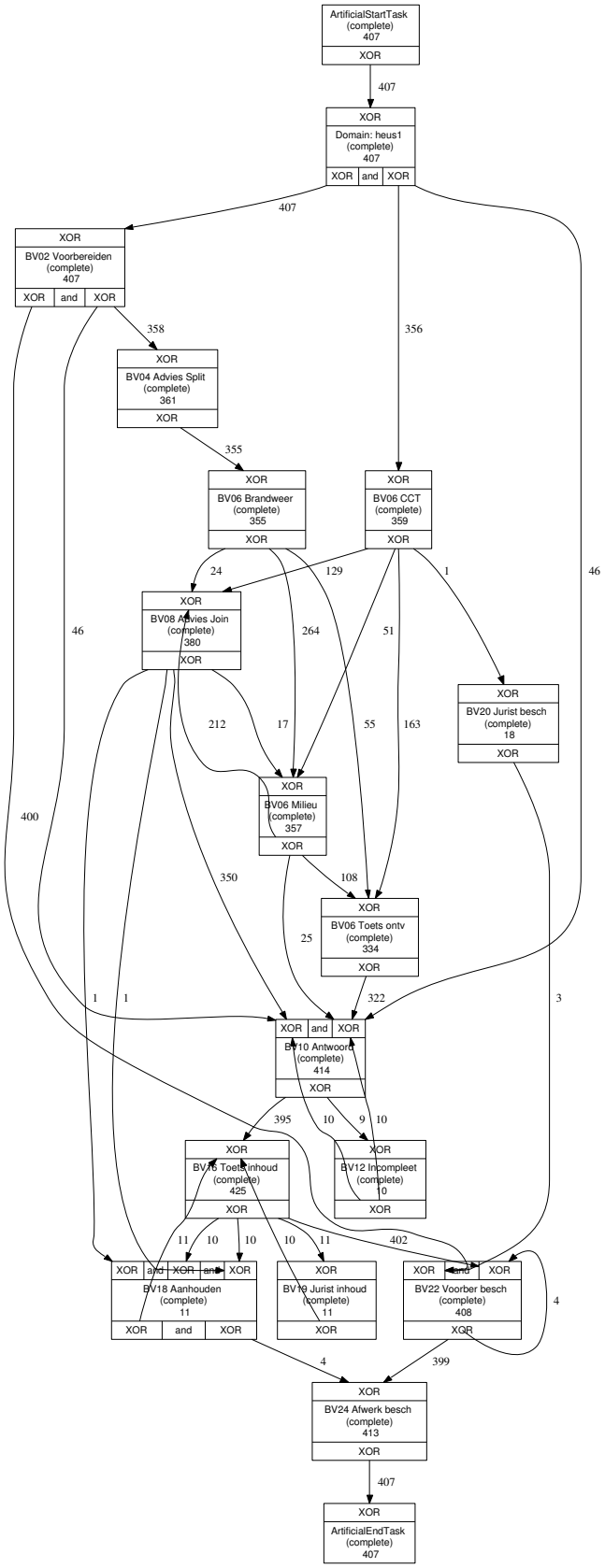


Figure 8.40: Mined model for Bouwvergunning - Replay of all traces.

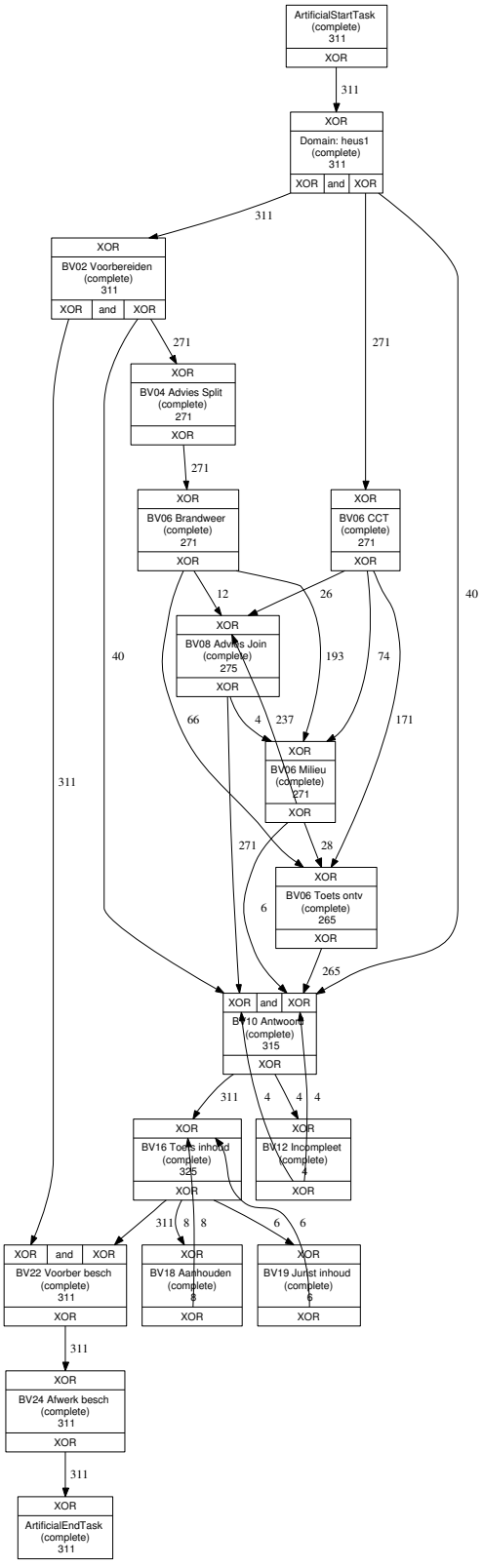


Figure 8.41: Bouwvergunning - 1% arc pruning over the mined model in Figure 8.40.

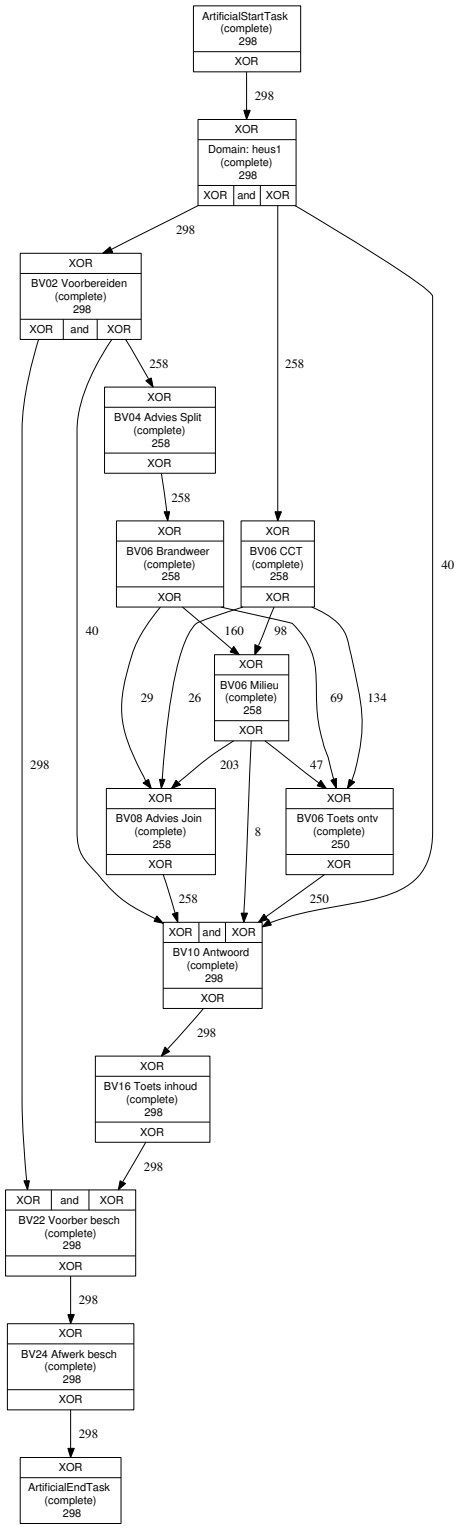


Figure 8.42: Bouwvergunning - 5% arc pruning over the mined model in Figure 8.40.

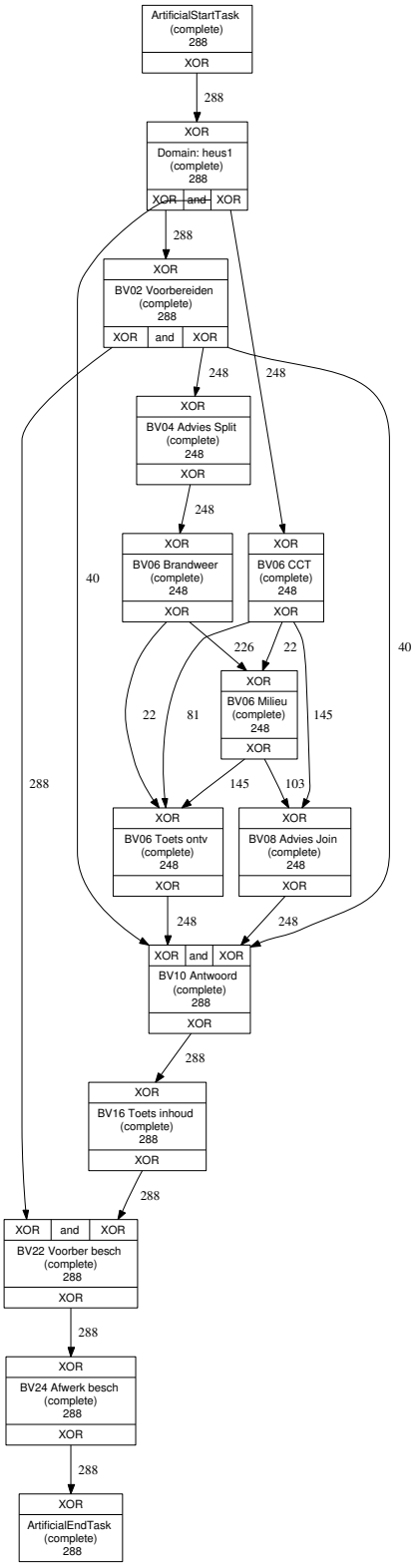


Figure 8.43: Bouwvergunning - 10% arc pruning over the mined model in Figure 8.40.

- The most frequent paths reported in Table 8.5 are all possible in the mined models. For instance, consider the most frequent path for process Bezwaar. Note that this path is non-compliant with the original model (cf. Figure 8.23). However, the mined model (cf. Figure 8.33) correctly captures the length-two loop situation between the activities "BZ14 Voorstel" and "BZ16 Wacht Besluit". Furthermore, the second most frequent path for Bezwaar is also captured by the mined model.
- The mined model for the process *Afschriften* (cf. Figure 8.30) better describes the behavior in the log than the original model (cf. Figure 8.21). The reason is that the mined model does not include the activities of the original model that were not executed. In other words, the mined model does not contain the activities "AG05 Wacht betrokken" and "AG09 Wacht afnemer" that are in the original model. Note that these activities were never executed during the log replay in Figure 8.21 (i.e., activity usage = 0 and arc usage = 0). Furthermore, the mined model is a substructure (or sub-net) of the original model. This can be easily seen by visually comparing the original model in Figure 8.21 with the mined model in Figure 8.30. However, this is also indicated by the values of the metrics $PF_{complete}$, S_P and B_P. This is the case because (i) both the original and mined models are 100% complete ($PF_{complete} = 1$), (ii) the mined model is precise ($B_P = 1$) and (ii) all the causality relations that are in the mined model are also in the original model ($S_P = 1$).
- The mined model for the process *Bezwaar* (cf. Figure 8.33) can correctly parse all the cases in the log (cf. Table 8.6), while the original model (cf. Figure 8.23) correctly parses only 51% of the cases in this same log. Note that the behavioral precision/recall metrics indicate that these two models are very similar from a behavioral point of view (both metrics are approximately 80%). Besides, the structural precision/recall metrics (cf. Table 8.7) indicate that the mined model contains many of the causality relations that are in the original model, but not all ($S_R = 0.92$). As an illustration, notice that the mined model (cf. Figure 8.33) contains only 2 out of the 7 input connections of the activity "BZ0 Intrekken" in the original model (cf. Figure 8.23). Furthermore, the mined model also has many connections that are not in the original model ($S_P = 0.73$).
- The mined model for BezwaarWOZ can correctly parse 98% of the log traces (cf. Table 8.6) while the original model correctly parses only 46% of the traces (cf. Table 8.4). The analysis metrics values in Table 8.7 point out that the mined model has less extra behavior that is not in

the original model ($B_P = 0.85$), than the other way around ($B_R = 0.73$). Furthermore, seen from the structural viewpoint, the precision and recall are similar ($S_P = 0.80$ and $S_R = 0.81$). By looking at the original and mined models for BezwaarWOZ (cf. the respective figures 8.25 and 8.36), we noticed that the mined model: (i) does not contain the low-frequent activities "OZ10 Horen", "OZ14 Plan. Taxeren" and "OZ18 Uitspr. wacht"; and (ii) captures frequently used connections that are not in the original model. For instance, note the connections from (i) activity "OZ06 Stop vordering" to "OZ12 Hertaxeren" and (ii) activity "OZ02 Voorbereiden" to "OZ08 Beoordelen" in Figure 8.36. These two connections are used respectively 133 and 114 times.

- Bouwvergunning is the only process for which fewer traces in the log are compliant with the mined model (77%) than with the original one (80%). The main difference between the structure of Bouwvergunning to the other three processes - Afschriften, Bezwaar and BezwaarWOZ - is the presence of a four-branch parallel construct (cf. original model in Figure 8.27) starting at the AND-split activity "BV04 Advies Split" and finishing at the AND-join activity "BV08 Advies Join". The experiments in Chapter 4 and the blind experiments in Section 8.2 already revealed that parallel constructs with more than two branches are difficult to mine. The result for Bouwvergunning reinforces this observation. As the reader notices from the metric values in Table 8.1, the mined model allows for more extra behavior than the original model ($B_P = 0.76$ but $B_R = 0.85$), and it has fewer extra connections than the mined model ($S_P = 0.62$ and $S_R = 0.59$). The mined model for Bouwvergunning is in Figure 8.40. Note that the activity "BV06 CCT" is in parallel with the activity "BV04 Advies Split". Although it is true that in some of the 20% non-compliant cases activity "BV06 CCT" was indeed executed before the activity "BV04 Advies Split", this is not the behavior observed in the remaining 80% compliant cases. Therefore, the GA should have captured the execution of "BV06 CCT" after "BV04 Advies Split". However, although the mined model has problems, it is interesting to see that it shows that the activity "BV04 Advies Split" was more frequently directly followed by the activity "BV06 Brandweer" than by the other three activities in this parallel construct. In fact, from the 284 compliant cases in which the activity "BV04 Advies Split" was executed (cf. activity usage in Figure 8.26), "BV06 Brandweer" was the next activity to be executed in 232 (81,6%) of these cases. This indicates that the fitness was in-

deed guiding the search towards an individual that exhibits the most frequent interleaving situations in the log. As final remark, we emphasize that the remaining net structure of Bouwvergunning was correctly mined. For instance, have a look at the pruned model in Figure 8.41 where the length-one loops are mined correctly.

The results for the case study show that the GA performs remarkably well when the nets do not have AND-split/join constructs. When they have AND-split/join constructs, the GA has a tendency to mine a model that portraits the most frequent interleaving situations between the tasks in the log.

8.3.3 Reflections

The reported results for the log replay (cf. Subsection 8.3.1) and the process re-discovery (cf. Subsection 8.3.2) were shown to the people responsible for these processes in the municipality. Since they work with a workflow system, we were curious to understand how the deviations have been possible. Recall that workflow management systems typically prescribe a model and forces users to execute the processes accordingly. Thus, in principle, all cases should comply with the deployed models.

After discussing with them, two possible sources of deviations came out: *temporary system misconfiguration* and *end-user explicit requests to change the state of a case*. Temporary *system misconfiguration* can be considered as "bugs" in deployed models. This situation happened for the process Bouwvergunning. When this process was initially configured, the AND-join task "BV08 Advies Join" could be executed whenever 3 out of the 4 parallel branches have been executed. The correct configuration should require all 4 tasks in the branches to execute, before the task "BV08 Advies Join" could be executed. The system administrator inspected (using the Conformance Checker) the cases that did not comply with the tasks in the parallel construct and confirmed that these cases were from a date before the "bug fix" had taken place. The other points of discrepancies in the Bouwvergunning process and in the other processes result from an *explicit change of the state of the case by the system administrator*. These changes happened, for instance, because the users of the system needed to re-edit an already completed case, or needed to jump to other tasks in the process (and the deployed model did not allow for this). It was very interesting to see that the system administrator could recognize some of the deviations captured in the mined models. For instance, we heard comments like "For the process Bezwaar, I can still remember when I had to change cases so that the task "BZ14 Voorstel" could be executed after the task "BZ16 Wacht Besluit"".

In short, the responsible people in the municipality found that the mined models were very insightful to give a *compact view* of how the processes are actually being executed. Together with the information about the arc usage, the deviations detected in the mined model indicate if it may be worth to update the deployed model, or if the deviations were rather infrequent and should not be incorporated in the deployed model. Additionally, the log replay allows for the detailed inspection of both discrepant and compliant behavior. Note that good and bad practices can be detected by replaying the log. Bad practices should be avoided, good practices should be made common knowledge in the organization. Either way, both types of information are important when using process mining techniques to improve deployed models.

8.4 Summary

This chapter described (i) the way models were selected for the experiments, (ii) how the logs for the experiments with our genetic approach were created, (iii) the results of the GA for the blind experiments, and (iv) the insights about the usefulness of process mining techniques in a real-life situation (the case study).

The logs for the experiments with known models and for the blind experiments were generated via simulation. The logs for the case study were provided by a Dutch municipality. Two ProM$_{import}$ converters - namely, CPN Tools and Eastman - were used to convert the logs to the MXML files format accepted by ProM.

The results for the blind experiments reinforced that the GA has problems to mine models with parallel constructs with more than two concurrent branches. Additionally, the blind experiments highlighted that the unbalance of AND-split/join points hinders even more the correct discovery of models by the GA. The main reason in both cases is that the GA "gets a bit lost" while guiding the search towards individuals that show the most frequent interleaving situations detected in the log. Note that highly concurrent systems allow for many interleavings and typically the distribution over the various possible interleavings is unbalanced. As a result, process mining becomes more difficult.

The case study confirmed the usefulness of process mining techniques to perform a *delta analysis* for deployed models. Even though the Dutch municipality uses a workflow management system (and, in principle, deviations from the prescribed behavior in the models are ruled out), the *Conformance Checker* and *Genetic Algorithm* plug-ins detected that there are differences between the behavior modelled in the deployed models and the actual ex-

ecution of these models. In the Dutch municipality situation, the detected differences were related to temporary system misconfiguration and manual interventions of the system administrator to change the state of some cases. The mined models showed that some of these manual interventions were quite frequent. This prompted the personnel in the municipality to consider the update of their deployed models to contemplate these frequent interventions.

Chapter 9

Conclusion

This chapter concludes the thesis. Section 9.1 summarizes the main contributions of the genetic process mining approach described in this thesis. Section 9.2 discusses the limitations of this approach and possible future work.

9.1 Contributions

As indicated in Chapter 1, the main aim of this thesis is to develop a control-flow process mining algorithm that can discover all the common control-flow structures (i.e. sequences, choices, parallelism, loops and *non-free-choices*, *invisible tasks* and *duplicate tasks*) while being *robust to noisy logs*. As a result, two genetic algorithms (GA and DGA) have been developed. These algorithms have been implemented as plug-ins of the ProM framework. To test these algorithms, synthetic logs were generated via simulation. To generate the synthetic logs, a common framework based on existing tools was developed. Furthermore, because different (mined) models can correctly portrait the behavior in the log, analysis metrics that go beyond the pure structural comparison of the mined models with the original models were also defined. The genetic algorithms and the analysis metrics constitute the main contributions of this thesis. In the following we elaborate on the contributions.

9.1.1 Genetic Algorithms

Two genetic algorithms have been defined: GA and DGA. The GA is a genetic algorithm that can mine all structural constructs, except for duplicate tasks. The DGA is an extension of the GA that is able to also discover duplicate tasks. As the results for the experiments reported in chapters 4, 5

and 6 indicate, both algorithms are robust to noise because they benefit the mining of models that correctly portrait the *most frequent behavior* in the log. For both algorithms, three core elements were defined:

- Internal representation: Individuals are represented as *causal matrices* (cf. Section 4.1 and 5.1). Causal matrices contain (i) a set of activities, (ii) a causality relation to set the dependencies between the activities, (iii) input/output condition functions to model the XOR/AND-split/join points, and (iv) a labelling function that associates tasks in the log to activities in the individuals. The main strength of causal matrices is that they support the modelling of all common control-flow constructs in process models. Besides, causal matrices can be mapped to Petri nets and, therefore, the mined models can benefit from the analysis techniques already provided to Petri nets. Note that the mapping heavily uses the so-called silent transitions. This way advanced constructs such as skipping can be dealt with in a more direct manner.
- Fitness measure: The fitness measure assesses the quality of individuals by replaying the log traces into these individuals. The main lesson learned for this replaying process is that the use of a *continuous parsing semantics* is better than the use of a blocking parsing semantics (cf. Subsection 4.2.1). In the blocking parsing semantics, the parsing process stops as soon as tasks in the log cannot be replayed in the individual. Alternatively, the continuous parsing semantics proceeds even when problems occur, but these problems are registered and considered while calculating the fitness value of a given individual. The fitness measure guides the search towards individuals that are *complete*, *precise* and *folded* (cf. Section 4.2 and 5.2). An individual is *complete* when it can replay all the traces of a log without having any problems during the parsing. Thus, to make sure that the fitness guides the search towards individuals that correctly capture the most frequent behavior in the log, the completeness requirement is based on the proportional amount (with respect to the most frequent behavior in the log) of tasks that can be correctly replayed by an individual (cf. Definition 21). An individual is *precise* when it does not allow for much more behavior than what can be derived from the log. Therefore, the preciseness requirement is based on the amount of activities that are simultaneously enabled while replaying the log (cf. Definition 22). The purpose of this requirement is to punish individuals that tend to be over-general. An individual is *folded* when none of its duplicates have input/output elements in common (cf. Definition 30). The folding requirement is only based on the structure of an individual. This re-

quirement punishes the individuals that have too many duplicates and, therefore, tend to be over-specific.

- Genetic operators: Crossover and mutation are the two genetic operators used in the GA and the DGA (cf. Section 4.3 and 5.3). The core concept of both operators is that the *causality relations* are the genetic material to be manipulated. The crossover operator works by exchanging causality relations between the input/output condition functions of duplicates in two individuals. The mutation operator works by inserting, replacing or removing causality relations from input/output condition functions of an individual's activities.

The experiments in Chapter 4 showed that the use of heuristics to set the causality relations while building individuals for the initial population speeds up the discovering process. Additionally, the results for the case study (cf. Chapter 8) showed that the genetic approach defined in this thesis is also applicable to real-life logs.

9.1.2 Analysis Metrics

As explained in Chapter 4 and 5, seven analysis metrics have been developed to quantify how complete, precise and folded the mined models are. These metrics are: the partial fitness for the completeness requirement ($PF_{complete}$) in Definition 21, the behavioral precision (B_P) and recall (B_R) in Definition 25 and 26, the structural precision (S_P) and recall (S_R) in Definition 32 and 33, and the duplicates precision (D_P) and recall (D_R) in Definition 34 and 35. From all these metrics, the more elaborate ones are the behavioral precision and recall. These metrics quantify how much behavior two models have in common while parsing an event log. We had to develop these analysis metrics because two individuals could model the same behavior in the log, but have completely different structures. Furthermore, because it is unrealistic to assume that event logs are exhaustive (i.e. contain all the possible behavior that can be generated by original models), metrics that compare state spaces (e.g. converability graph comparison or branching bisimulation) were not applicable anymore (cf. Section 4.5.1). Thus, the defined metrics can detect differences in the individuals, but can also quantify how much behavior they have in common regardless of the log being exhaustive or not. The concepts captured by these analysis metrics are applicable to situations in which two models need to be compared with respect to some exemplary behavior. In other words, these analysis metrics are also useful beyond the scope of our genetic process mining approach.

9.1.3 ProM Plug-ins

All the algorithms described in this thesis have been implemented as plug-ins in the ProM framework (cf. Chapter 7). ProM is an open-source tool that is made available via www.processmining.org. In the context of this thesis, 17 ProM plug-ins have been developed. From these, we highlight (i) the *Genetic algorithm* and *Duplicate Tasks GA* mining plug-ins that are the respective implementations of the GA and the DGA algorithms, (ii) the analysis plug-in "Prune Arcs" that implements the post-pruning mechanism to "clean" (mined) models (cf. Chapter 6), and (iii) the analysis plug-ins that implement the seven analysis metrics (cf. Subsection 9.1.2) to quantify how similar two models are. All the implemented plug-ins are provided together with the ProM framework.

9.1.4 Common Framework to Build Synthetic Logs

To assess the quality of the mined models, simulation was used to create synthetic event logs. The idea was to mine these synthetic logs and compare how close the behavior of the mined models are with respect to the original models. To generate these logs, we have implemented a library for the CPN Tools software package. CPN Tools allows for the design and simulation of Colored Petri nets (CP-nets). The library we have developed supports the creation of event logs whenever CP-nets are simulated. These event logs can be converted to the format that the ProM framework accepts (the MXML format presented in Subsection 7.1.1) by executing a ProM$_{import}$ plug-in - also called CPN Tools - that we have developed (cf. Subsection 7.6.1) as well. Both the ML library and the ProM$_{import}$ plug-in are made available via www.processmining.org.

9.2 Limitations and Future Work

Although the GA and the DGA can mine structural constructs that other techniques cannot and are also robust to noise, they have a drawback that cannot be neglected: the computational time. The situation is more critical for the DGA than for the GA because the DGA often uses a bigger search space. Therefore, it needs more iterations (i.e. generations) to converge to good solutions (i.e. process models that are complete, precise and folded). In the following we elaborate more on the limitations of the genetic algorithms and give suggestions for possible future work.

9.2.1 Genetic Algorithms

As the results for the experiments and case study in Chapter 4, 5 and 8 suggest, the genetic algorithms have more difficulties to mine models with constructs that allow for many interleaving situations. This is related to the fact that the fitness measure of these algorithms is tailored to benefit the individuals that correctly capture the most frequent behavior in the log. Therefore, as a side-effect, the models tend to correctly capture only the most frequent interleavings in the log. Examples of difficult constructs to mine are: (i) parallel constructs with many (say more than five) branches, (ii) unbalanced AND-slit/join points, especially when combined with long parallel branches, and (iii) more than three length-two loops connected to the same point. Again, the situation is worse for the DGA than for the GA because the interleavings tend to lead to the inference of more duplicates per task and, consequently, the DGA requires more iterations to converge to good solutions than the GA.

Related to duplicate tasks, in Chapter 5 we explained that the DGA can only capture models in which the duplicates do not share input/output elements. A possible future work would be to remove this constraint. However, perhaps this will affect even more the performance of the algorithm since bigger search spaces will be possible and it will become trickier to detect over-specializations. Another possible future work is to use heuristics in other steps of the genetic algorithms. In this thesis, heuristics have only been used to build the individuals in the initial population. These heuristics helped in setting the causality relations for these initial individuals and, in case of the DGA, in determining the amount of duplicates per task in the log. However, perhaps it would be worthwhile to use heuristics while performing the genetic operators as well. For instance, for the crossover operator, heuristics based on the problems encountered during the log replay (continuous semantics parsing) could be used to select the crossover point. In the current situation, duplicates are randomly chosen to crossover, regardless of how well they could be parsed during the log replay. Another possible extension for the crossover operator is to lift the crossover point from the duplicate level to a substructure level. In other words, instead of swapping the input/output elements of duplicates in two individual, the crossover operator could swap entire subparts of these individuals. This process could speed up the search process. In a similar way, heuristics could also be used to "guide" the mutation operator. For instance, in the current situation, one of the actions performed by the mutation operator is the random inclusion of causality relations in the input/output condition functions of activities in individuals. Perhaps heuristics to determine the set of causality relations

that seem to be appropriate to be inserted in individuals could be developed. Note that it does not make sense to insert a causality relation (dependency) in-between activities whose labels (tasks) never appear in a same trace in the log. Incorporating these heuristics may result in both better mined models and shorter computational time per run. Finally, future research could aim at extending the causal matrix to also support activities with an arbitrary OR-split/join semantics. The current representation only works with AND/XOR-split/join semantics.

9.2.2 Experiments

Although the experiments conducted in the context of this thesis are enough to give an indication of the quality of the mined models returned by the GA and DGA, there is space for more experimentation. For instance, for the noisy logs experiments, we have generated a single noisy log per net. A more rigorous setting would have used more logs per nets, generated for different seeds. Thus, we propose to test the genetic algorithm in a wide variety of experimental settings. First of all, start with smaller population sizes. The results for the blind-experiments and the case study suggest that smaller population sizes that run for more iterations give good results. However, the limits of this were not explored. Furthermore, we have not experimented with different settings for the κ and γ constants of the fitness measure. In fact, we have made some small experiments to choose the values we used during the experiments reported in this thesis. However, more extensive experimentations with different values would be interesting to give more insight on the proper calibration of these two parameters.

9.2.3 Process Mining Benchmark

The creation of a common set of event logs to test and compare the different process mining techniques seems to be vital for progressing process mining research. The process mining community should aim at a comprehensive set of benchmark examples. In this thesis, we have generated logs from models that have been derived from related work (especially from Herbst's work). However, a rigorous comparison was not possible because, among others, the logs used to test our approach are not the same ones used by the other related approaches. Only some of the original models that were simulated to create these logs are the same. It would be nice to have a common repository to (i) store event logs, (ii) describe the main characteristics of the process models used to create these logs and also provide these models, and (iii) show results of the process mining algorithms that have already being evaluated for these

logs. These results could provide information about the quality of the mined models. For instance, they could report the values for the analysis metrics used in this thesis, the structural constructs that the mined models contain, etc.

Appendix A

Causal Matrix: Mapping Back-and-Forth to Petri Nets

In this appendix we relate the causal matrix to Petri nets. We first map Petri nets (in particular WF-nets) onto the notation used by our genetic algorithm. Then we consider the mapping of the causal matrix onto Petri nets. First some preliminaries in Petri Nets.

A.1 Preliminaries

This subsection introduces the basic *Petri net* terminology and notations, and also discusses concepts such as *WF-nets* and *soundness*.

The classical Petri net is a directed bipartite graph with two node types called *places* and *transitions*. The nodes are connected via directed *arcs*.

Definition 36 (Petri net). *A Petri net is a triple* (P, T, F)*:*

- P *is a finite set of places,*
- T *is a finite set of transitions* ($P \cap T = \emptyset$)*,*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation).*

A place p is called an *input place* of a transition t iff there exists a directed arc from p to t. Place p is called an *output place* of transition t iff there exists a directed arc from t to p. *For any relation/directed graph* $G \subseteq N \times N$ we define the preset $\bullet n = \{(m_1, m_2) \in G \mid n = m_2\}$ and postset $n\bullet = \{(m_1, m_2) \in G \mid n = m_1\}$ for any node $n \in N$. We use $\overset{G}{\bullet} n$ or $n \overset{G}{\bullet}$ to explicitly indicate the context G if needed. Based on the flow relation F we use this notation as follows. $\bullet t$ denotes the set of input places for a transition t. The notations $t\bullet$, $\bullet p$ and $p\bullet$ have similar meanings, e.g., $p\bullet$ is the set of transitions sharing

p as an input place. Note that we do not consider multiple arcs from one node to another.

At any time a place contains zero or more *tokens*, drawn as black dots. The *state*, often referred to as marking, is the distribution of tokens over places, i.e., $M \in P \to \mathbb{N}$. To compare states we define a partial ordering. For any two states M_1 and M_2, $M_1 \leq M_2$ iff for all $p \in P$: $M_1(p) \leq M_2(p)$.

The number of tokens may change during the execution of the net. Transitions are the active components in a Petri net: they change the state of the net according to the following *firing rule*:

(1) A transition t is said to be *enabled* iff each input place p of t contains at least one token.

(2) An enabled transition may *fire*. If transition t fires, then t *consumes* one token from each input place p of t and *produces* one token for each output place p of t.

Given a Petri net (P, T, F) and a state M_1, we have the standard notations for a transition t that is enabled in state M_1 and firing t in M_1 results in state M_2 (notation: $M_1 \xrightarrow{t} M_2$) and a firing sequence $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ leads from state M_1 to state M_n via a (possibly empty) set of intermediate states (notation: $M_1 \xrightarrow{\sigma} M_n$). A state M_n is called *reachable* from M_1 (notation $M_1 \xrightarrow{*} M_n$) iff there is a firing sequence σ such that $M_1 \xrightarrow{\sigma} M_n$. Note that the empty firing sequence is also allowed, i.e., $M_1 \xrightarrow{*} M_1$.

In this appendix, we will focus on a particular type of Petri nets called *WorkFlow nets* (WF-nets) [8, 9, 30, 48].

Definition 37 (WF-net). *A Petri net $PN = (P, T, F)$ is a WF-net (Workflow net) if and only if:*

(i) There is one source place $i \in P$ such that $\bullet i = \emptyset$.

(ii) There is one sink place $o \in P$ such that $o\bullet = \emptyset$.

(iii) Every node $x \in P \cup T$ is on a path from i to o.

A WF-net represents the life-cycle of a case that has some initial state represented by a token in the unique input place (i) and a desired final state represented by a token in the unique output place (o). The third requirement in Definition 37 has been added to avoid "dangling transitions and/or places". In the context of workflow models or business process models, transitions can be interpreted as *activities* or *tasks* and places can be interpreted as *conditions*. Although the term "WorkFlow net" suggests that the application is limited to workflow processes, the model has wide applicability, i.e., any process where each case has a life-cycle going from some initial state to some final state fits this basic model.

The three requirements stated in Definition 37 can be verified statically, i.e., they only relate to the structure of the Petri net. To characterize desirable dynamic properties, the notation of *soundness* has been defined [8, 9, 30, 48].

Definition 38 (Sound). *A procedure modeled by a WF-net $PN = (P, T, F)$ is sound if and only if:*

(i) For every state M reachable from state i, there exists a firing sequence leading from state M to state o. Formally: $\forall_M (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o)$.[1]

(ii) State o is the only state reachable from state i with at least one token in place o. Formally: $\forall_M (i \xrightarrow{*} M \ \wedge \ M \geq o) \Rightarrow (M = o)$.

(iii) There are no dead transitions in (PN, i). Formally: $\forall_{t \in T} \ \exists_{M,M'} \ i \xrightarrow{*} M \xrightarrow{t} M'$.

Note that the soundness property relates to the dynamics of a WF-net. The first requirement in Definition 38 states that starting from the initial state (state i), it is always possible to reach the state with one token in place o (state o). The second requirement states that the moment a token is put in place o, all the other places should be empty. The last requirement states that there are no dead transitions (activities) in the initial state i.

A.2 Mapping a Petri net onto a Causal Matrix

In this thesis, we use the concept of a *causal matrix* to represent an individual. Table A.1 and A.2 show two alternative visualizations. In this section, we first formalize the notion of a causal matrix. This formalization will be used to map a causal matrix onto a Petri net and vice-versa.

Definition 39 (Causal Matrix). *Let LS be a set of labels. A Causal Matrix is a tuple $CM = (A, C, I, O, Label)$, where*

- *A is a finite set of activities,*
- *$C \subseteq A \times A$ is the causality relation,*
- *$I : A \rightarrow \mathcal{P}(\mathcal{P}(A))$ is the input condition function,*[2]
- *$O : A \rightarrow \mathcal{P}(\mathcal{P}(A))$ is the output condition function,*
- *$Label \in A \rightarrow LS$ is a labeling function that maps each activity in A to a label in LS,*

[1] Note that there is an overloading of notation: the symbol i is used to denote both the *place* i and the *state* with only one token in place i.

[2] $\mathcal{P}(A)$ denotes the powerset of some set A.

	INPUT								
	true	*A*	*A*	*A*	*D*	*D*	$E \wedge F$	$B \vee C \vee G$	
$\rightarrow$	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*OUTPUT*
A	0	1	1	1	0	0	0	0	$B \vee C \vee D$
B	0	0	0	0	0	0	0	1	*H*
C	0	0	0	0	0	0	0	1	*H*
D	0	0	0	0	1	1	0	0	$E \wedge F$
E	0	0	0	0	0	0	1	0	*G*
F	0	0	0	0	0	0	1	0	*G*
G	0	0	0	0	0	0	0	1	*H*
H	0	0	0	0	0	0	0	0	*true*

Table A.1: A causal matrix is used for the internal representation of an individual.

ACTIVITY	*INPUT*	*OUTPUT*
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E\}, \{F\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{B, C, G\}\}$	$\{\}$

Table A.2: A more succinct encoding of the individual shown in Table A.1.

such that

- $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$,[3]
- $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$,
- $C \cup \{(a_o, a_i) \in A \times A \mid a_o \overset{C}{\bullet}= \emptyset \wedge \overset{C}{\bullet} a_i = \emptyset\}$ *is a strongly connected graph.*

The mapping of Table A.2 onto $CM = (A, C, I, O, Label)$ is straightforward (the latter two columns represent I and O). Note that C can be derived from both I and O. Its main purpose is to ensure consistency between I and O. For example, if a_1 has an output condition mentioning a_2, then a_2 has an input condition mentioning a_1 (and vice versa). This is enforced by the first two constraints. The last requirement has been added to avoid that the causal matrix can be partitioned in two independent parts or that nodes are not on a path from some source activity a_i to a sink activity a_o. Furthermore, we have removed the requirement for the labelling function (cf. Definition 18 in Chapter 4) because the mappings shown in this appendix work independently of the labelling function be injective or not.

The mapping from an arbitrary Petri net to its corresponding causal matrix illustrates the expressiveness of the internal format used for genetic mining. First, we give the definition of the mapping $\Pi_{PN \to CM}$.

Definition 40 ($\Pi_{PN \to CM}$). *Let $PN = (P, T, F, Label_{PN})$ be a Petri net. The mapping of PN is a tuple $\Pi_{PN \to CM}(PN) = (A, C, I, O, Label)$, where*

- $A = T$,
- $C = \{(t_1, t_2) \in T \times T \mid t_1 \bullet \cap \bullet t_2 \neq \emptyset\}$,
- $I : T \to \mathcal{P}(\mathcal{P}(T))$ *such that* $\forall_{t \in T}\ I(t) = \{\bullet p \mid p \in \bullet t\}$,
- $O : T \to \mathcal{P}(\mathcal{P}(T))$ *such that* $\forall_{t \in T}\ O(t) = \{p \bullet \mid p \in t \bullet\}$,
- $Label = Label_{PN}$.

Let PN be the Petri net shown in Figure A.1. It is easy to check that $\Pi_{PN \to CM}(PN)$ is indeed the causal matrix in Table A.1. However, there may be Petri nets PN for which $\Pi_{PN \to CM}(PN)$ is not a causal matrix. The following lemma shows that for the class of nets we are interested in, i.e., WF-nets, the requirement that there may not be two different places two different places with a same set of input and output transitions (i.e., $\nexists_{p_1, p_2 \in P}[\bullet p_1 = \bullet p_2 \wedge p_1 \bullet = p_2 \bullet]$) is sufficient to prove that $\Pi_{PN \to CM}(PN)$ represents a causal matrix as defined in Definition 39.

Lemma 1. *Let $PN = (P, T, F, Label_{PN})$ be a WF-net with no two places with the same input and output transitions, i.e., $\forall_{p_1, p_2 \in P}\ [(\bullet p_1 = \bullet p_2) \wedge (p_1 \bullet =$*

[3] $\bigcup I(a_2)$ is the union of the sets in set $I(a_2)$.

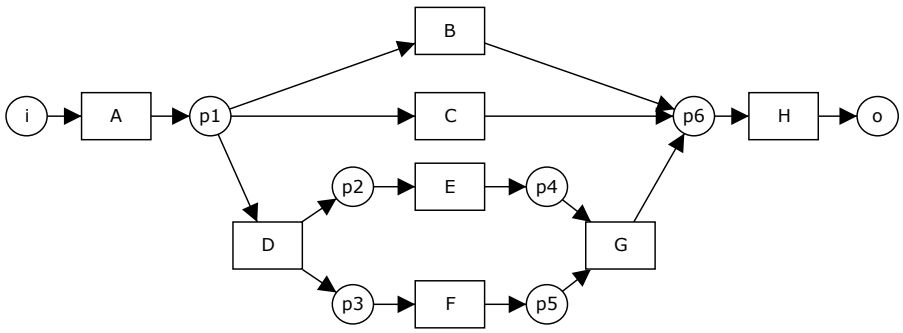


Figure A.1: Petri net that can be mapped to the causal matrix in Table A.1.

$p_2\bullet) \Rightarrow p_1 = p_2]$. $\Pi_{PN \to CM}(PN)$ *represents a causal matrix as defined in Definition 39.*

Proof. Let $\Pi_{PN \to CM} = (A, C, I, O, Label)$. Clearly, $A = T$ is a finite set, $C \subseteq A \times A$ and $Label = Label_{PN}$ is a function that maps each activity to a label. $I, O : A \to \mathcal{P}(\mathcal{P}(A))$ because $\forall_{p_1,p_2 \in P} [(\bullet p_1 = \bullet p_2) \wedge (p_1\bullet = p_2\bullet) \Rightarrow p_1 = p_2]$. $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$ because $a_1 \in \bigcup I'(a_2)$ if and only if $a_1 \bullet \cap \bullet a_2 \neq \emptyset$. Similarly, $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$. Finally, it is easy to verify that $C \cup \{(a_o, a_i) \in A \times A \mid a_o\bullet = \emptyset \wedge \bullet a_i = \emptyset\}$ is a strongly connected graph. □

The requirement $\forall_{p_1,p_2 \in P} [(\bullet p_1 = \bullet p_2) \wedge (p_1\bullet = p_2\bullet) \Rightarrow p_1 = p_2]$ is a direct result of the fact that the I and O condition functions map to a powerset of a powerset of the set of activities in A (i.e., $I, O : A \to \mathcal{P}(\mathcal{P}(A))$). If this requirement would not be there, the I/O condition functions would allow for *bags* of a powerset of activities in A. However, the added places would not impact the *behavior* of the net. Therefore, since the behavior of a mined model is not affected by this restriction, it has been added to reduce the search space of the genetic mining algorithm.

A.3 Mapping a Causal Matrix onto a Petri net

The mapping from a causal matrix onto a Petri net is more involved because we need to "discover places" and, as we will see, the causal matrix is slightly more expressive than classical Petri nets.[4] Let us first define the mapping.

[4]Expressiveness should not be interpreted in a formal sense but in the sense of convenience when manipulating process instances, e.g., crossover operations.

ACTIVITY	*INPUT*	*OUTPUT*
A	$\{\}$	$\{\{C, D\}\}$
B	$\{\}$	$\{\{D\}\}$
C	$\{\{A\}\}$	$\{\}$
D	$\{\{A, B\}\}$	$\{\}$

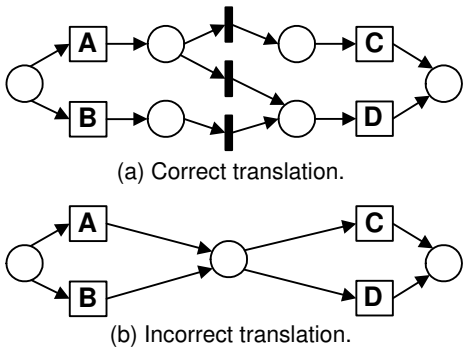


Figure A.2: A causal matrix (left) and two potential mappings onto Petri nets (right).

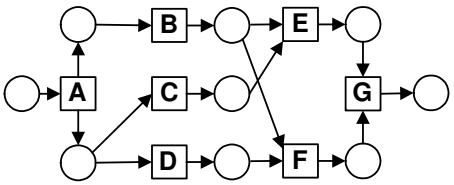
Definition 41 ($\Pi^N_{CM\to PN}$). *Let $CM = (A, C, I, O, Label)$ be a causal matrix. The Petri net mapping of CM is a tuple $\Pi^N_{CM\to PN} = (P, T, F, Label_{PN})$, where*

- $P = \{i, o\} \cup \{i_{t,s} \mid t \in A \ \wedge \ s \in I(t)\} \cup \{o_{t,s} \mid t \in A \ \wedge \ s \in O(t)\}$,
- $T = A \cup \{m_{t_1,t_2} \mid (t_1, t_2) \in C\}$,
- $F = \{(i, t) \mid t \in A \wedge \overset{C}{\bullet} t = \emptyset\} \cup \{(t, o) \mid t \in A \wedge t \overset{C}{\bullet} = \emptyset\} \cup \{(i_{t,s}, t) \mid t \in A \wedge s \in I(t)\} \cup \{(t, o_{t,s}) \mid t \in A \wedge s \in O(t)\} \cup \{(o_{t_1,s}, m_{t_1,t_2}) \mid (t_1, t_2) \in C \wedge s \in O(t_1) \wedge t_2 \in s\} \cup \{(m_{t_1,t_2}, i_{t_2,s}) \mid (t_1, t_2) \in C \wedge s \in I(t_2) \wedge t_1 \in s\}$,
- $\forall_{t\in T}, \ Label_{PN}(t) = \begin{cases} Label(t) & \text{if } t \in A, \\ \emptyset & \text{otherwise.} \end{cases}$

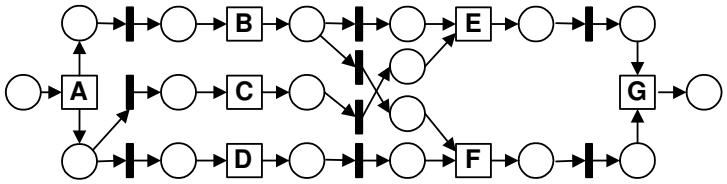
The mapping $\Pi^N_{CM\to PN}$ maps activities onto transitions and adds input places and output places to these transitions based on functions I and O. These places are local to one activity. To connect these local places, one transition m_{t_1,t_2} is added for every $(t_1, t_2) \in C$. Figure A.2 shows a causal matrix and the mapping $\Pi^N_{CM\to PN}$ (we have partially omitted place/transition names).

Figure A.2 shows two WF-nets illustrating the need for "silent transitions" of the form m_{t_1,t_2}. The dynamics of the WF-net shown in Figure A.2(a) is consistent with the causal matrix. If we try to remove the silent transitions, the best candidate seems to be the WF-net shown in Figure A.2(b). Although this is a sound WF-net capturing the behavior of the WF-net shown in Figure A.2(a), the mapping is *not* consistent with the causal matrix. Note that Figure A.2(b) allows for a firing sequence where B is followed by C. This does not make sense because $C \notin \bigcup O(B)$ and $B \notin \bigcup I(C)$. Therefore, we use the mapping given in Definition 41 to give Petri-net semantics to causal matrices.

ACTIVITY	*INPUT*	*OUTPUT*
A	$\{\}$	$\{\{B\},\{C,D\}\}$
B	$\{\{A\}\}$	$\{\{E,F\}\}$
C	$\{\{A\}\}$	$\{\{E\}\}$
D	$\{\{A\}\}$	$\{\{F\}$
E	$\{\{B\},\{C\}\}$	$\{\{G\}\}$
F	$\{\{B\},\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\},\{F\}\}$	$\{\}$



(a) Mapping without silent transitions.



(b) Mapping with silent transitions.

Figure A.3: Another causal matrix (left) and two potential mappings onto Petri nets (right).

It is easy to see that a causal matrix defines a WF-net. However, note that the WF-net does not need to be sound.

Lemma 2. *Let* $CM = (A, C, I, O, Label)$ *be a causal matrix.* $\Pi^N_{CM \to PN}(CM)$ *is a WF-net.*

Proof. It is easy to verify the three properties mentioned in Definition 37. Note that the "short-circuited" C is strongly connected and that each m_{t_1,t_2} transition makes a similar connection in the resulting Petri net. □

Figure A.3 shows that despite the fact that $\Pi^N_{CM \to PN}(CM)$ is a WF-net, the introduction of silent transitions may introduce a problem. Figure A.3(b) shows the WF-net based on Definition 41, i.e., the mapping with silent transitions. Clearly, Figure A.3(b) is not sound because there are two potential deadlocks, i.e., one of the input places of E is marked and one of the input places of F is marked but none of them is enabled. The reason for this is that the choices introduced by the silent transitions are not "coordinated" properly. If we simply remove the silent transitions, we obtain the WF-net shown in Figure A.3(a). This network is consistent with the causal matrix. This can easily be checked because applying the mapping $\Pi_{PN \to CM}$ defined in Definition 40 to this WF-net yields the original causal matrix shown in Figure A.3.

Figures A.2 and A.3 show a dilemma. Figure A.2 demonstrates that silent transitions are needed while Figure A.3 proves that silent transitions can be

harmful. There are two ways to address this problem taking the mapping of Definition 41 as a starting point.

First of all, we can use *relaxed soundness* [30] rather than soundness [8]. This implies that we only consider so-called sound firing sequences and thus avoid the two potential deadlocks in Figure A.3(b). See [30] for transforming a relaxed sound WF-net into a sound one.

Second, we can change the firing rule such that silent transitions can only fire if they actually enable a non-silent transition. The enabling rule for non-silent transitions is changed as follows: *a non-silent transition is enabled if each of its input places is marked or it is possible to mark all input places by just firing silent transitions*, i.e., silent transitions only fire when it is possible to enable a non-silent transition. Note that non-silent and silent transitions alternate and therefore it is easy to implement this semantics in a straightforward and localized manner.

In this thesis we use the second approach, i.e., a slightly changed enabling/-firing rule is used to specify the semantics of a causal matrix in terms of a WF-net. This semantics allows us also to define a notion of fitness required for the genetic algorithms. Using the Petri-net representation we can play the "token game" to see how each event trace in the log fits the individual represented by a causal matrix.

Appendix B

All Models Used During the Experiments with Known Models

This appendix has the nets that were used during the experiments with known models (cf. Section 8.1 and the experiments reported in chapters 4 to 6). For every net, we show the CPN model that was used to generate the logs, as well as the corresponding causal matrix (or heuristic net in the ProM terminology) for this net. Additionally, Table B.1 summarizes the constructs that the each net contains.

Net	Figure	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Structured Loop	Arbitrary Loop	Non-Local NFC	Invisible Tasks	Duplicates in Sequence	Duplicates in Parallel
a10skip	B.1	✓	✓	✓						✓		
a12	B.3	✓	✓	✓								
a5	B.5	✓	✓	✓	✓							
a6nfc	B.7	✓	✓	✓						✓		
a7	B.9	✓	✓	✓								
a8	B.11	✓	✓	✓								
al1	B.13	✓	✓	✓				✓				
al2	B.15	✓	✓	✓				✓				
betaSimplified	B.17	✓	✓						✓	✓	✓	
bn1	B.19	✓	✓									

Continued on next page

Net	Figure	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Structured Loop	Arbitrary Loop	Non-Local NFC	Invisible Tasks	Duplicates in Sequence	Duplicates in Parallel
bn2	B.21	✓	✓				✓			✓		
bn3	B.23	✓	✓				✓			✓		
choice	B.25	✓	✓									
driversLicense	B.27	✓	✓						✓			
flightCar	B.29	✓	✓	✓								✓
herbstFig3p4	B.31	✓	✓	✓			✓					
herbstFig5p1AND	B.33	✓		✓							✓	
herbstFig5p1OR	B.35	✓	✓								✓	
herbstFig5p19	B.37	✓	✓	✓						✓	✓	
herbstFig6p10	B.39	✓	✓	✓			✓			✓	✓	
herbstFig6p18	B.41	✓	✓		✓	✓	✓			✓		
herbstFig6p25	B.43	✓	✓			✓	✓			✓	✓	
herbstFig6p31	B.45	✓	✓								✓	
herbstFig6p33	B.47	✓	✓								✓	
herbstFig6p34	B.49	✓	✓	✓			✓			✓	✓	
herbstFig6p36	B.51	✓	✓						✓			
herbstFig6p37	B.53	✓		✓								
herbstFig6p38	B.55	✓		✓								✓
herbstFig6p39 nc	B.57	✓		✓						✓		✓
herbstFig6p41	B.59	✓	✓	✓								
herbstFig6p42 nc	B.61	✓	✓	✓						✓	✓	
herbstFig6p45	B.63	✓		✓								
herbstFig6p9	B.65	✓	✓							✓	✓	
l1l	B.67	✓	✓		✓							
l1lSkip	B.69	✓	✓	✓	✓					✓		
l2l	B.71	✓	✓			✓						
l2lOptional	B.73	✓	✓			✓						
l2lSkip	B.75	✓	✓			✓				✓		
parallel5	B.77	✓		✓								

Table B.1: Nets for the experiments with known models.

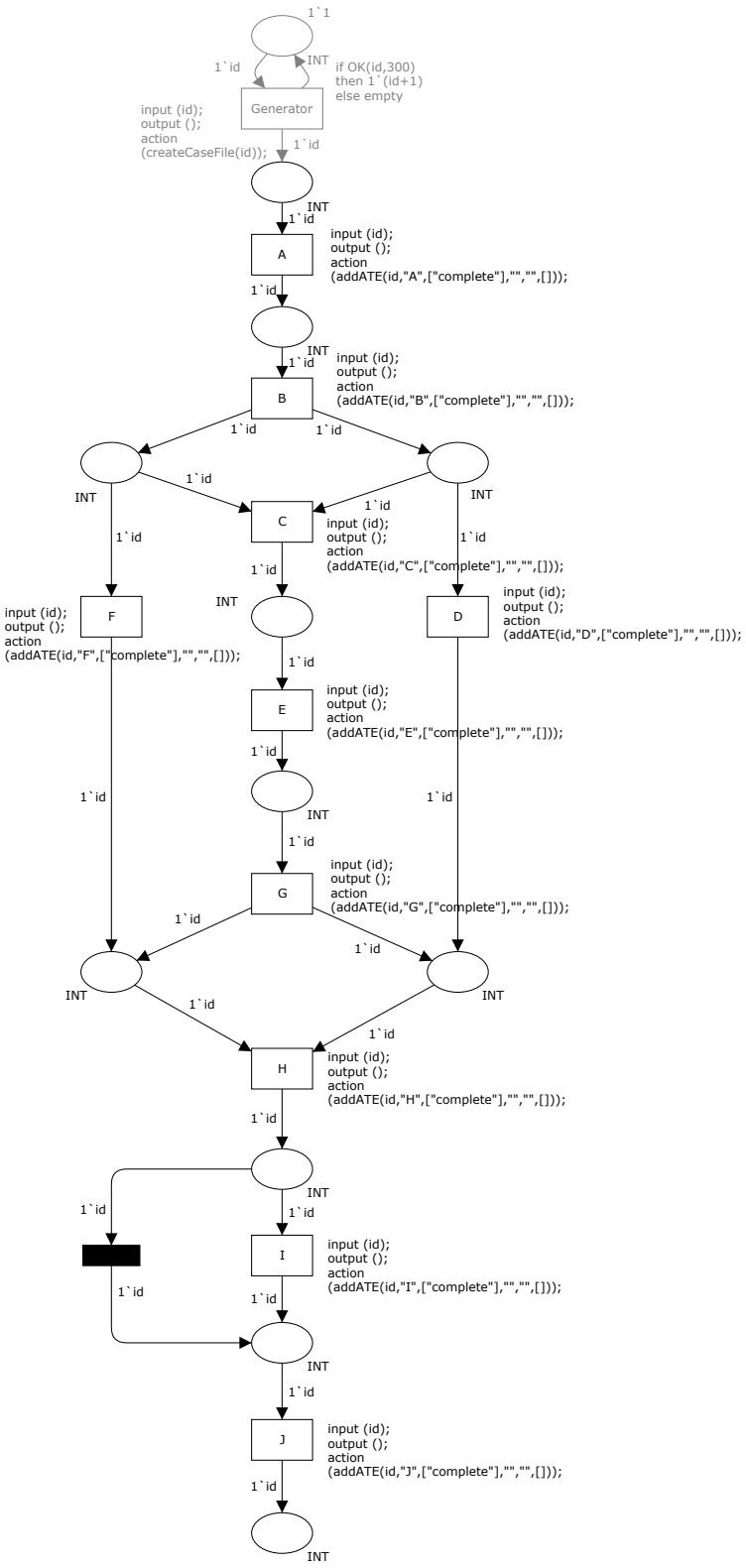


Figure B.1: CPN model for net a10Skip.

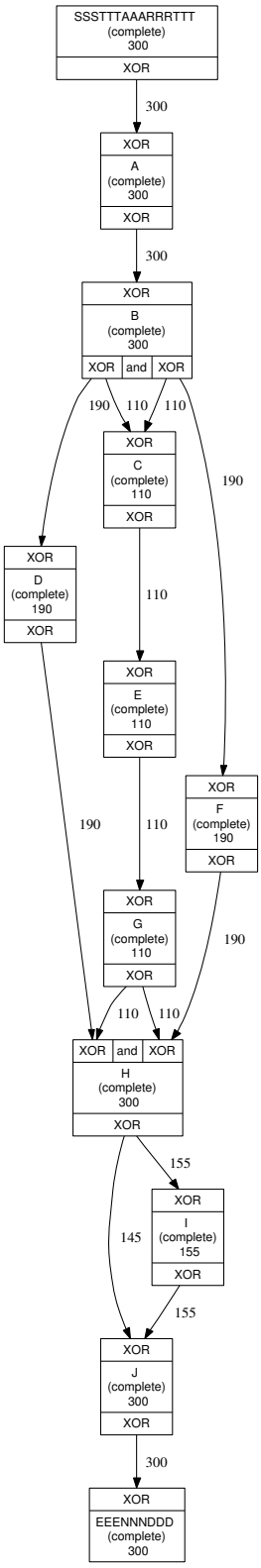


Figure B.2: Heuristic net for a10Skip.

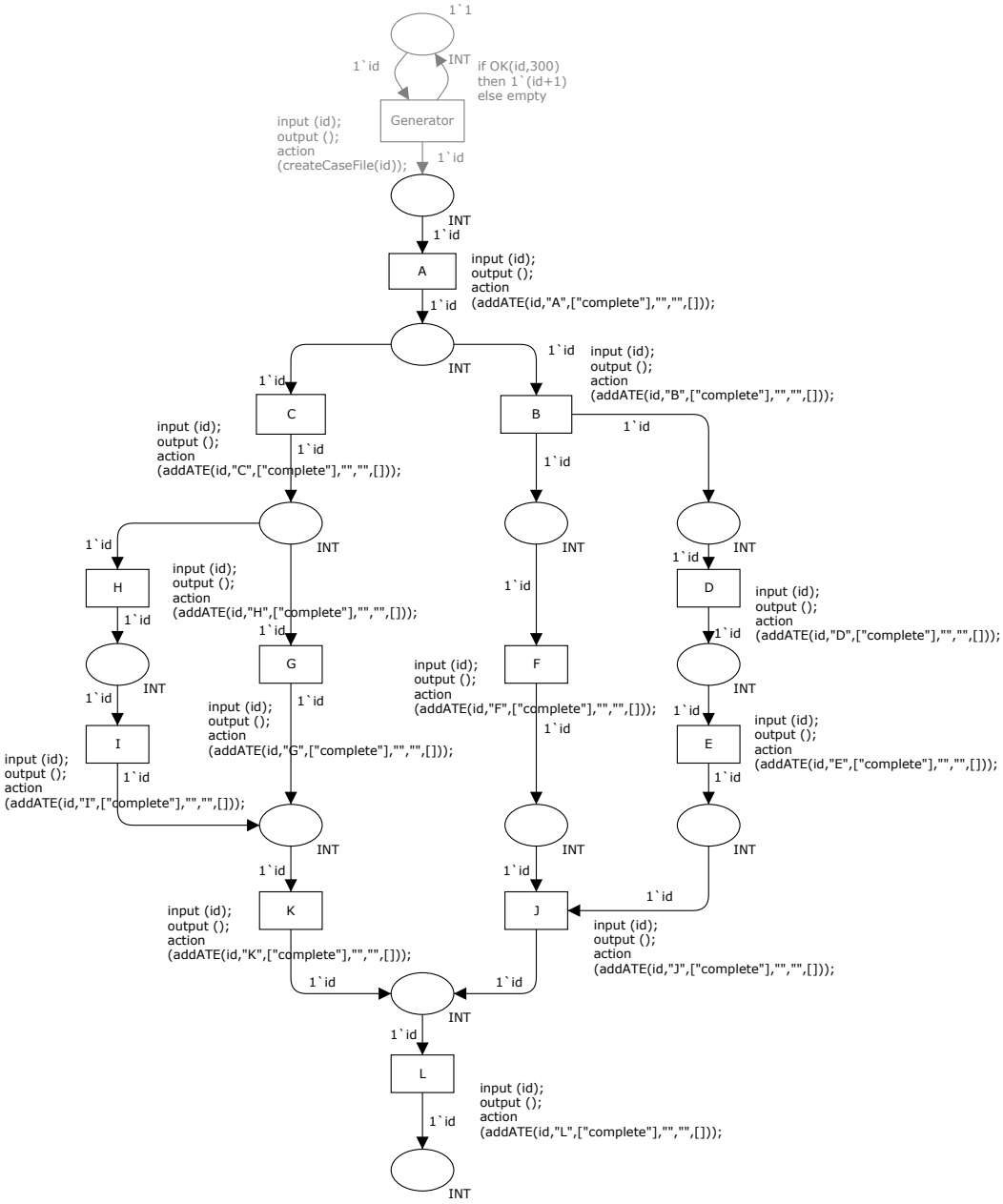


Figure B.3: CPN model for net a12.

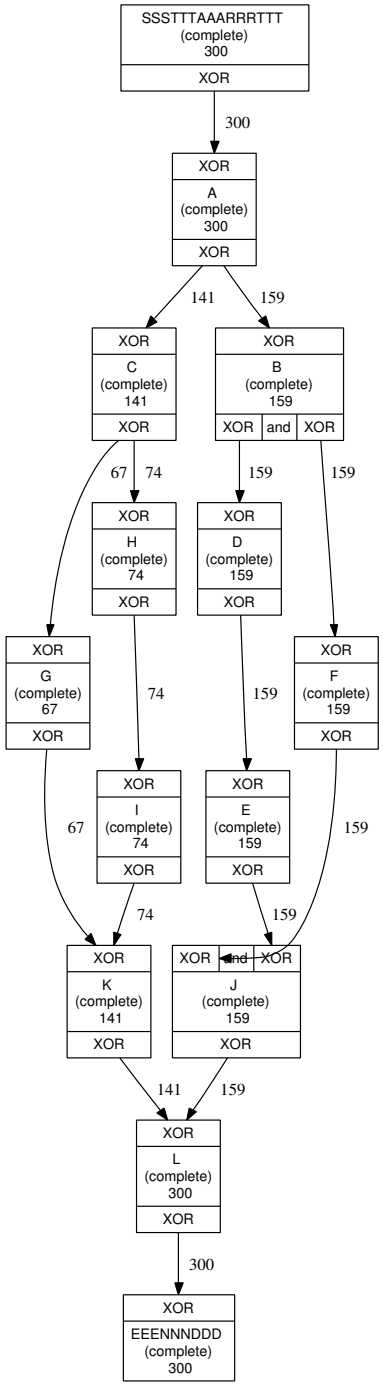


Figure B.4: Heuristic net for a12.

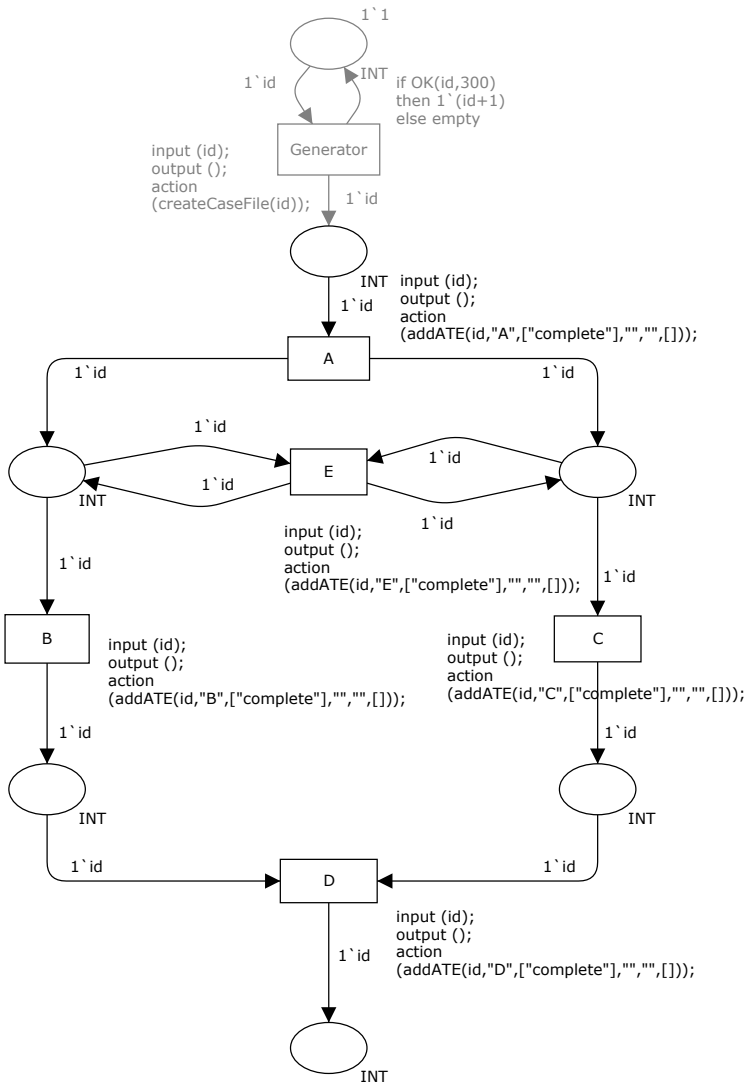


Figure B.5: CPN model for net a5.

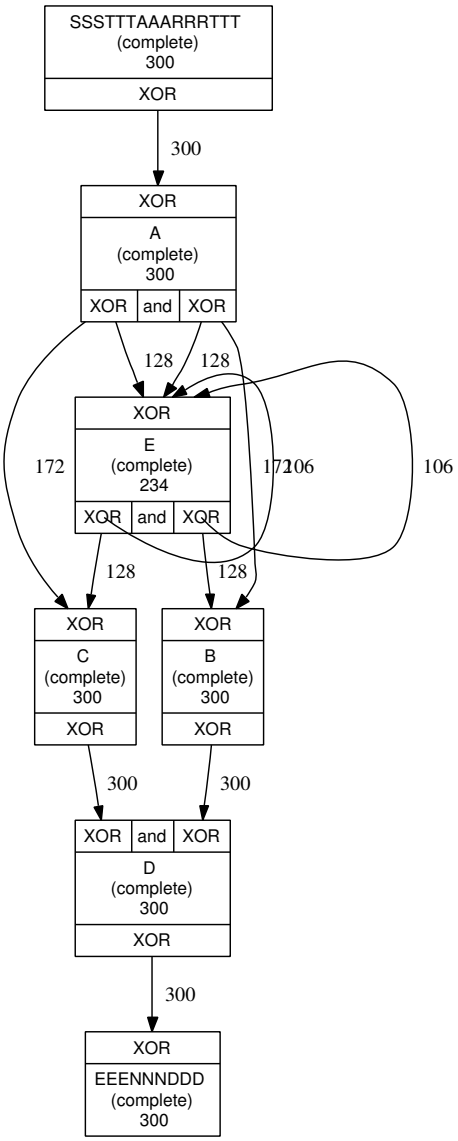


Figure B.6: Heuristic net for a5.

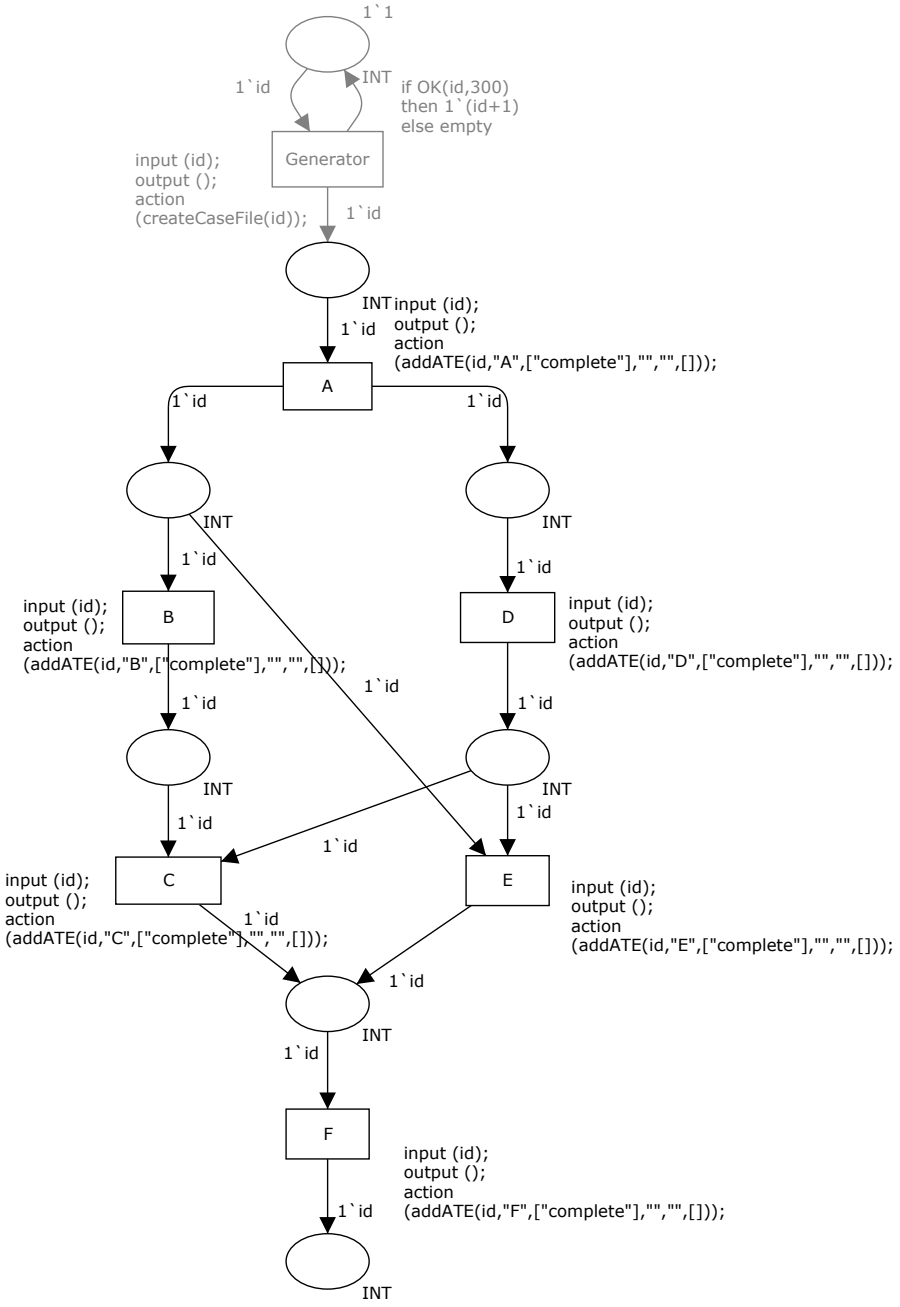


Figure B.7: CPN model for net a6nfc.

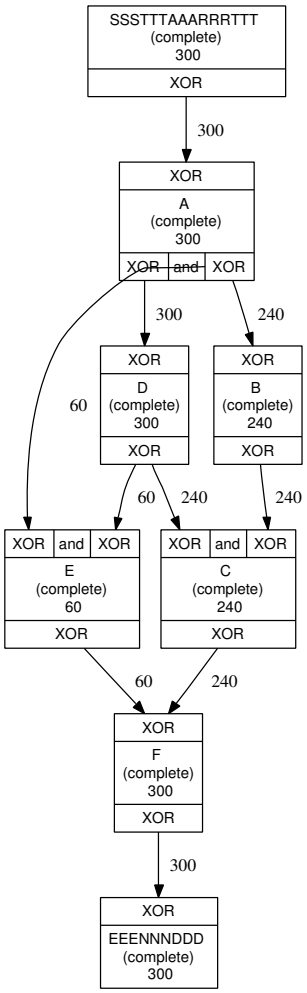


Figure B.8: Heuristic net for a6nfc.

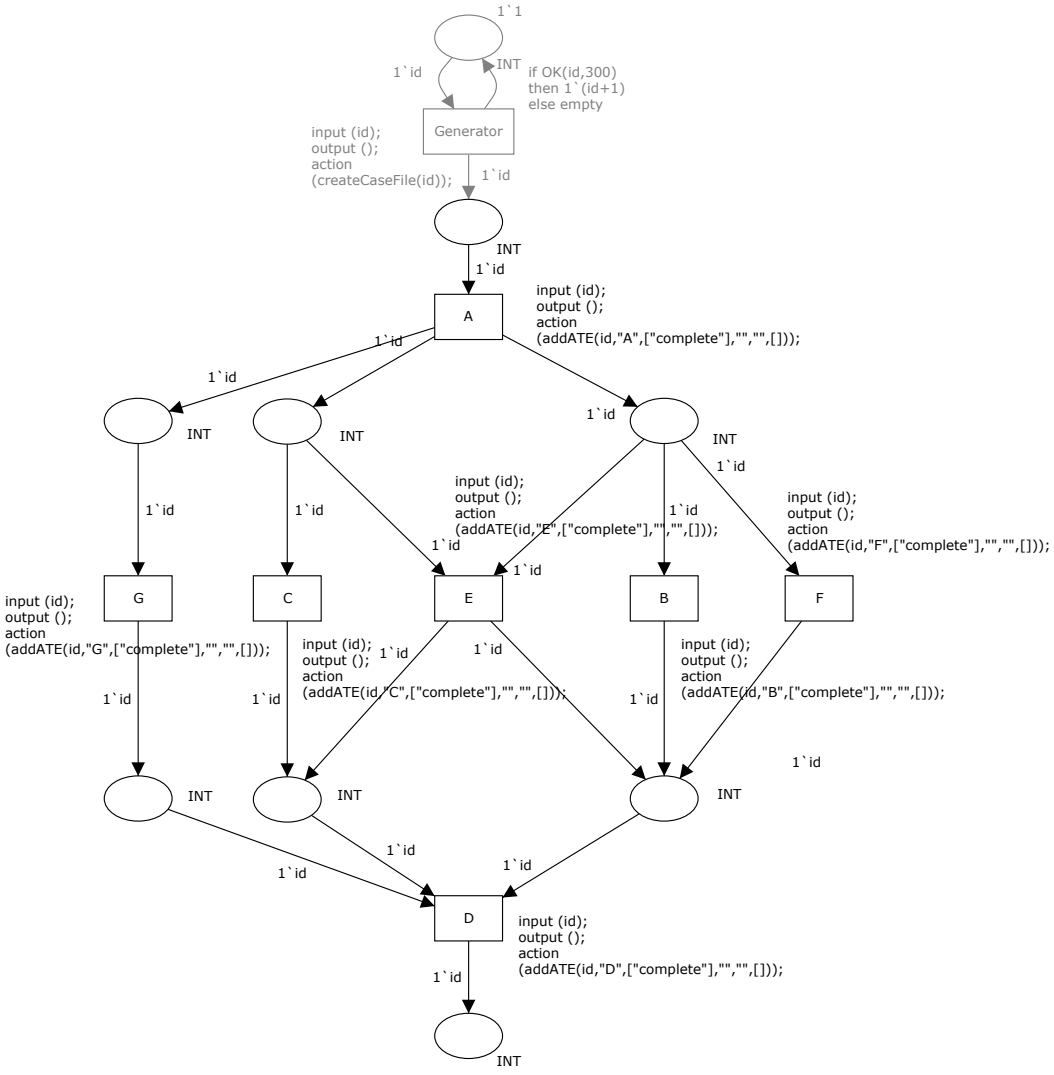


Figure B.9: CPN model for net a7.

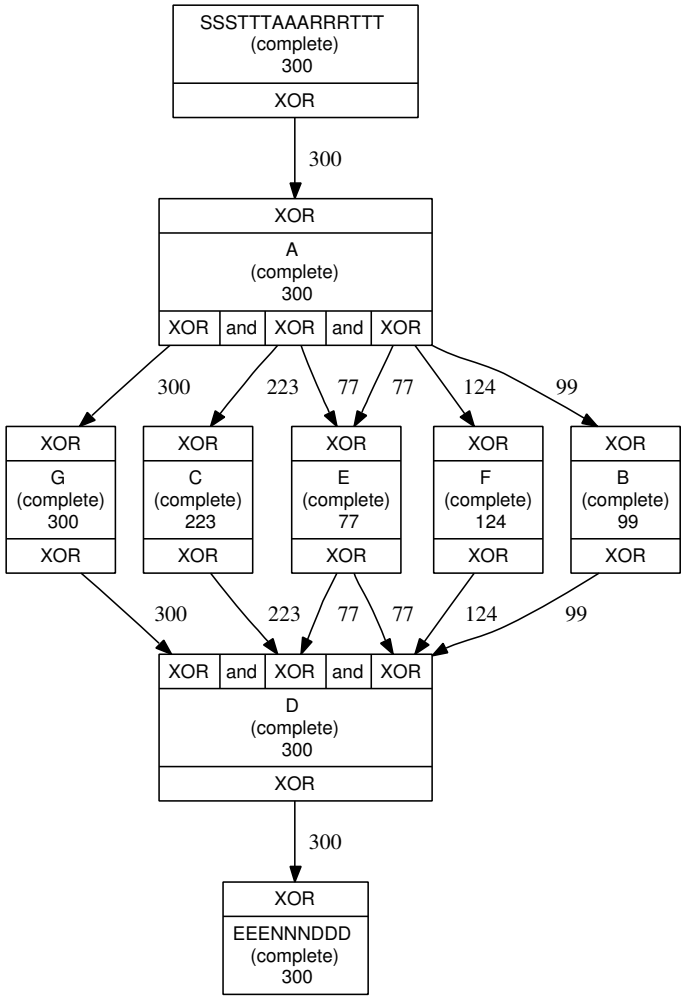


Figure B.10: Heuristic net for a7.

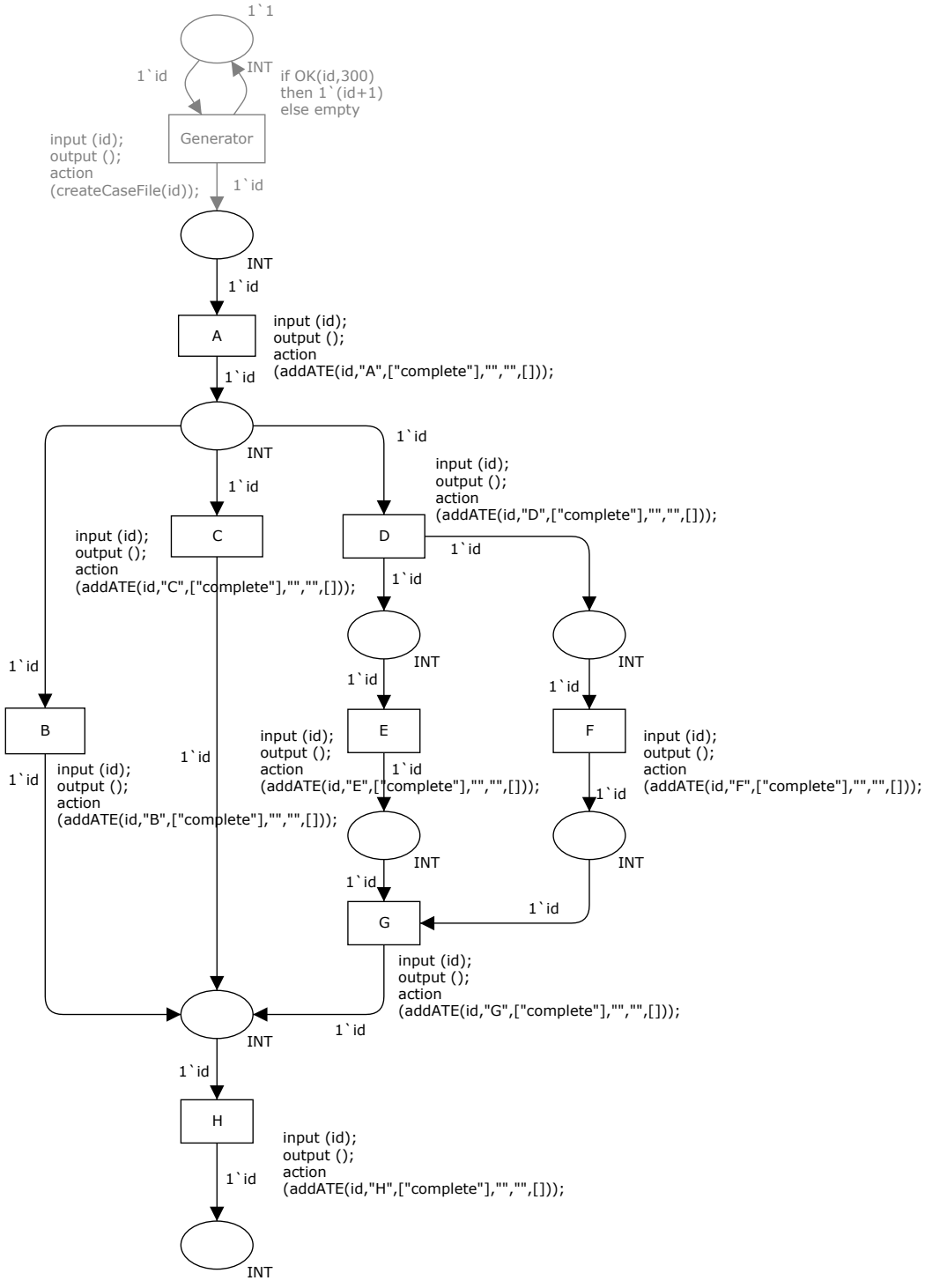


Figure B.11: CPN model for net a8.

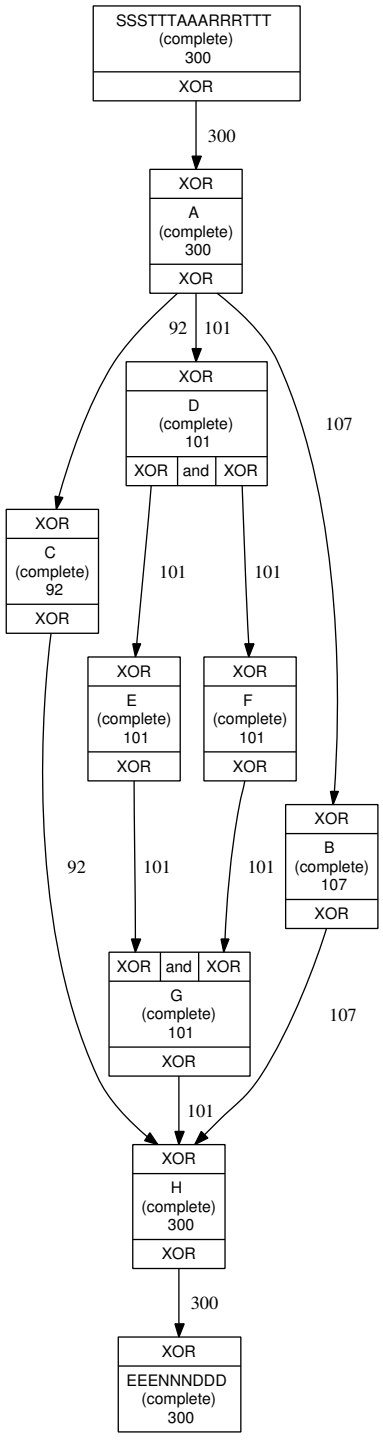


Figure B.12: Heuristic net for a8.

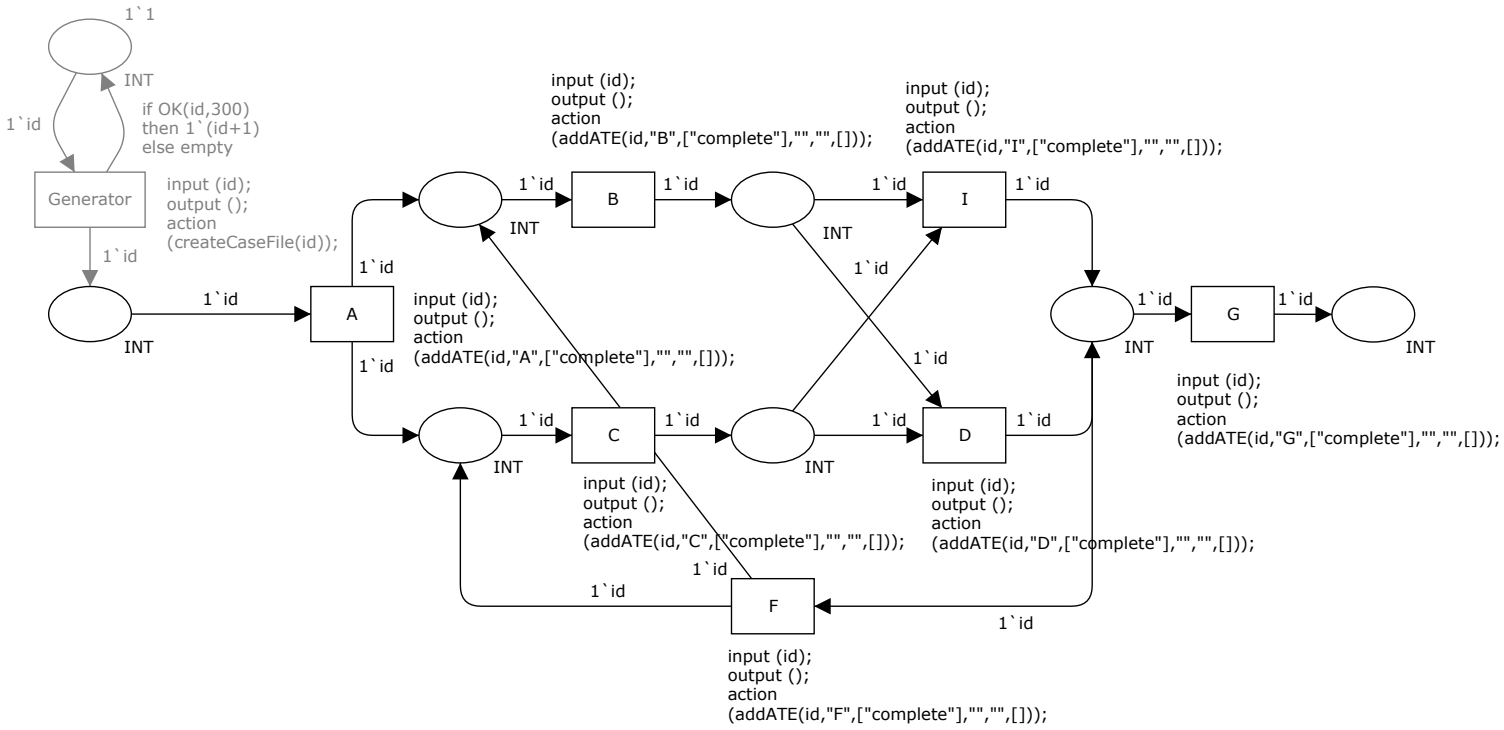


Figure B.13: CPN model for net al1.

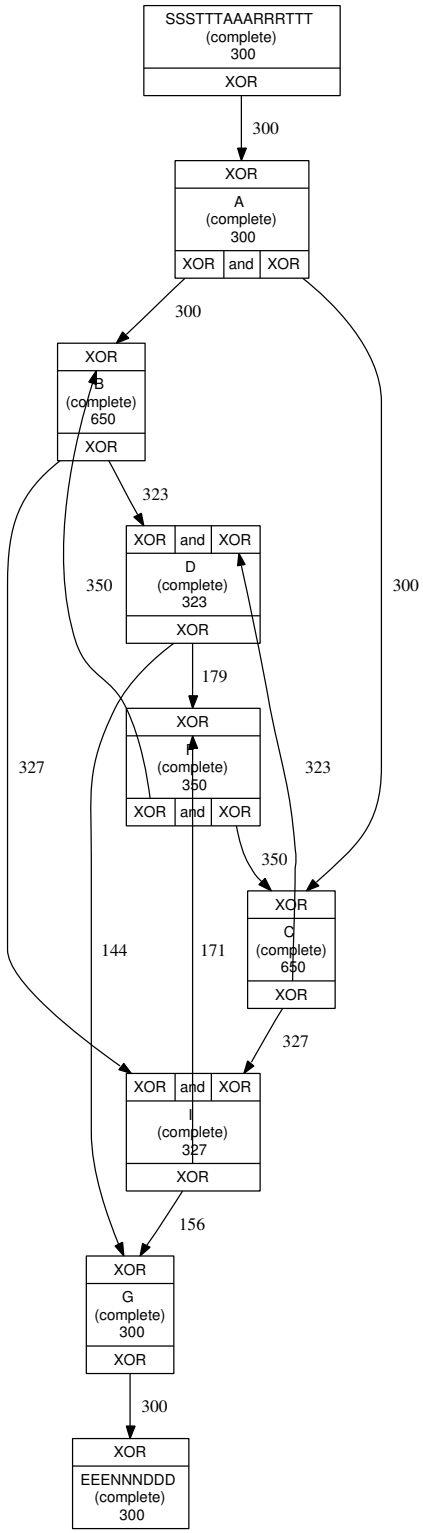


Figure B.14: Heuristic net for al1.

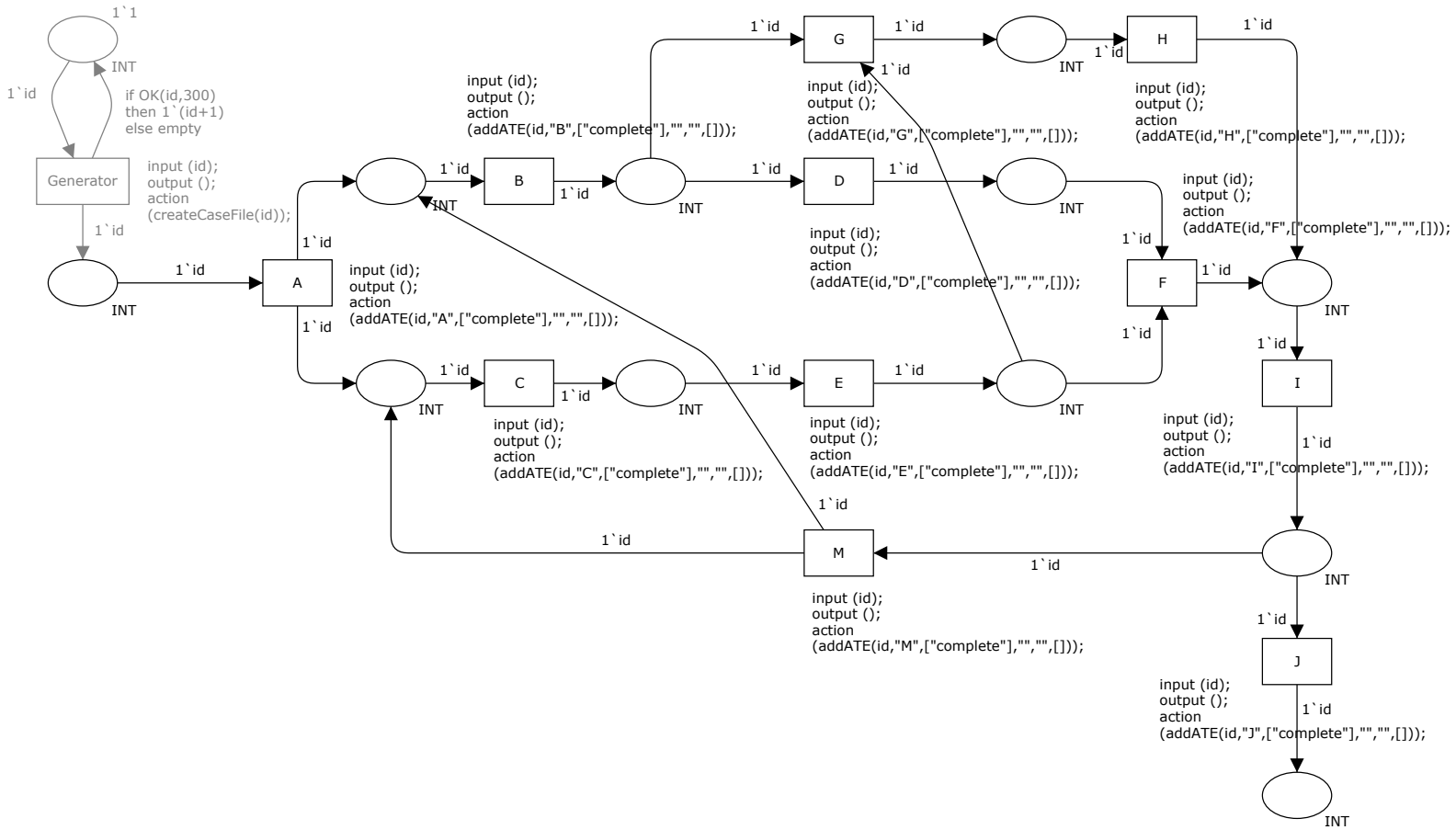


Figure B.15: CPN model for net al2.

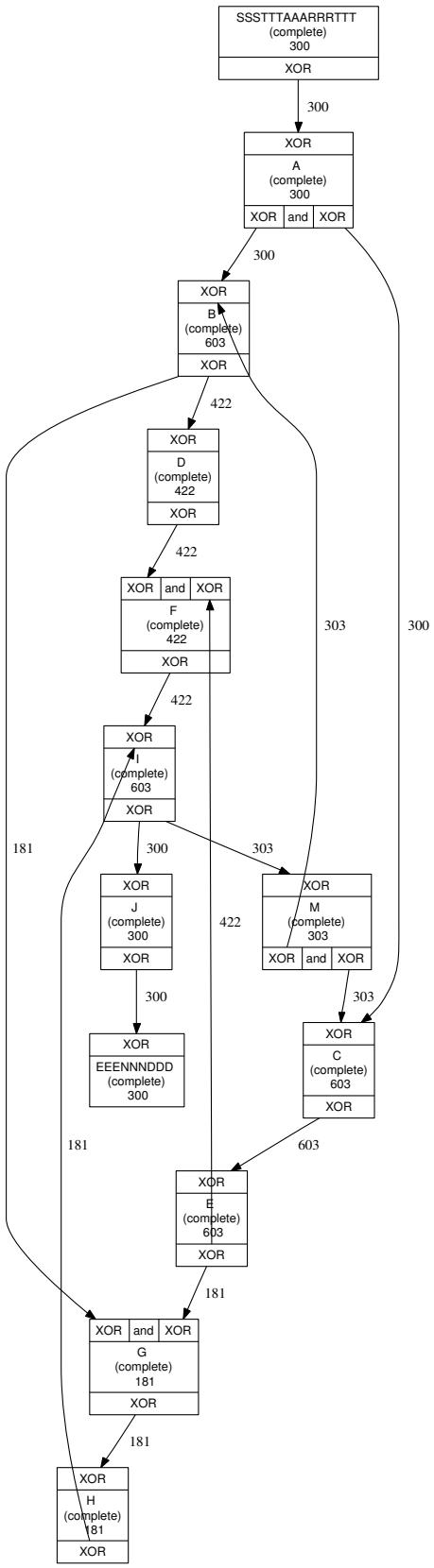


Figure B.16: Heuristic net for al2.

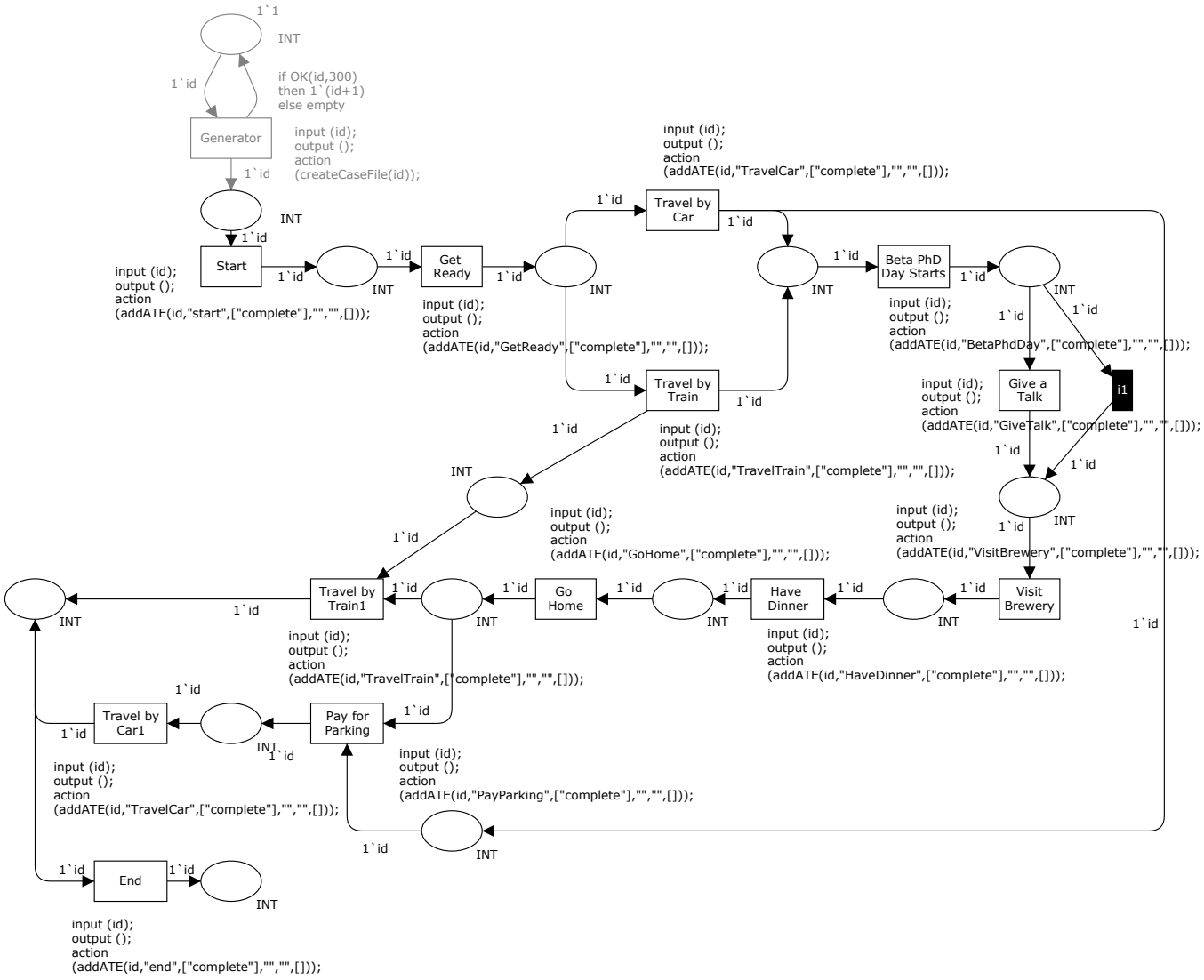


Figure B.17: CPN model for net betaSimplified.

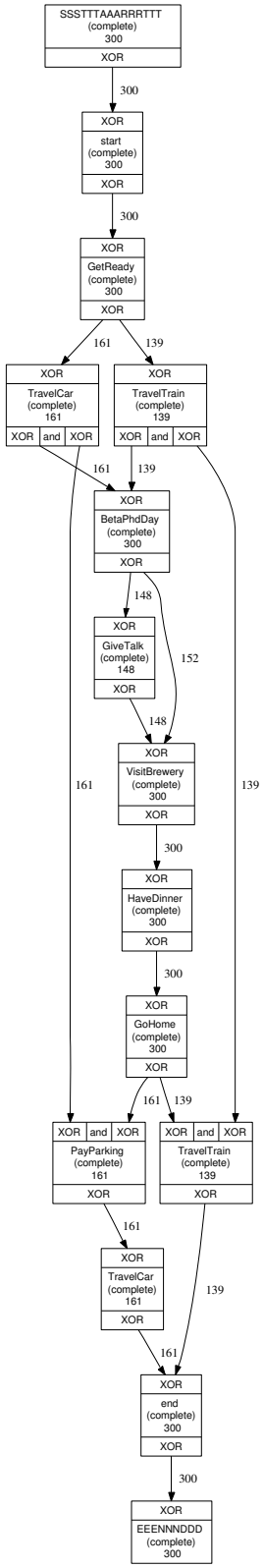


Figure B.18: Heuristic net for betaSimplified.

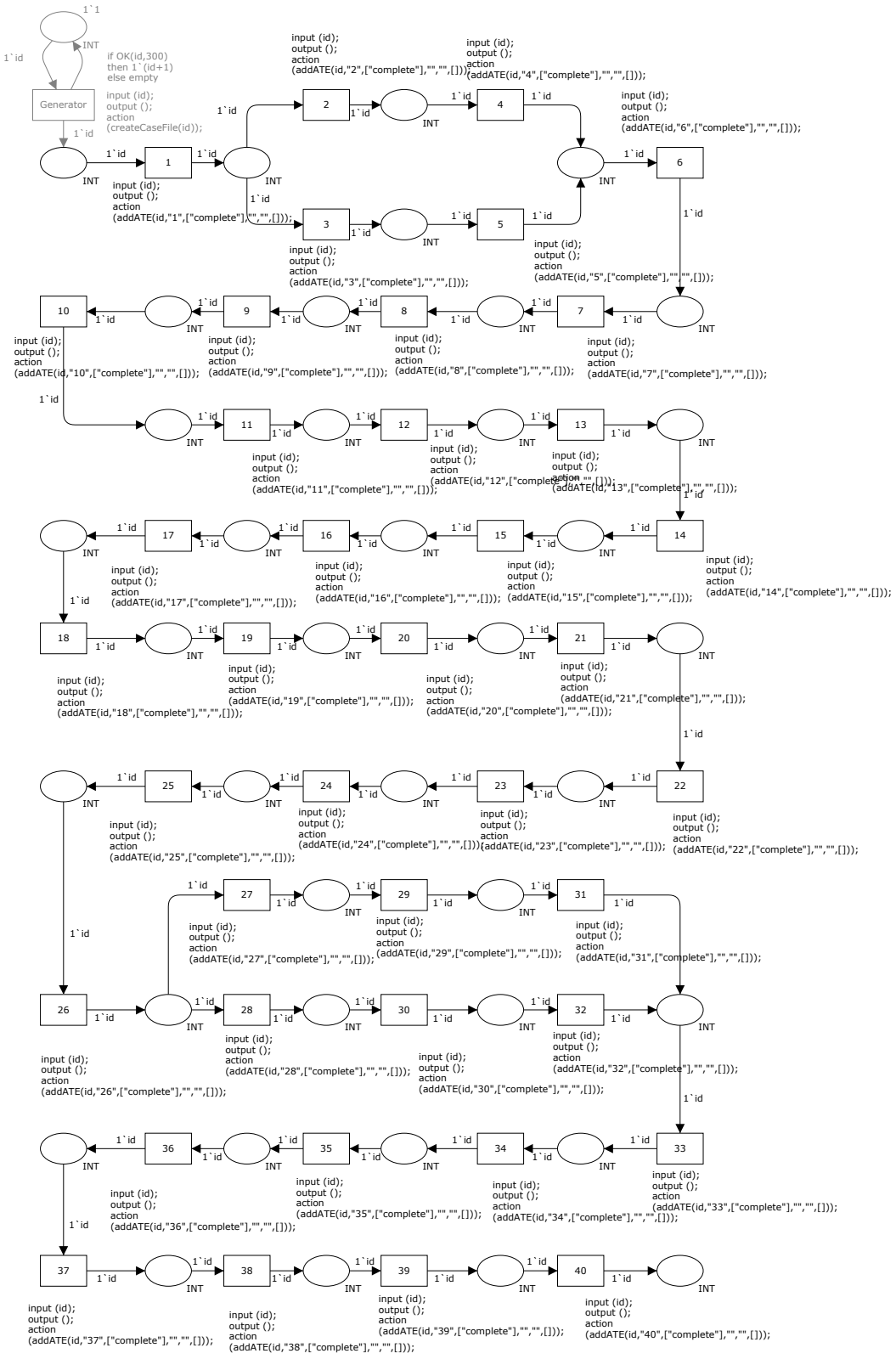


Figure B.19: CPN model for net bn1.

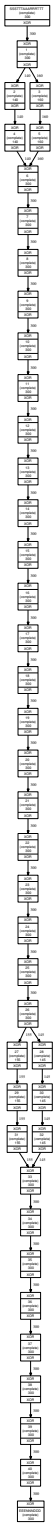


Figure B.20: Heuristic net for bn1.

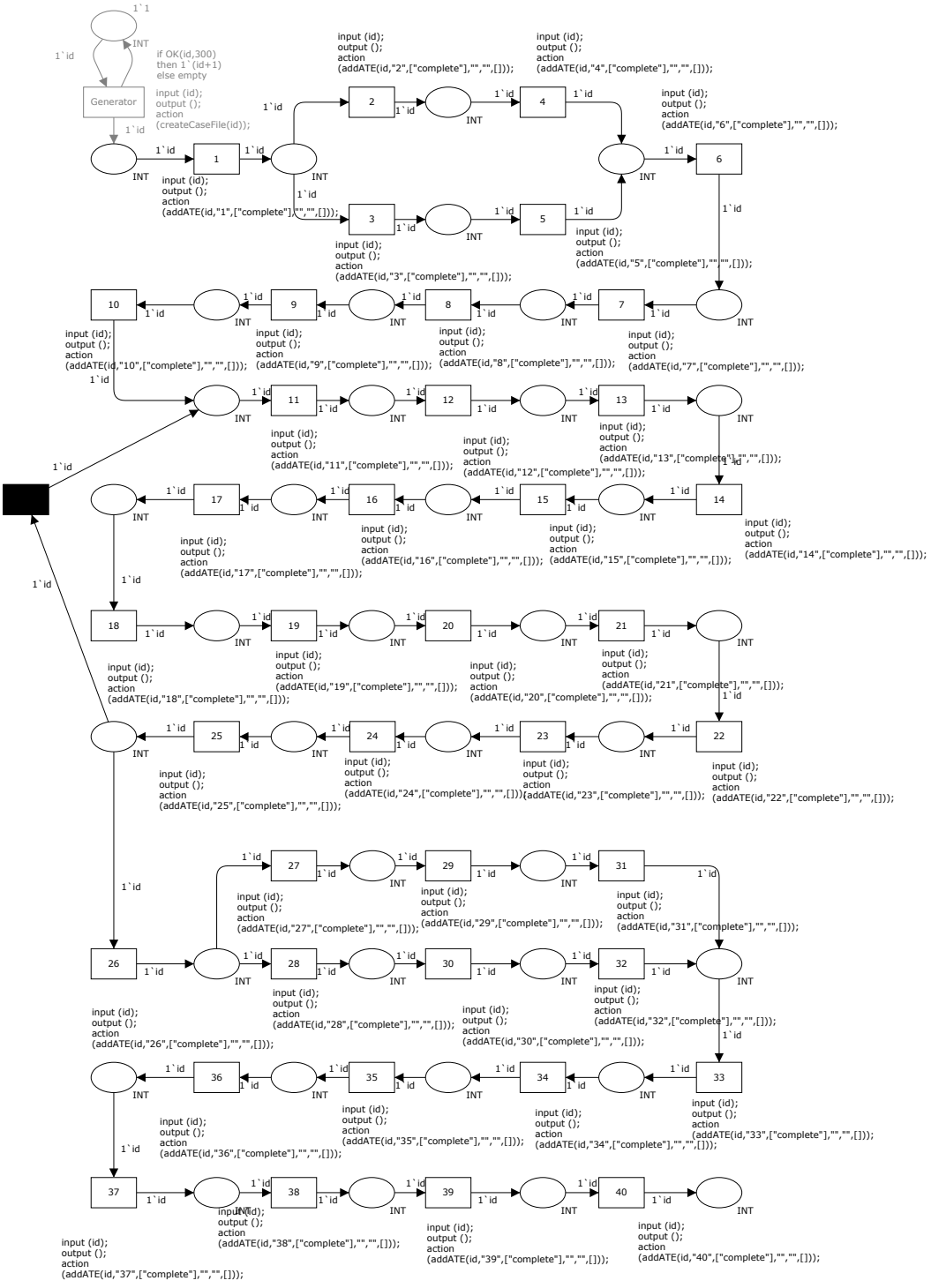


Figure B.21: CPN model for net bn2.

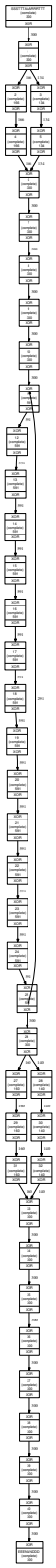


Figure B.22: Heuristic net for bn2.

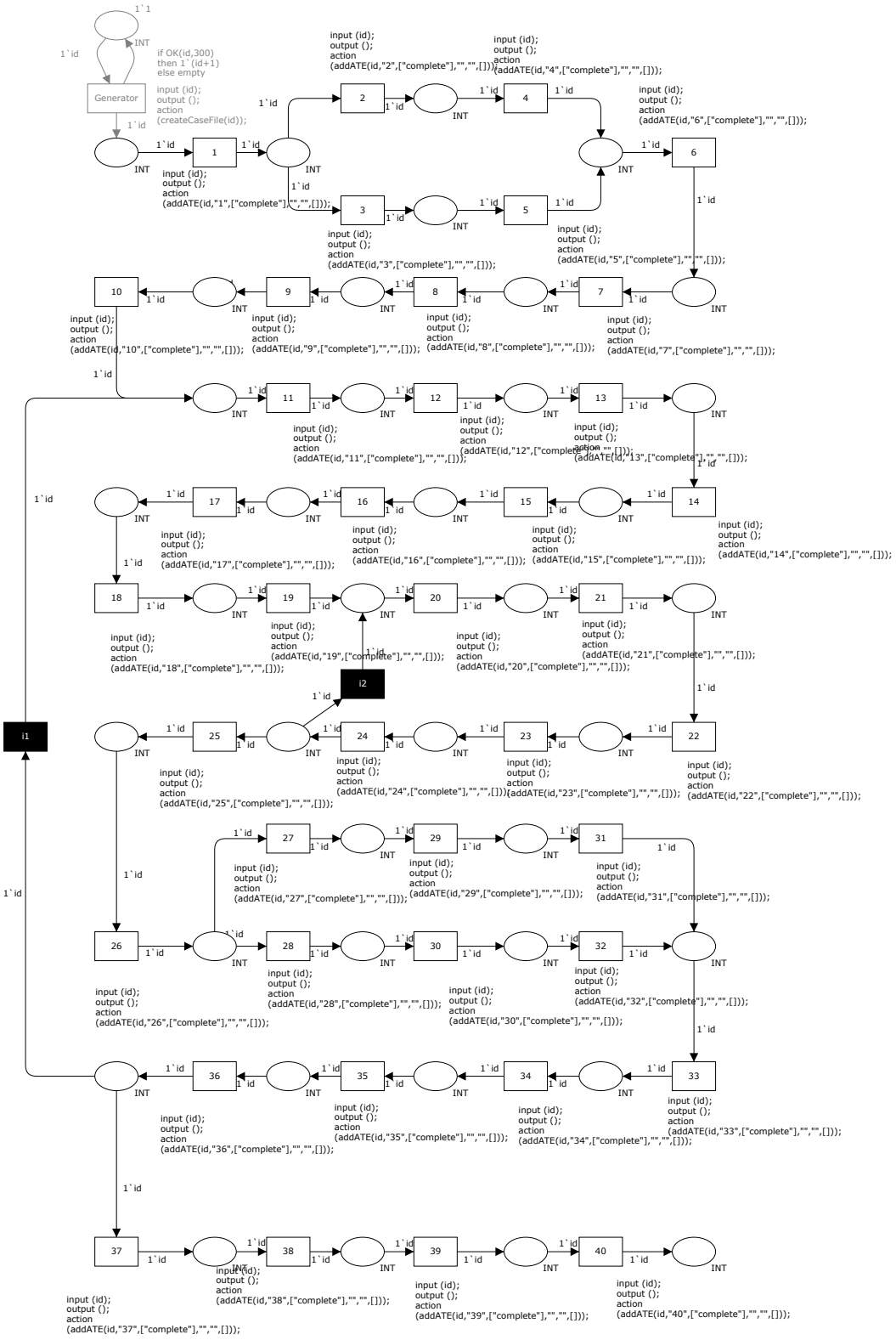


Figure B.23: CPN model for net bn3.

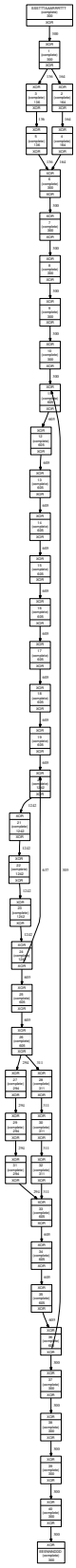


Figure B.24: Heuristic net for bn3.

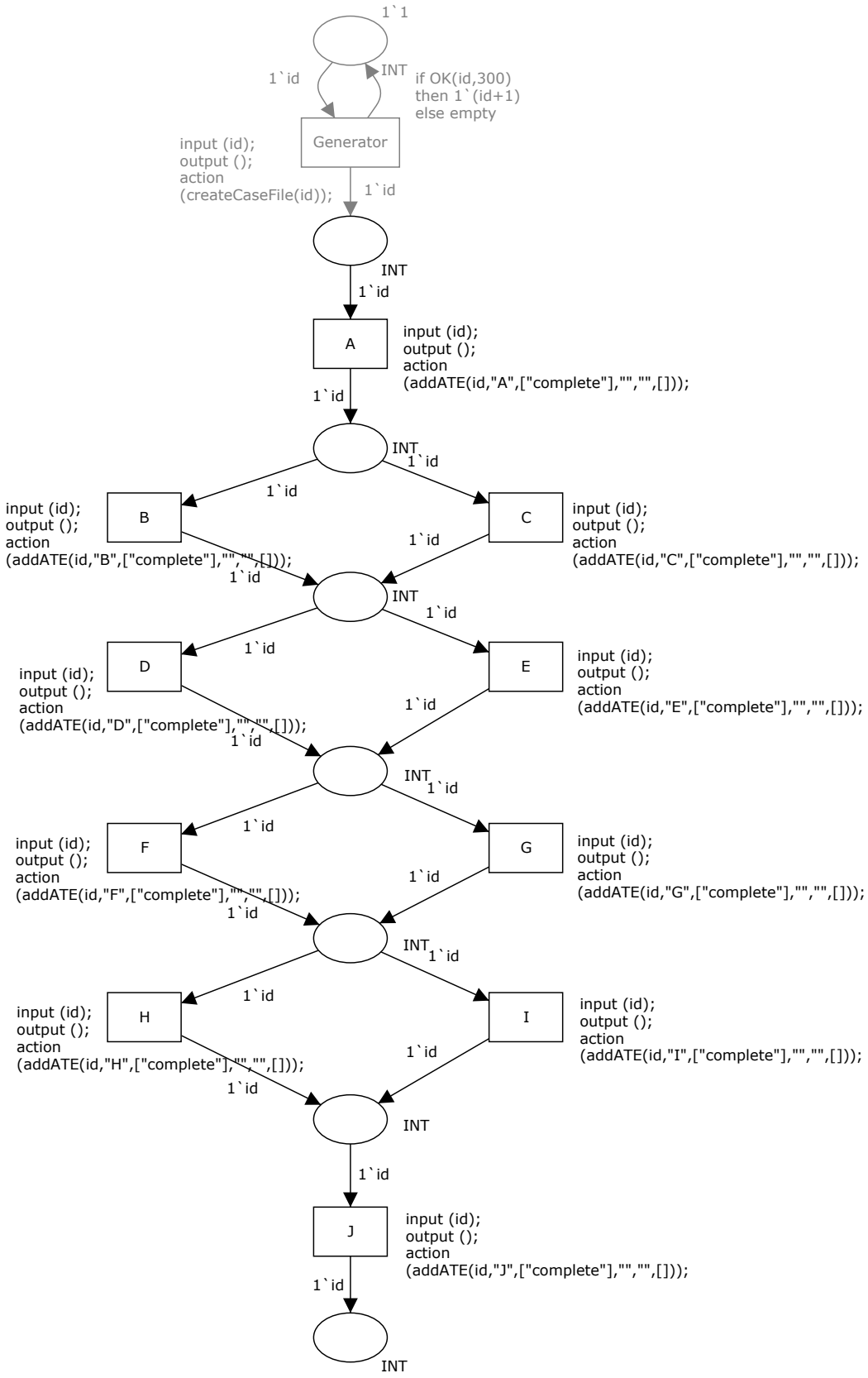


Figure B.25: CPN model for net choice.

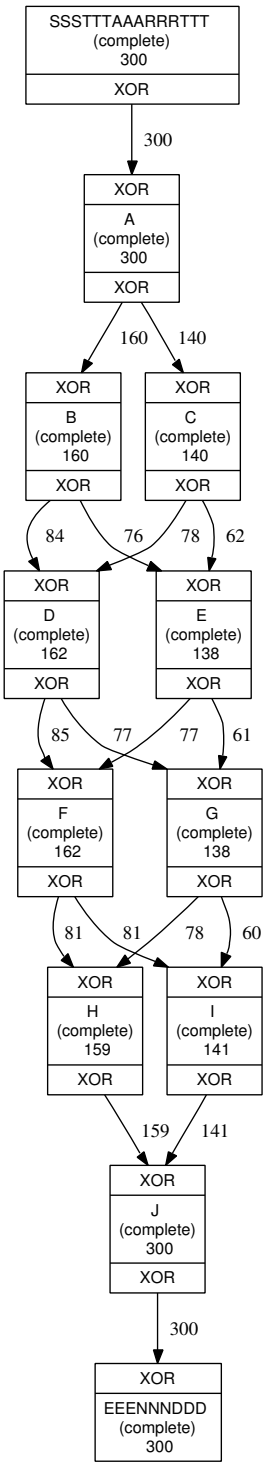


Figure B.26: Heuristic net for choice.

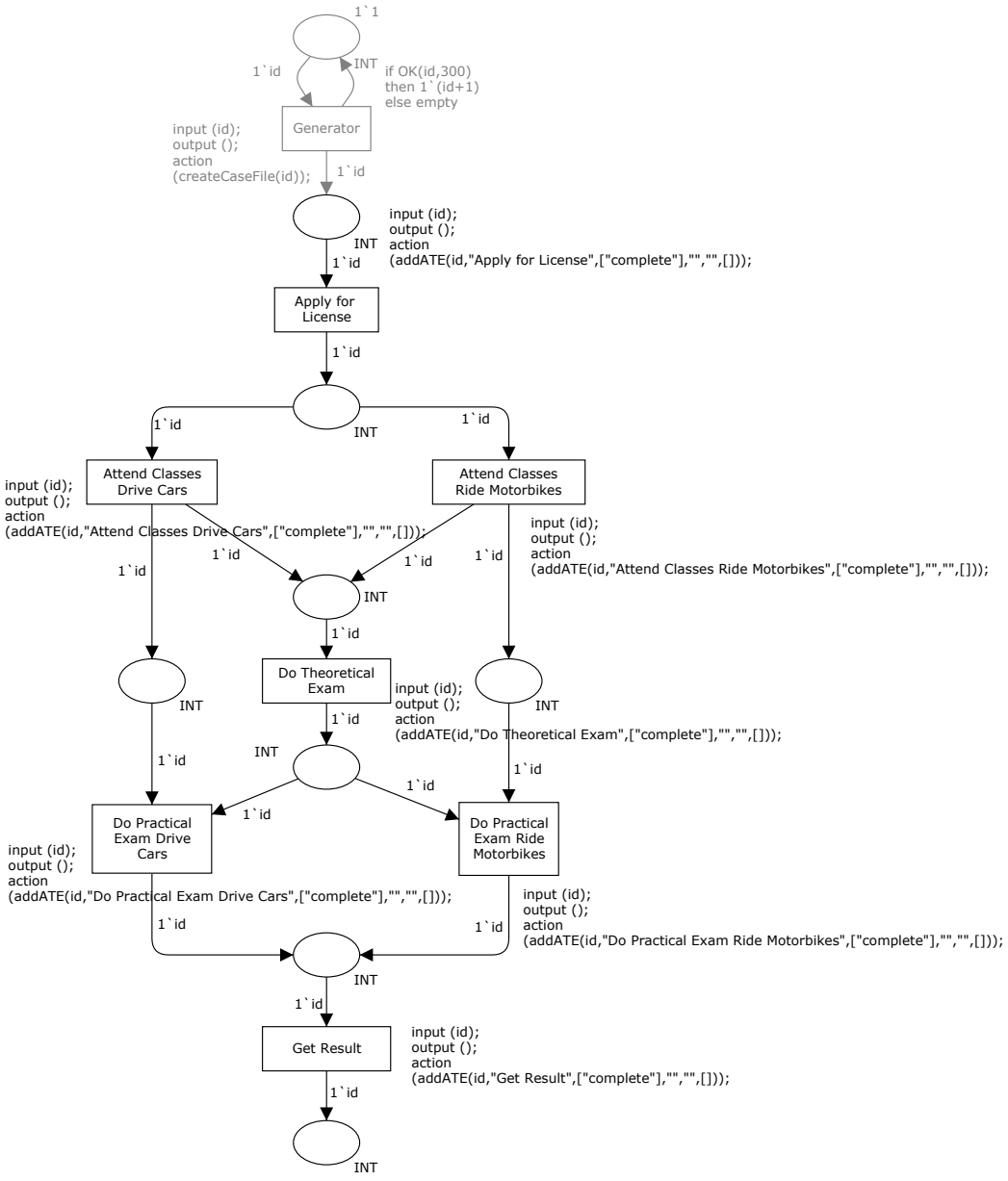


Figure B.27: CPN model for net driversLicense.

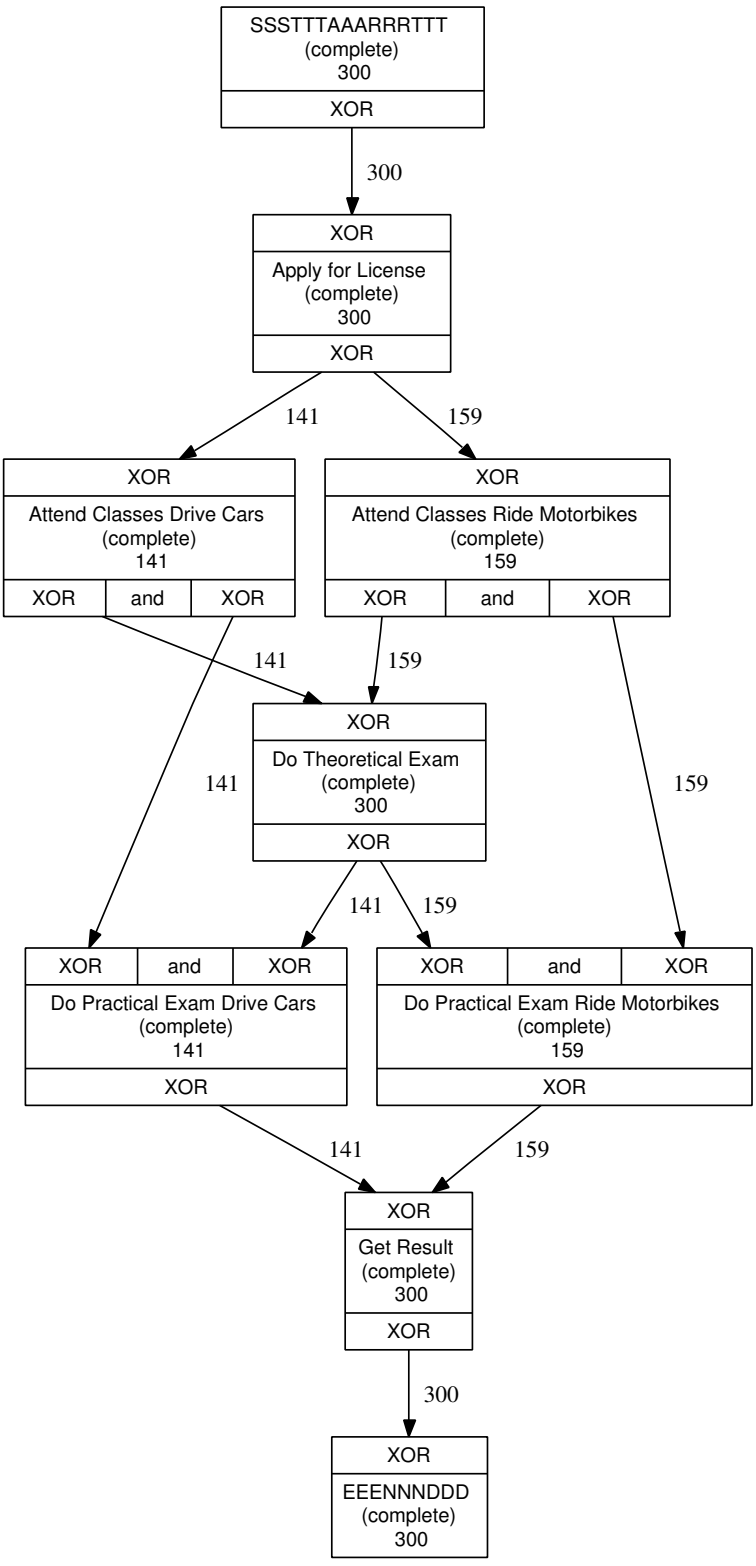


Figure B.28: Heuristic net for driversLicense.

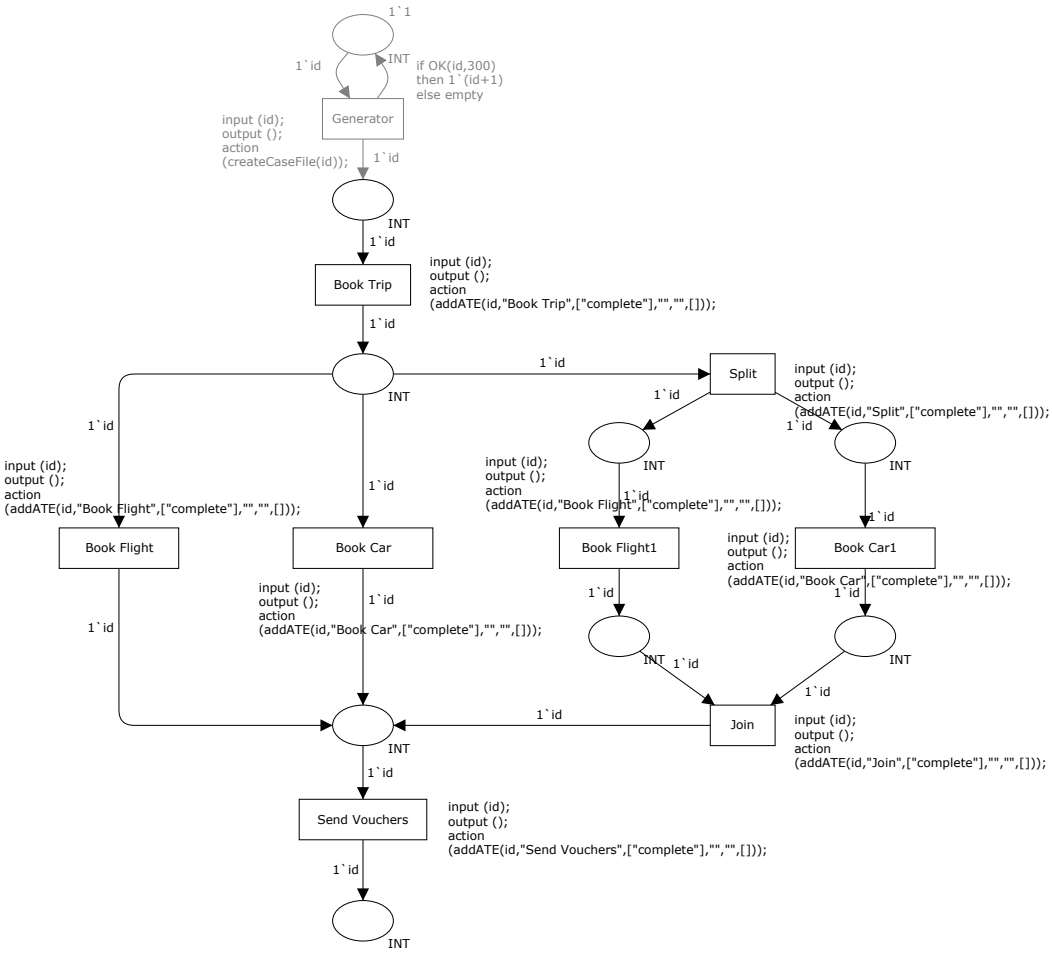


Figure B.29: CPN model for net flightCar.

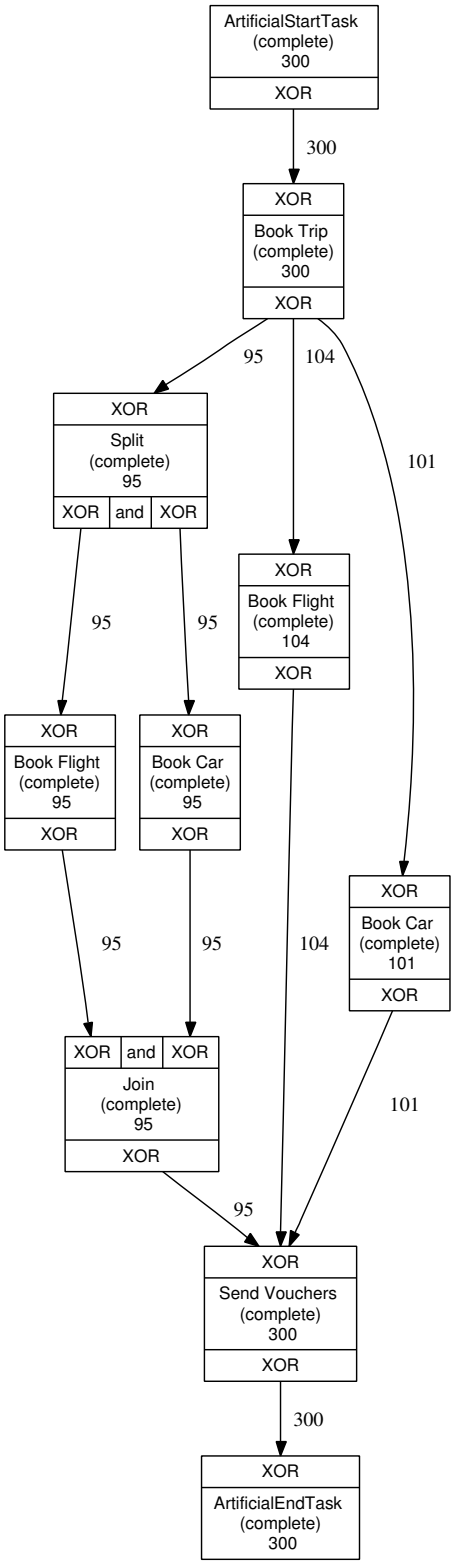


Figure B.30: Heuristic net for flightCar.

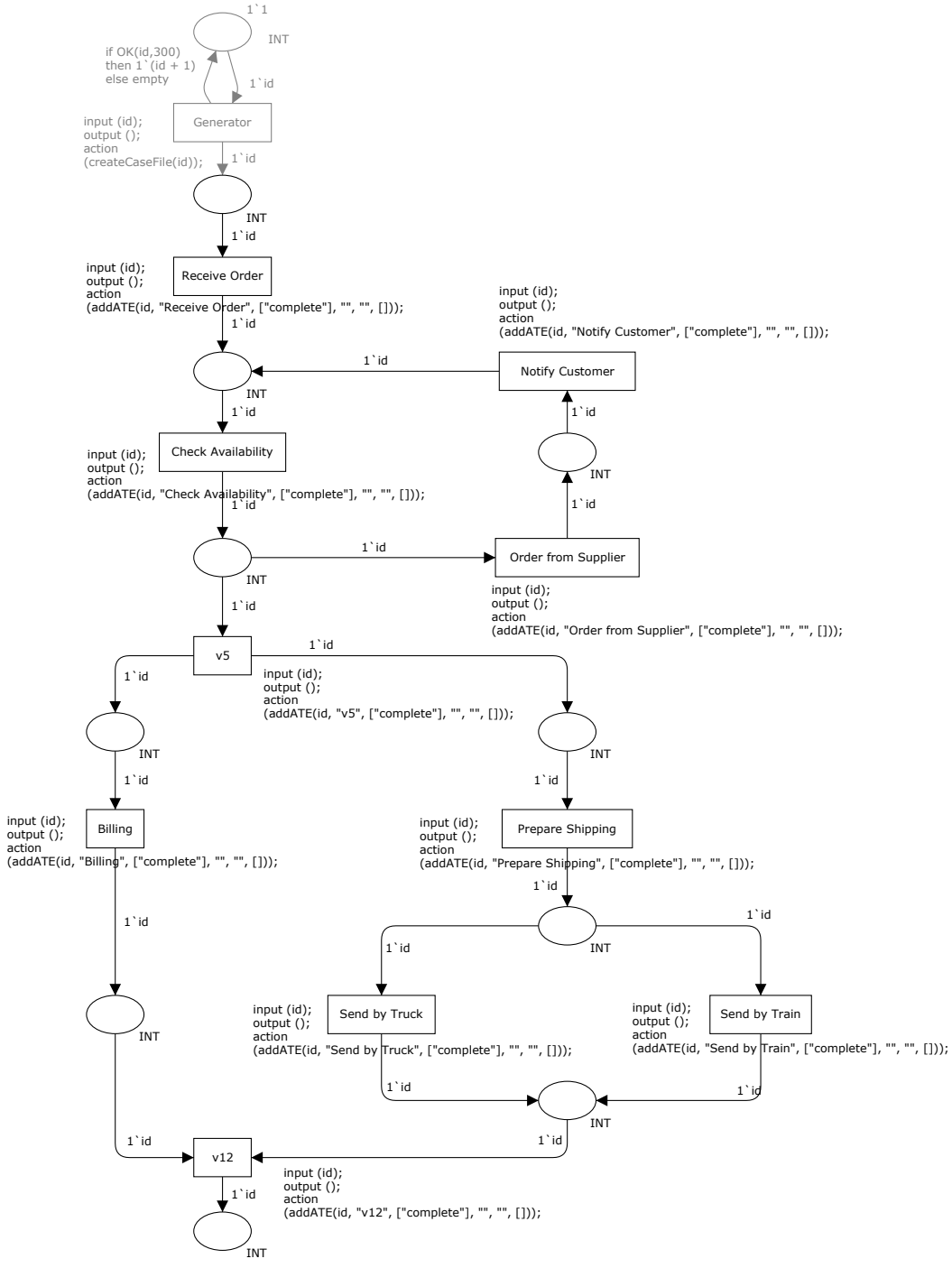


Figure B.31: CPN model for net herbstFig3p4.

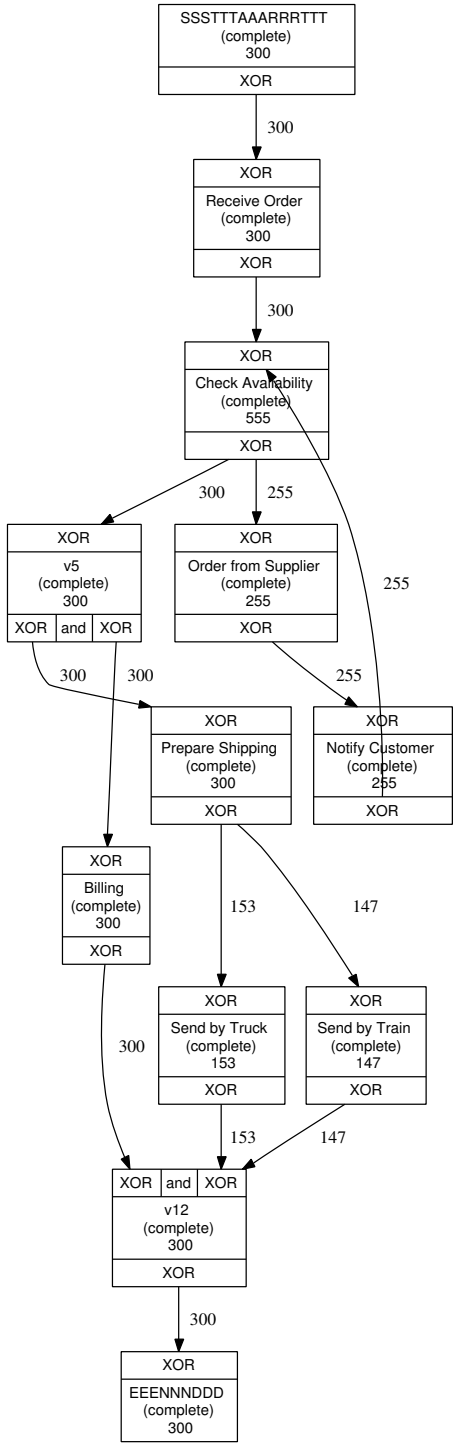


Figure B.32: Heuristic net for herbstFig3p4.

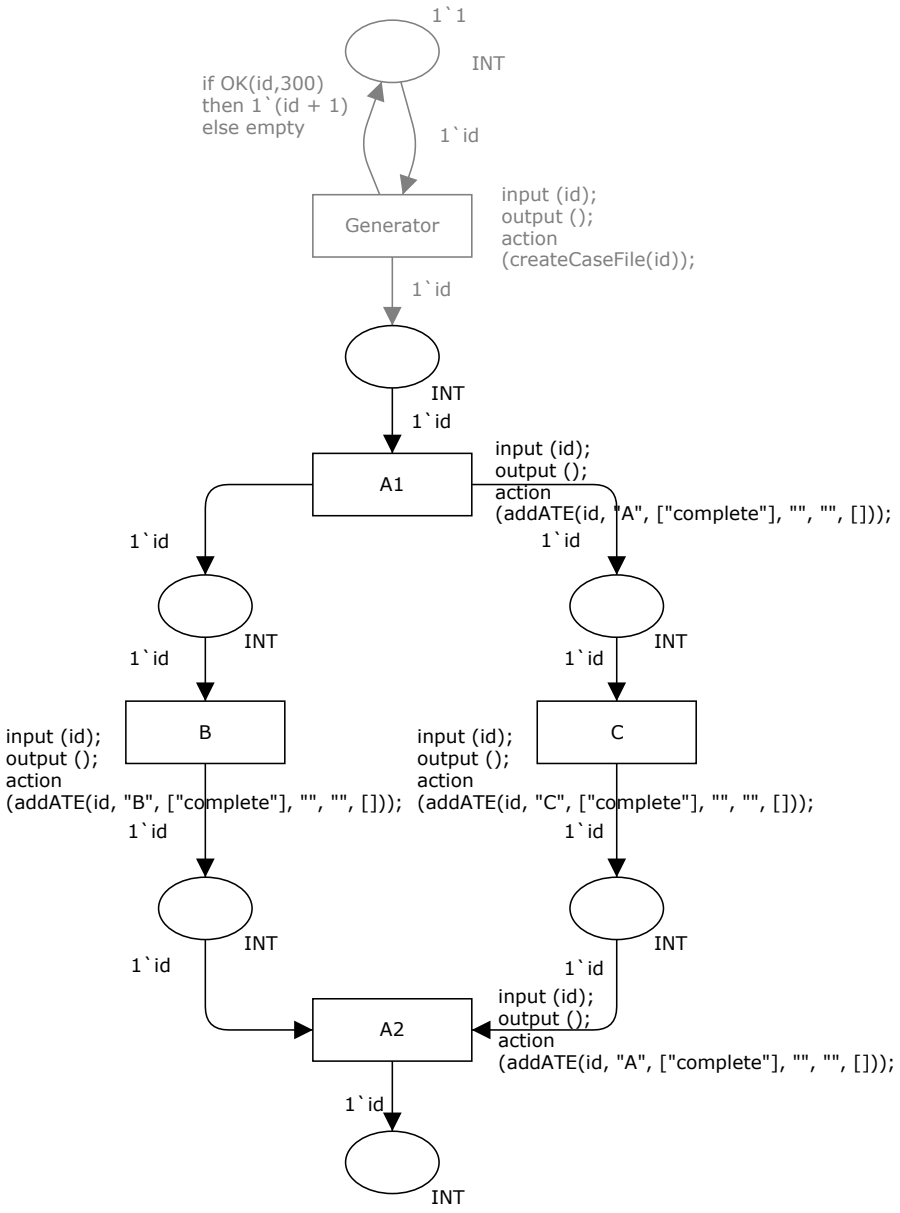


Figure B.33: CPN model for net herbstFig5p1AND.

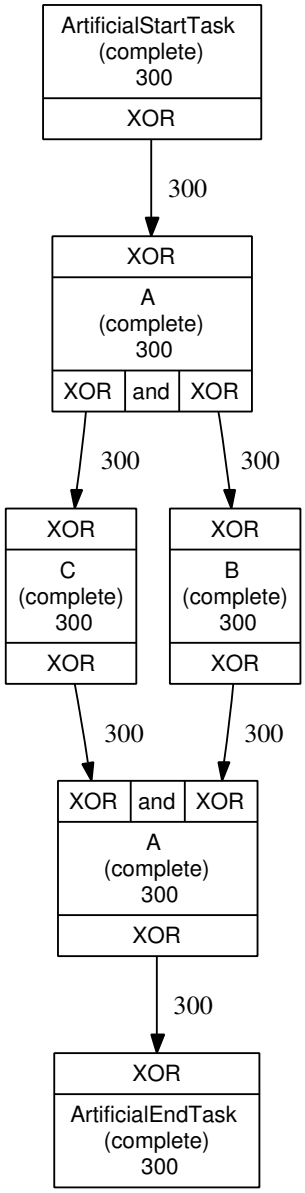


Figure B.34: Heuristic net for herbstFig5p1AND.

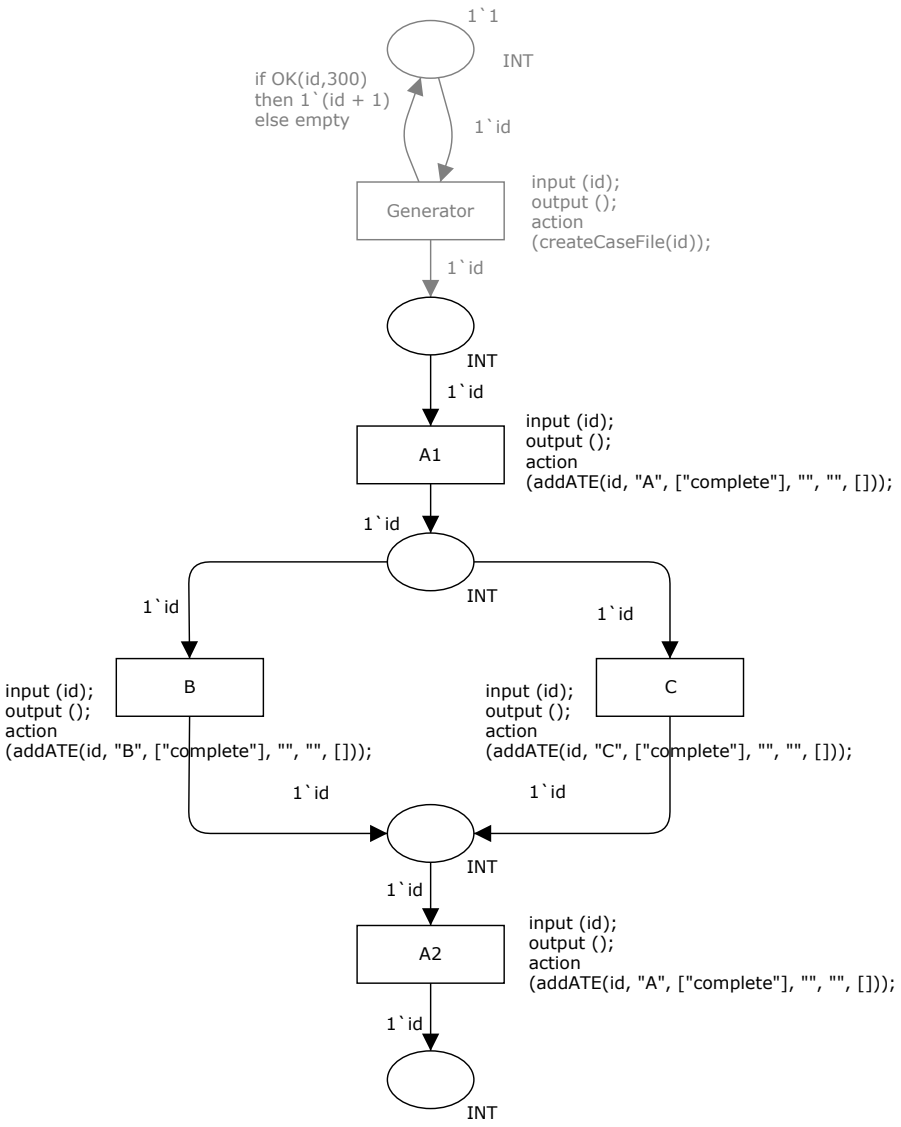


Figure B.35: CPN model for net herbstFig5p1OR.

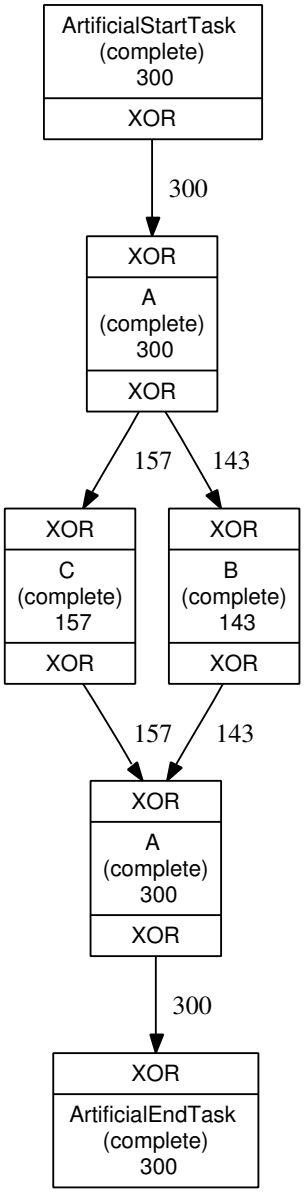


Figure B.36: Heuristic net for herbstFig5p1OR.

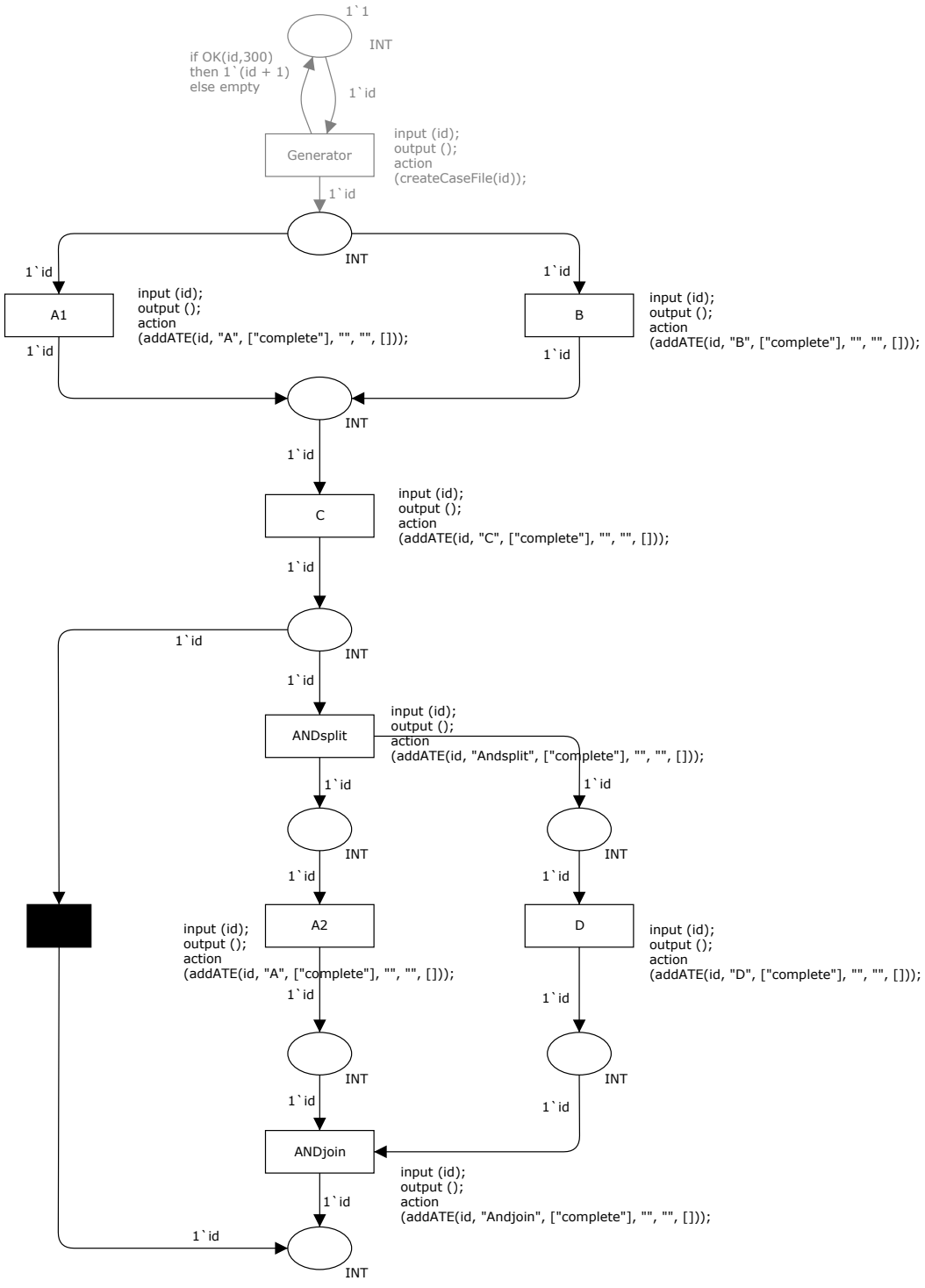


Figure B.37: CPN model for net herbstFig5p19.

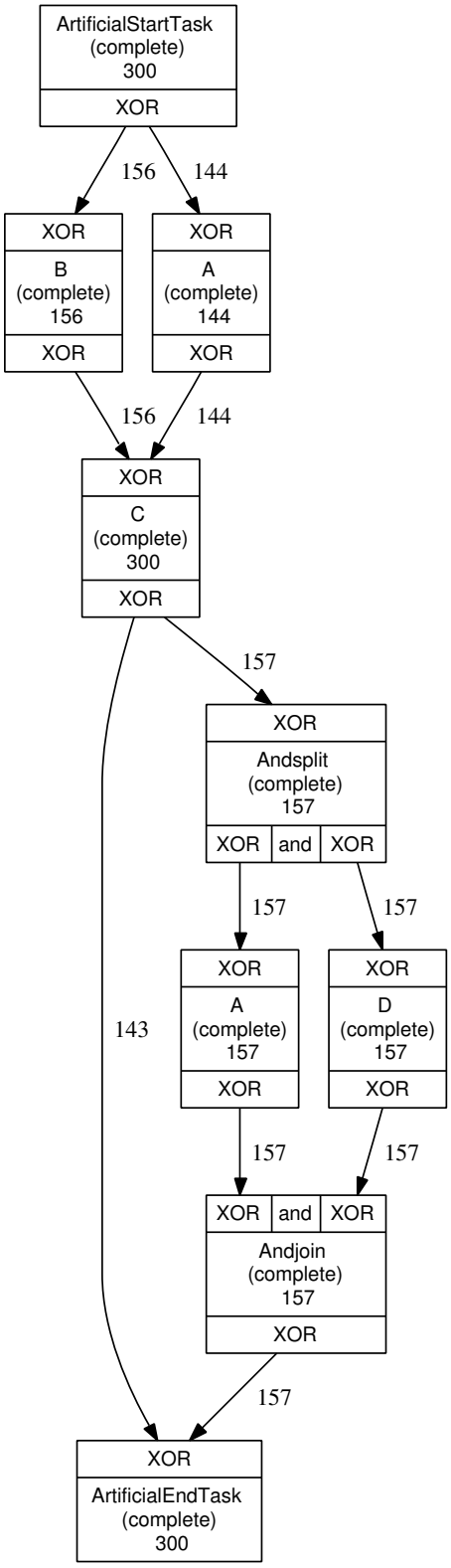


Figure B.38: Heuristic net for herbstFig5p19.

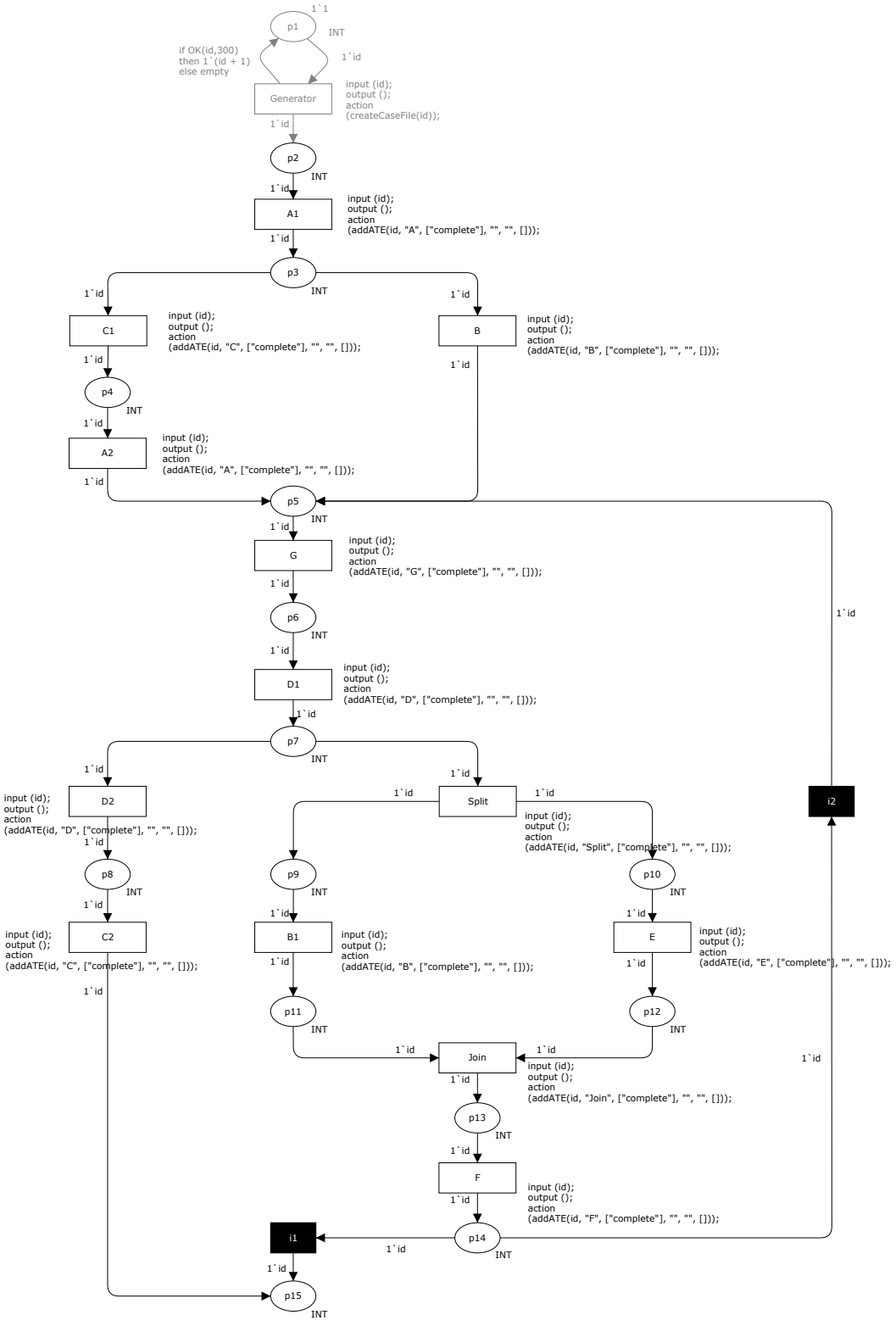


Figure B.39: CPN model for net herbstFig6p10.

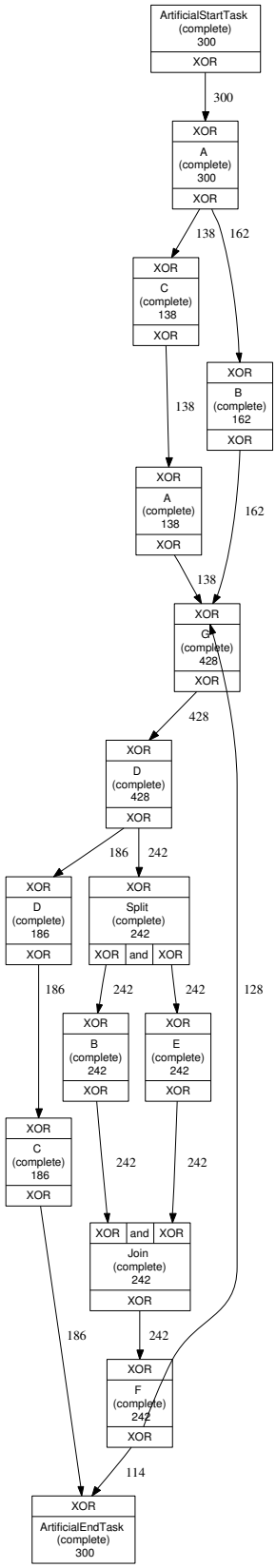


Figure B.40: Heuristic net for herbstFig6p10.

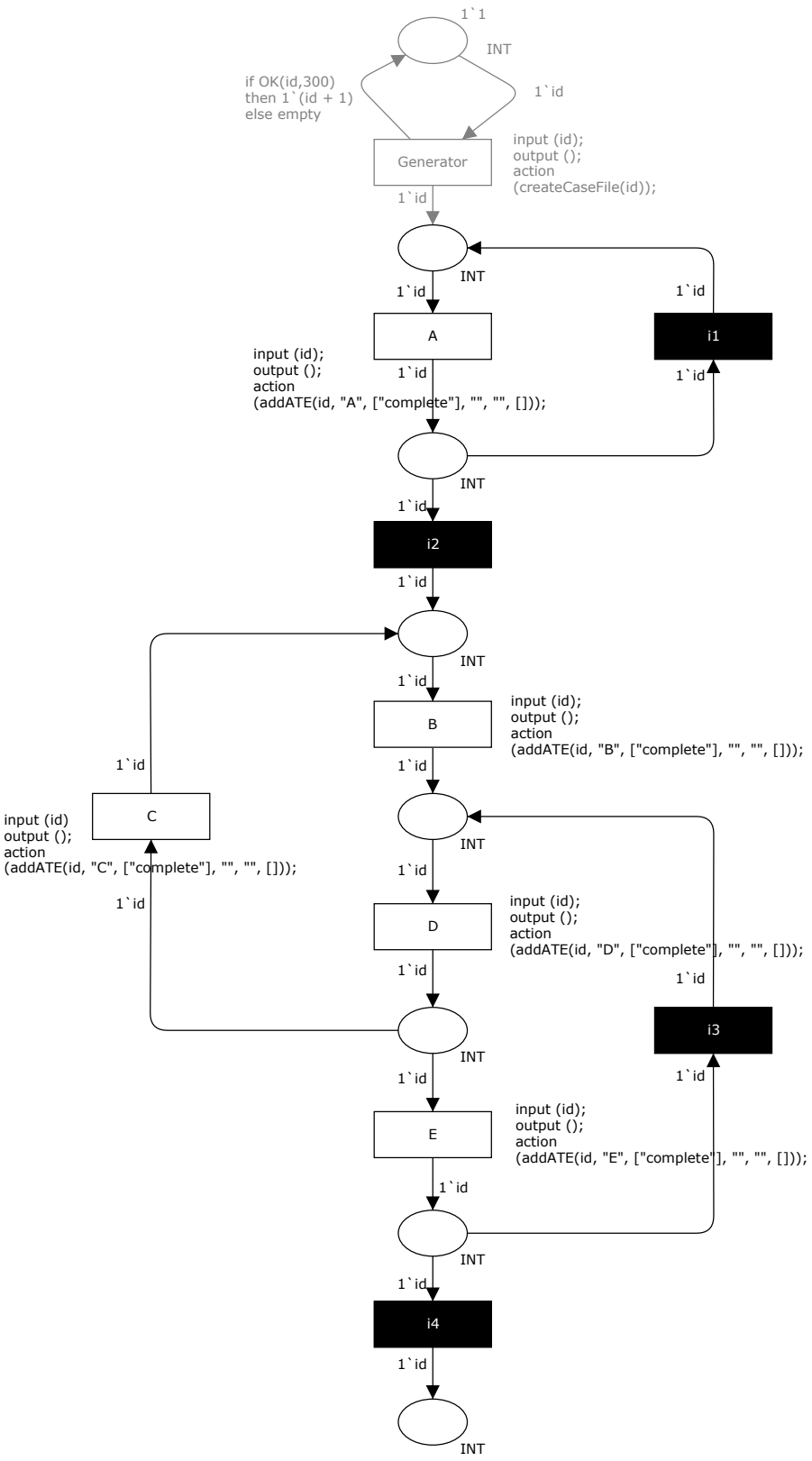


Figure B.41: CPN model for net herbstFig6p18.

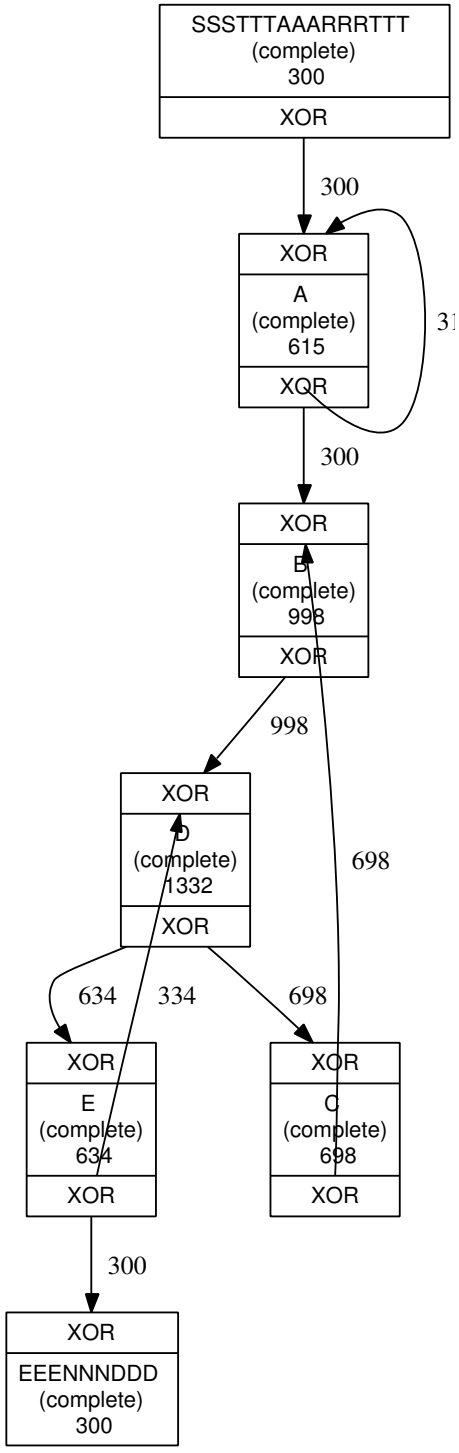


Figure B.42: Heuristic net for herbstFig6p18.

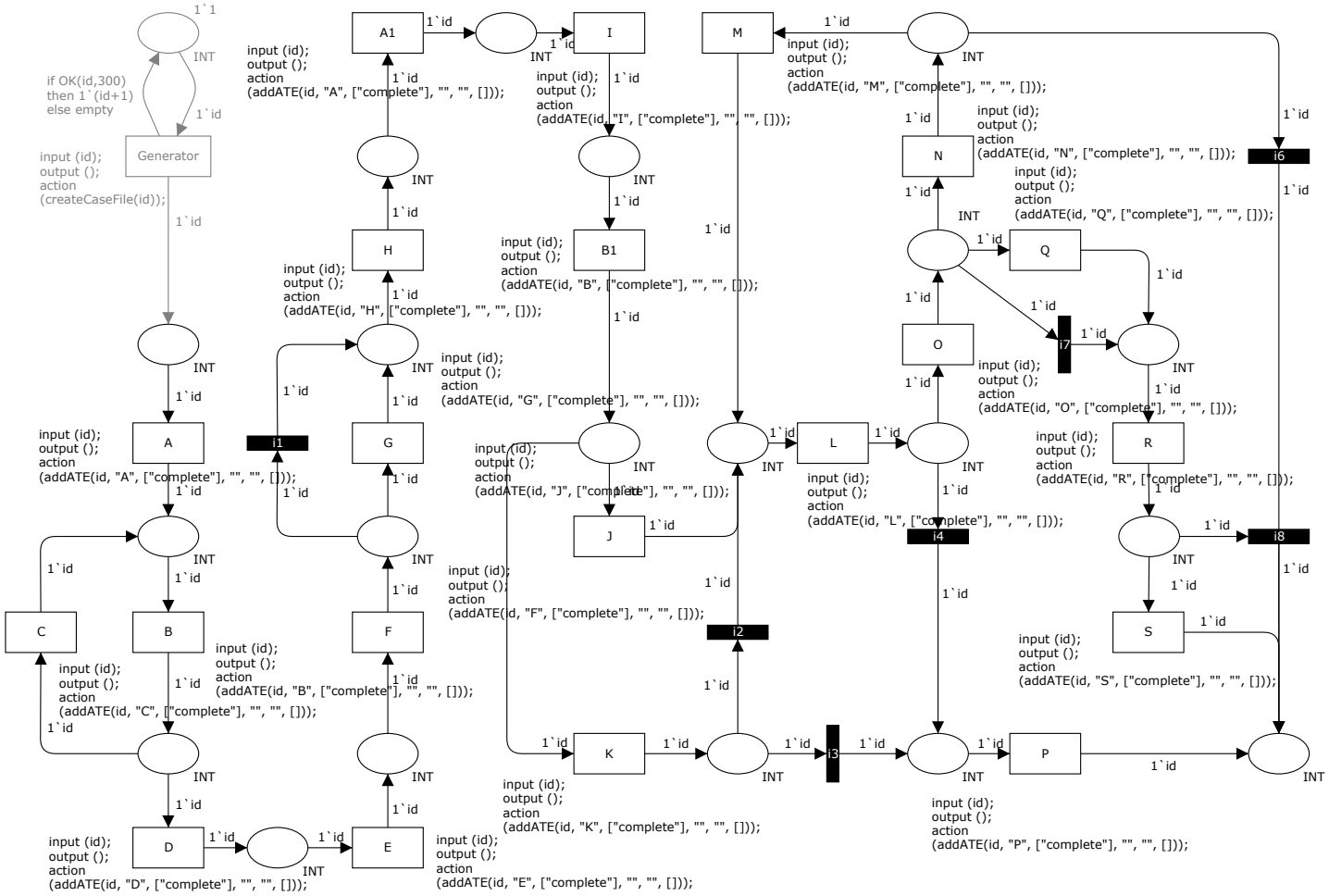


Figure B.43: CPN model for net herbstFig6p25.

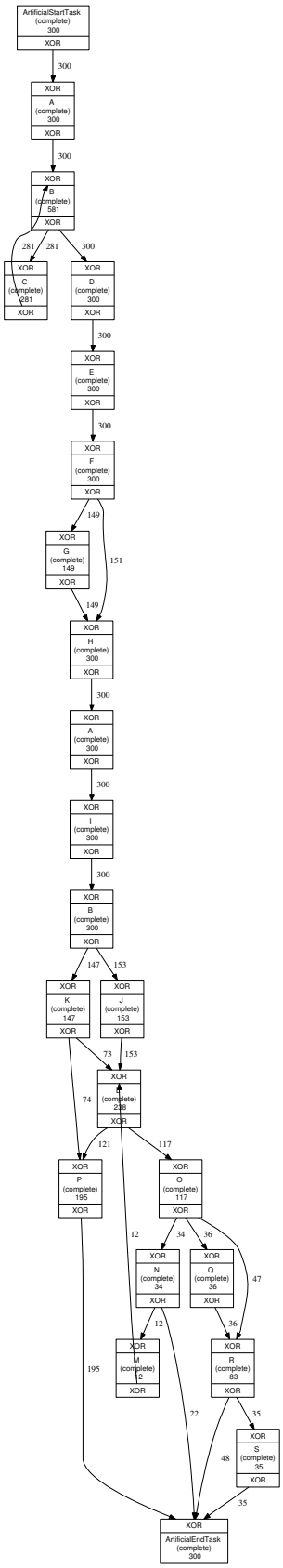


Figure B.44: Heuristic net for herbstFig6p25.

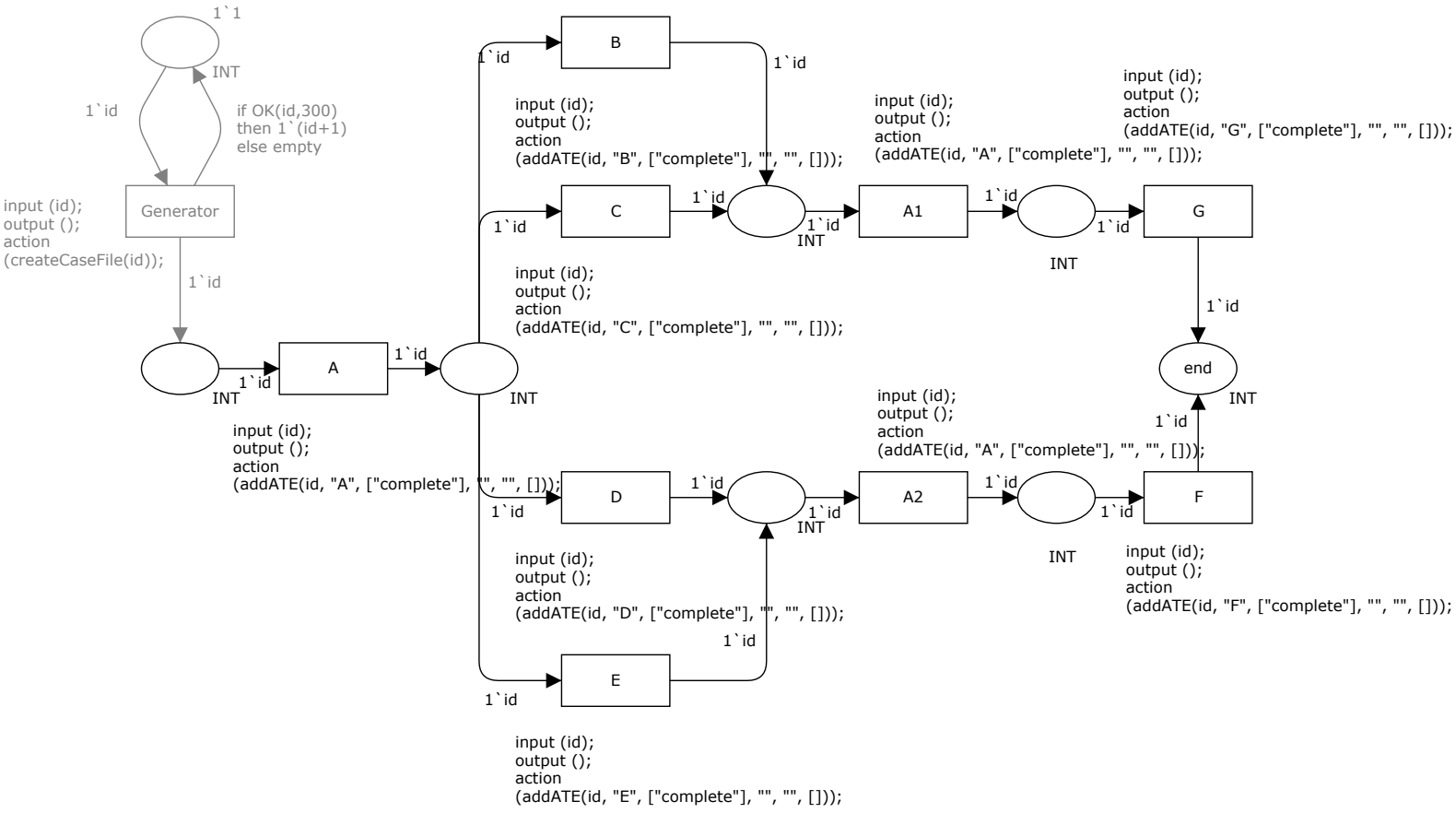


Figure B.45: CPN model for net herbstFig6p31.

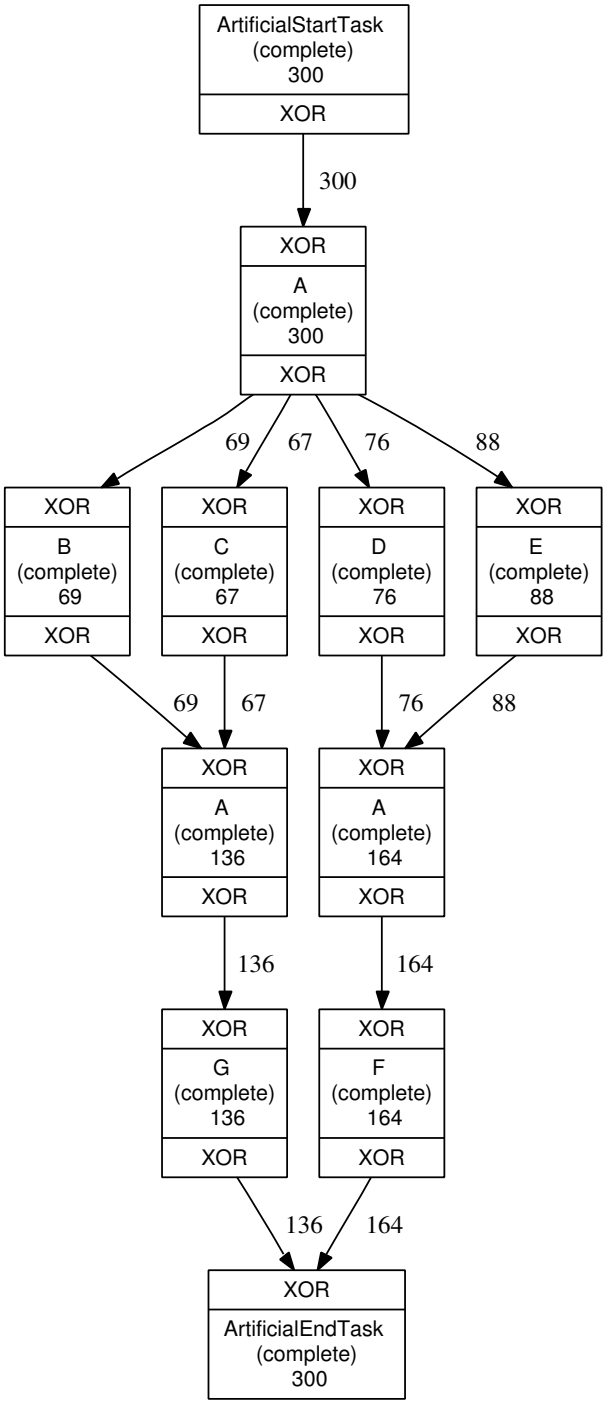


Figure B.46: Heuristic net for herbstFig6p31.

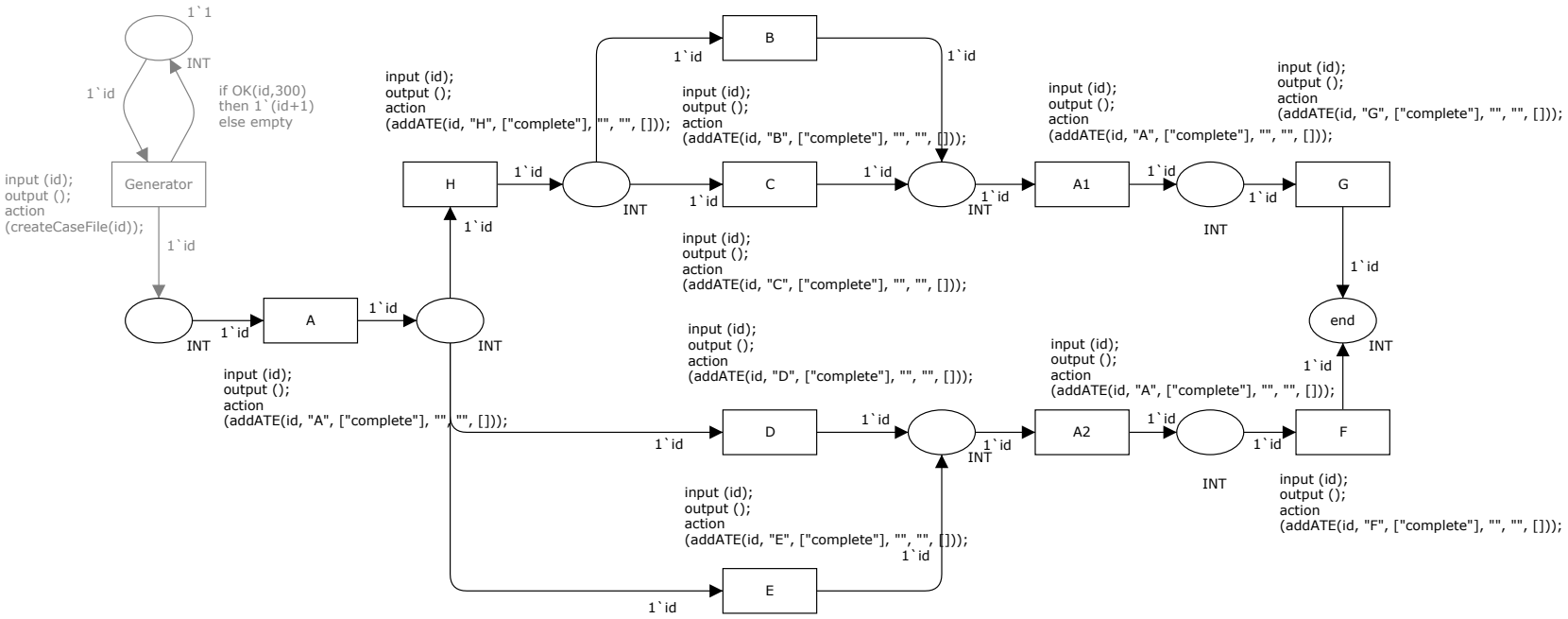


Figure B.47: CPN model for net herbstFig6p33.

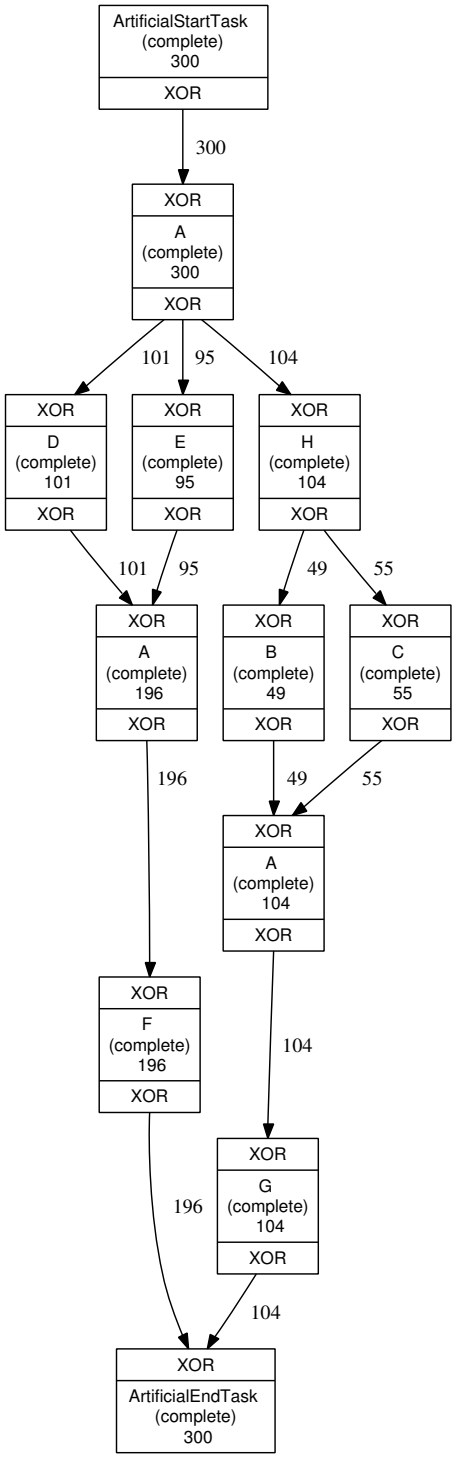


Figure B.48: Heuristic net for herbstFig6p33.

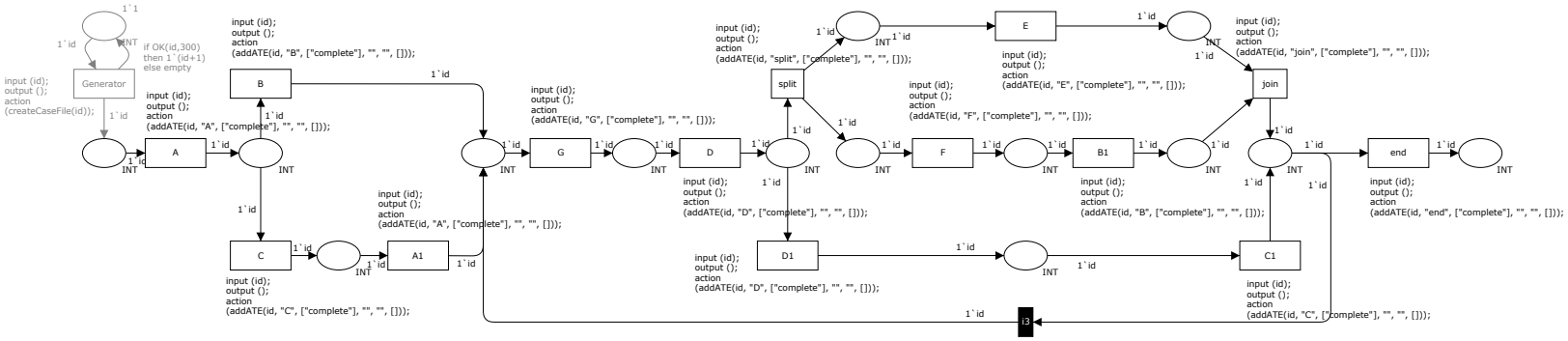


Figure B.49: CPN model for net herbstFig6p34.

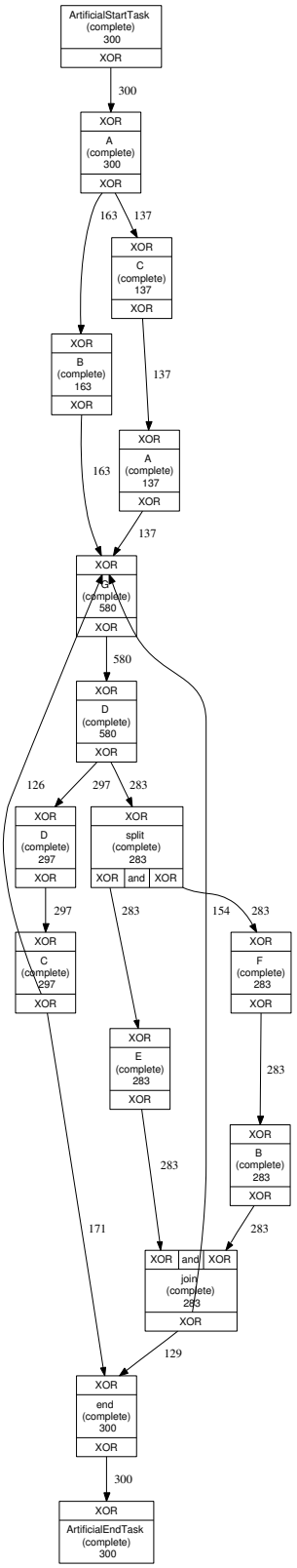


Figure B.50: Heuristic net for herbstFig6p34.

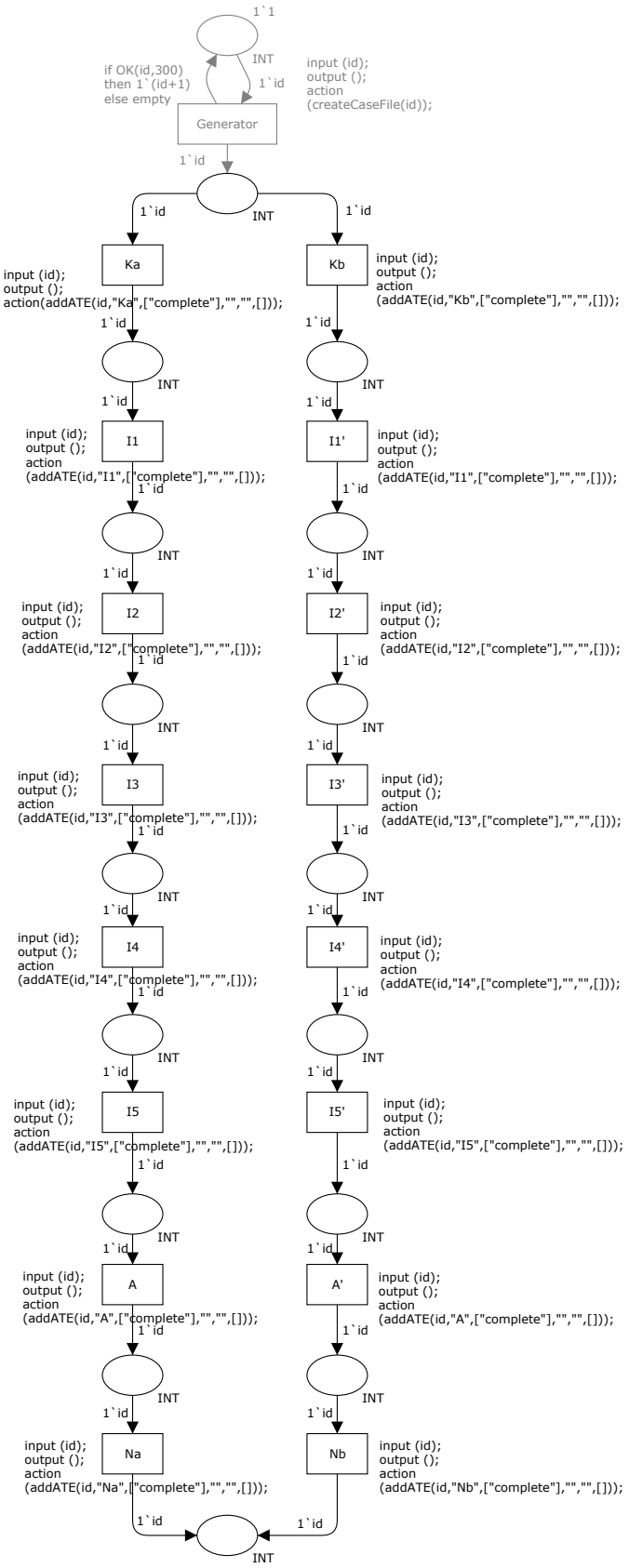


Figure B.51: CPN model for net herbstFig6p36.

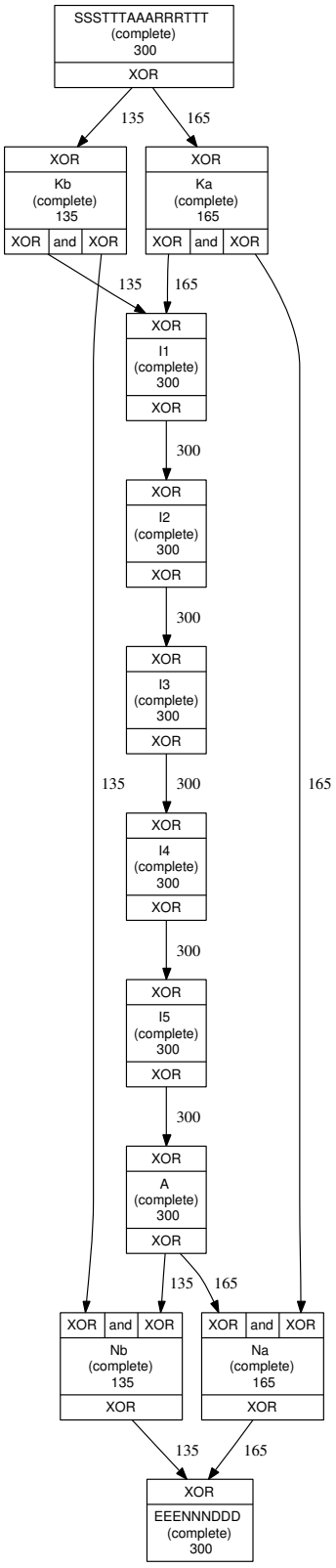


Figure B.52: Heuristic net for herbstFig6p36.

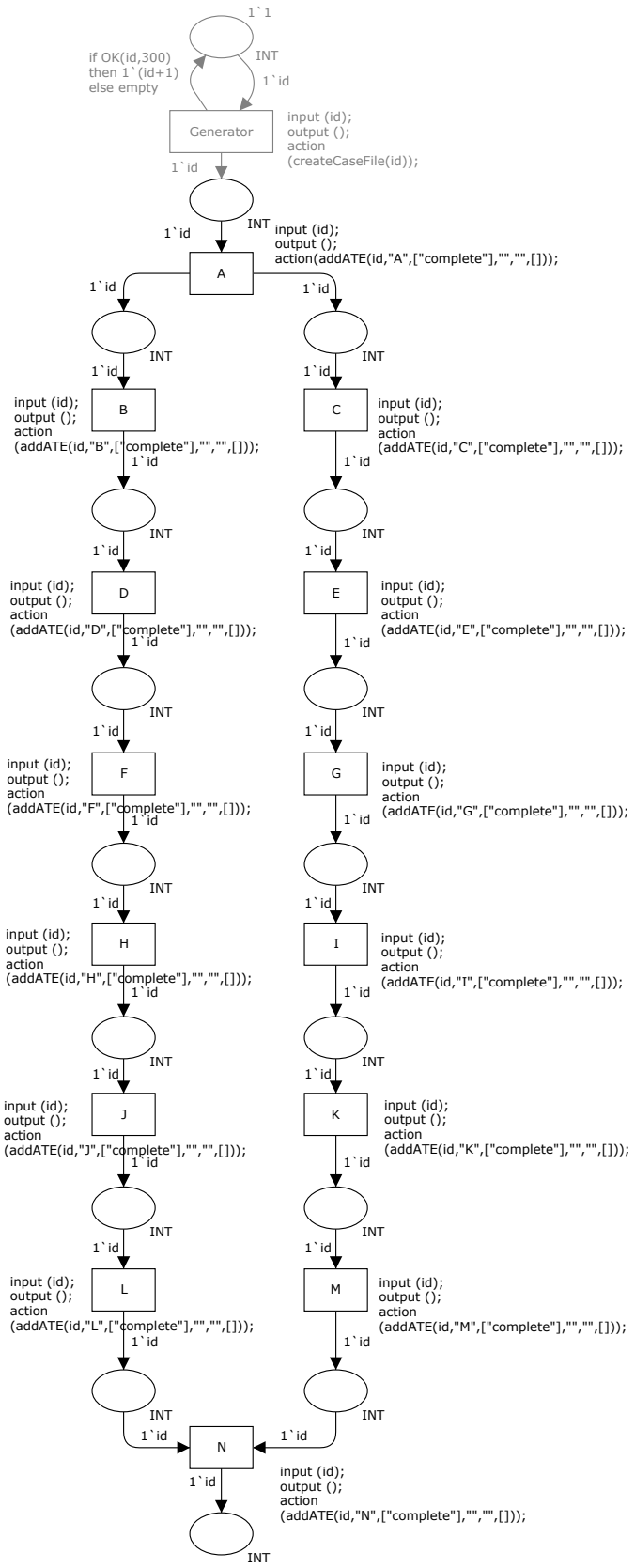


Figure B.53: CPN model for net herbstFig6p37

.

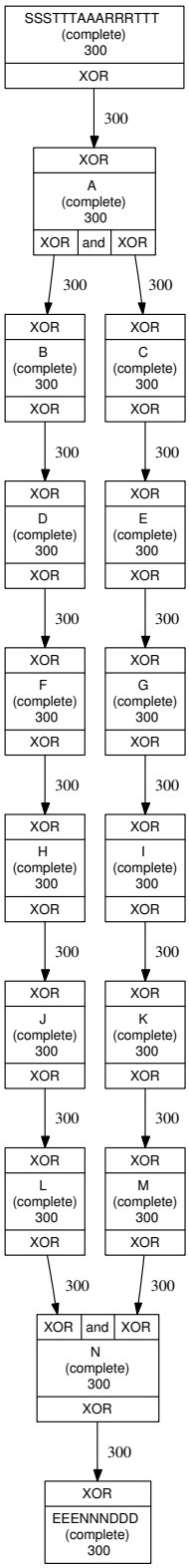


Figure B.54: Heuristic net for herbstFig6p37.

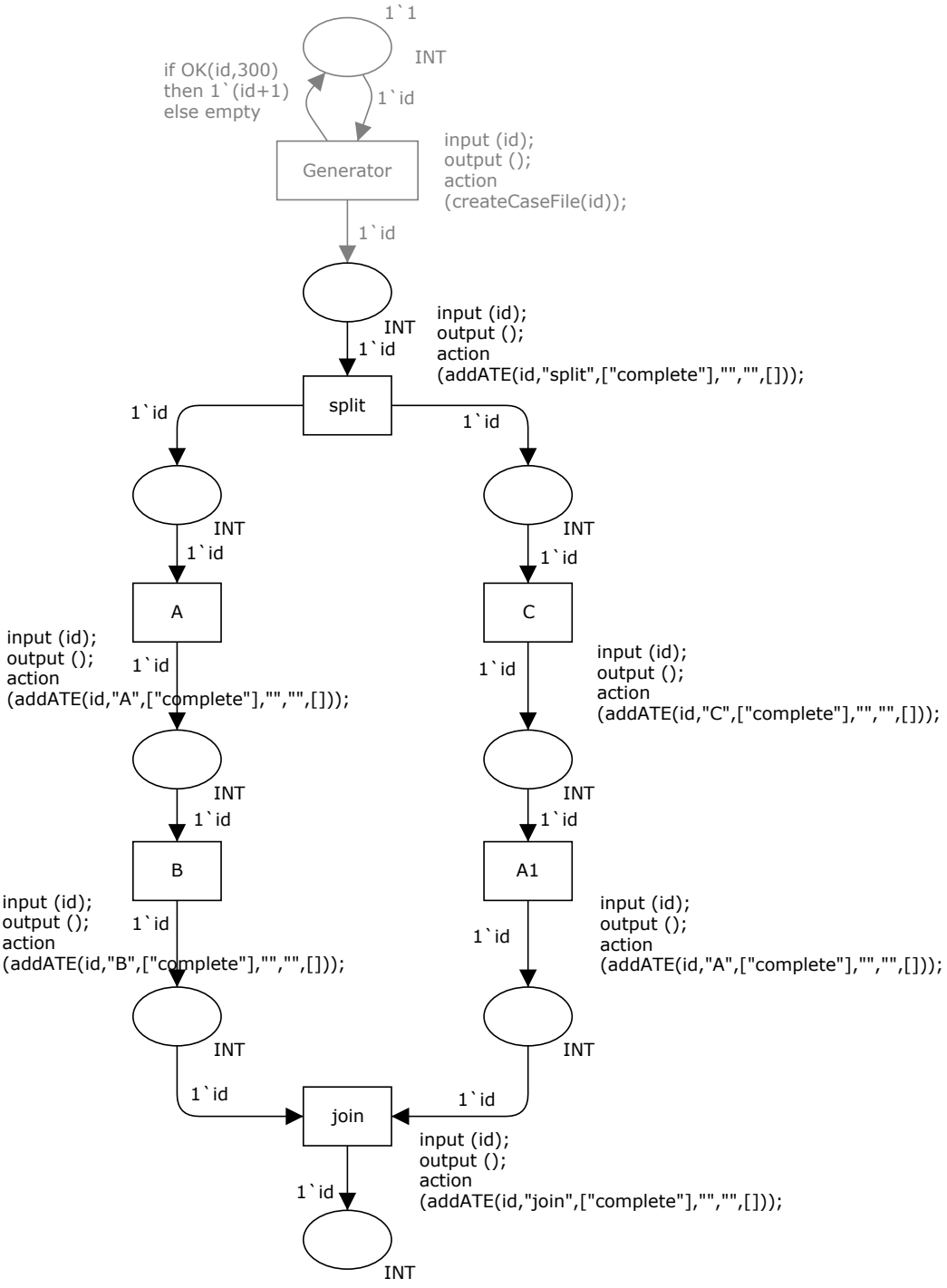


Figure B.55: CPN model for net herbstFig6p38.

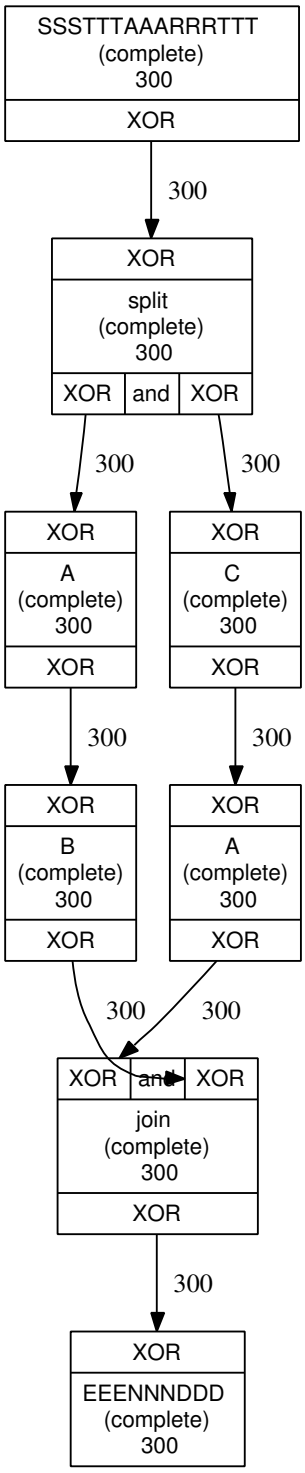


Figure B.56: Heuristic net for herbstFig6p38.

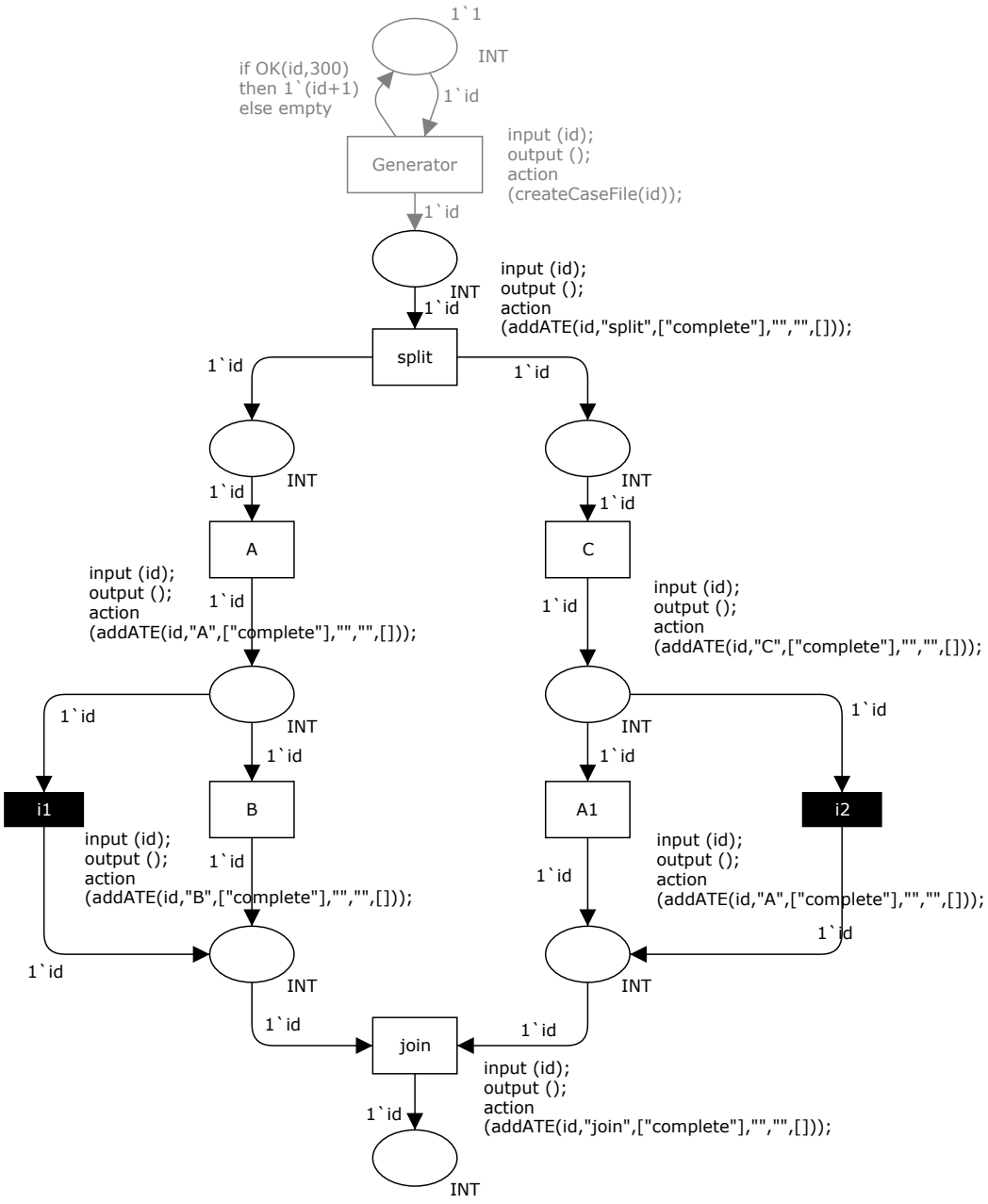


Figure B.57: CPN model for net herbstFig6p39.

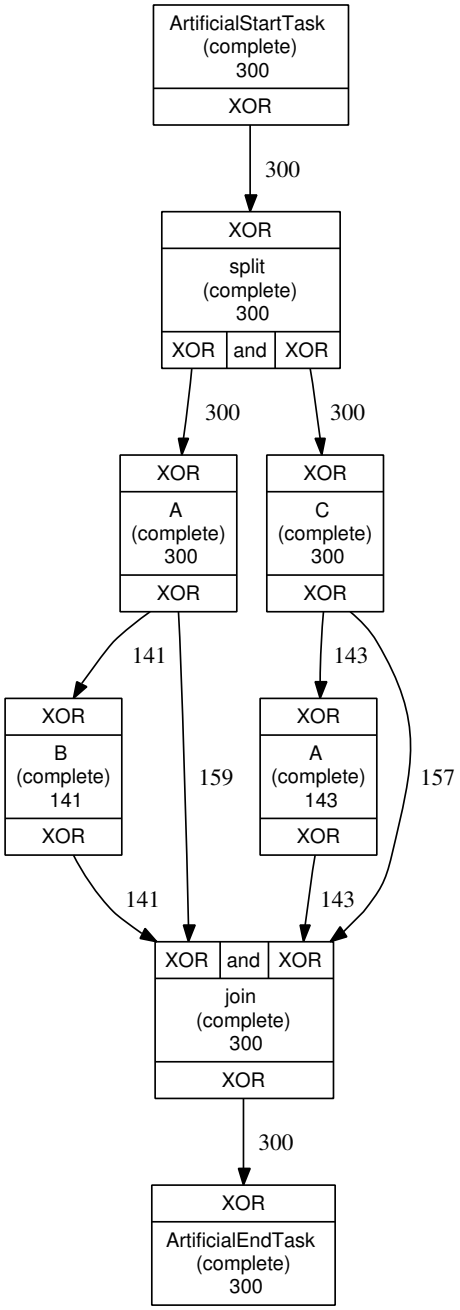


Figure B.58: Heuristic net for herbstFig6p39.

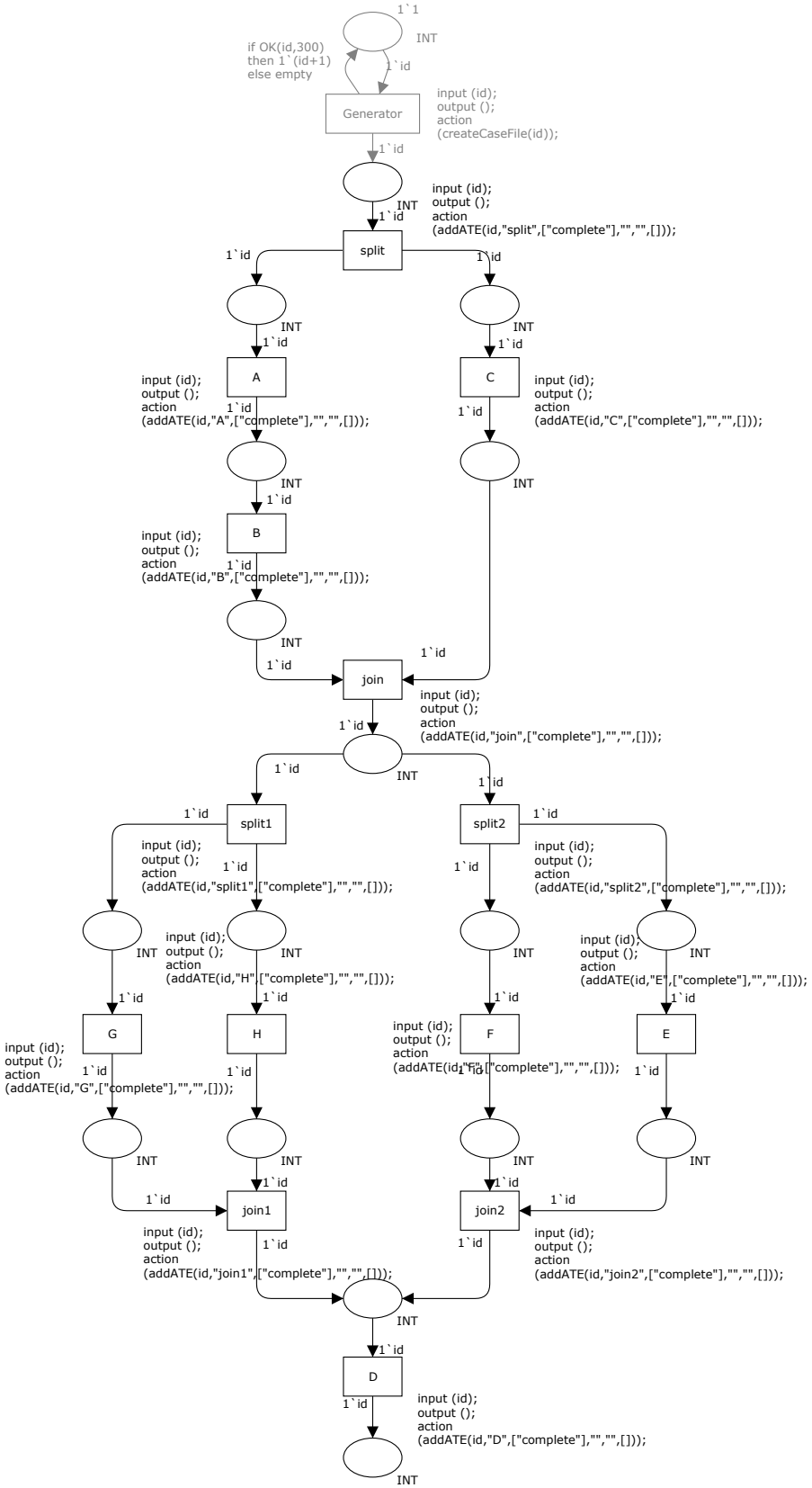


Figure B.59: CPN model for net herbstFig6p41.

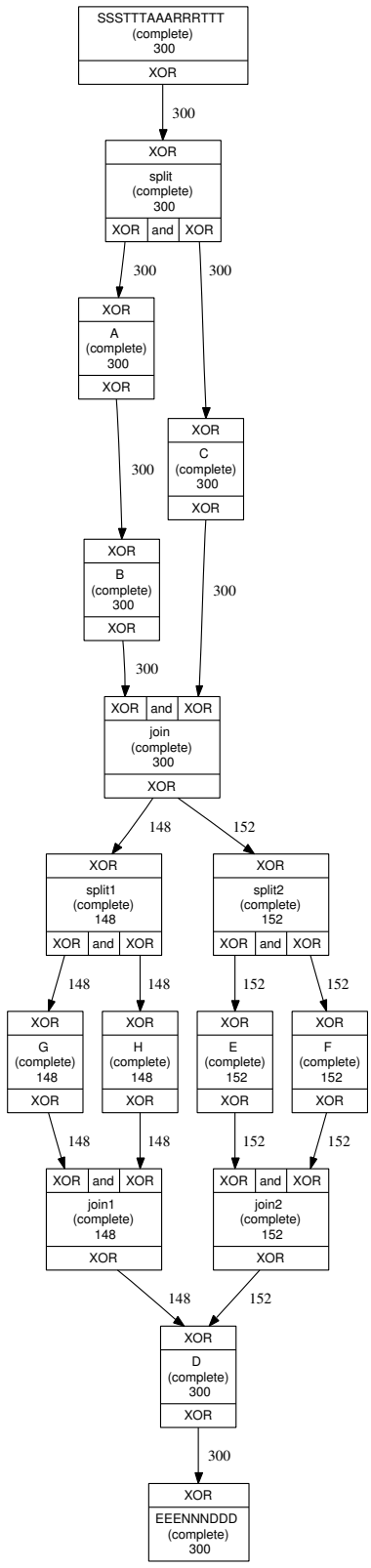


Figure B.60: Heuristic net for herbstFig6p41.

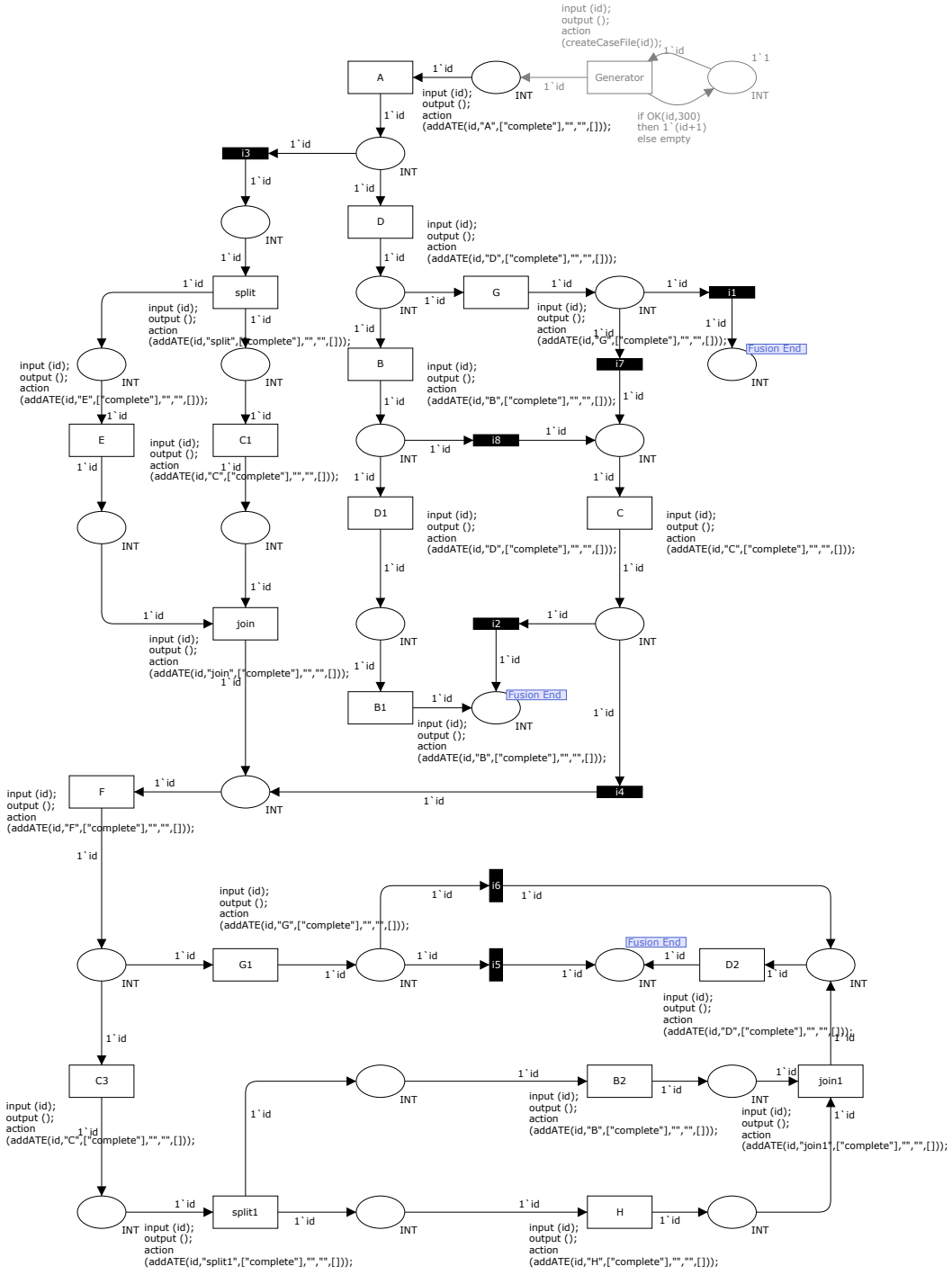


Figure B.61: CPN model for net herbstFig6p42.

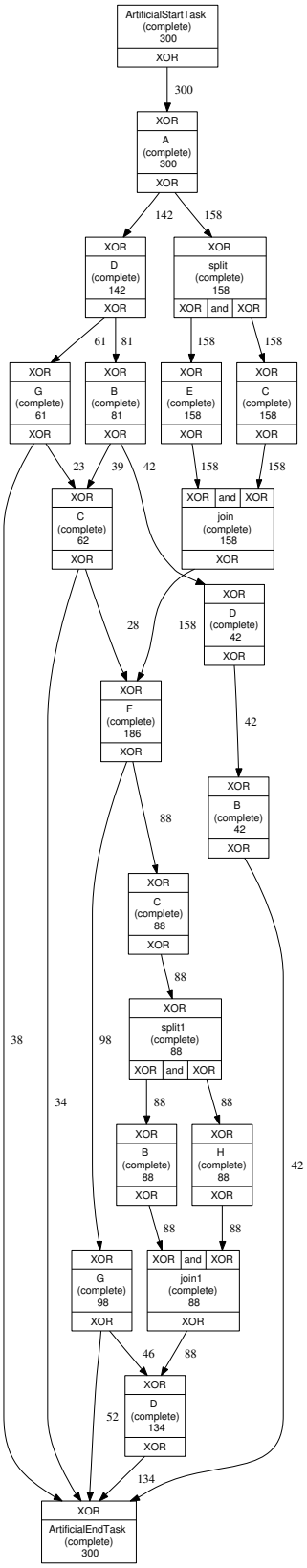


Figure B.62: Heuristic net for herbstFig6p42.

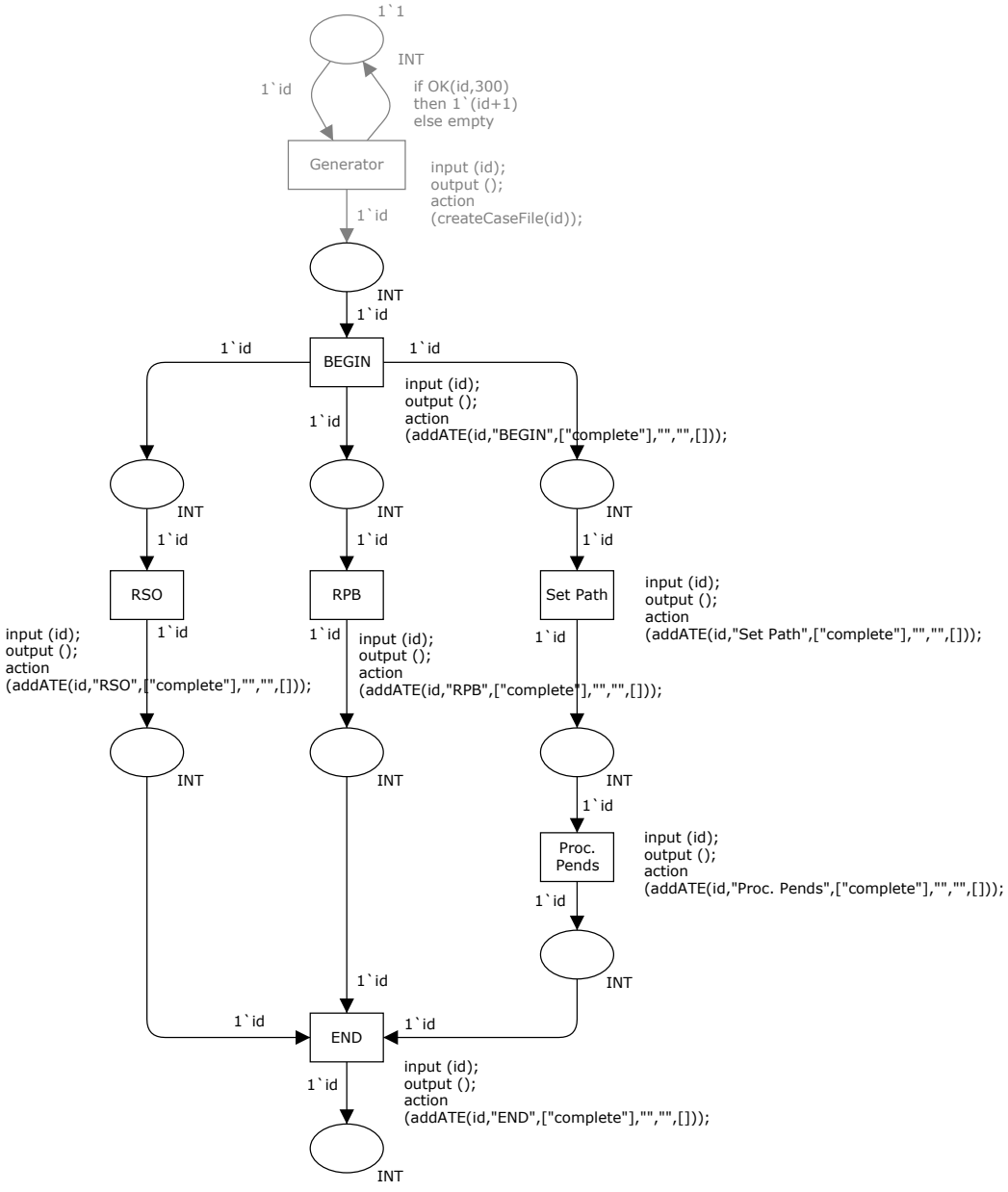


Figure B.63: CPN model for net herbstFig6p45.

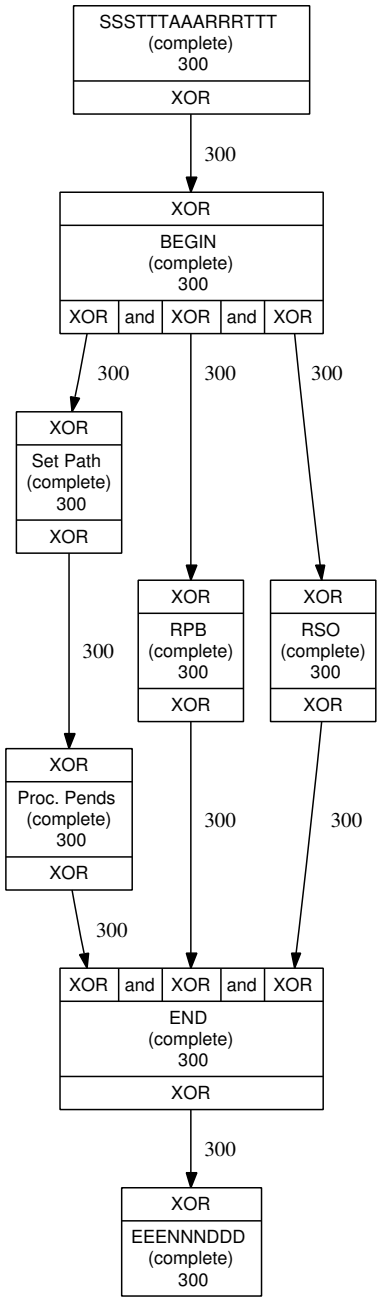


Figure B.64: Heuristic net for herbstFig6p45.

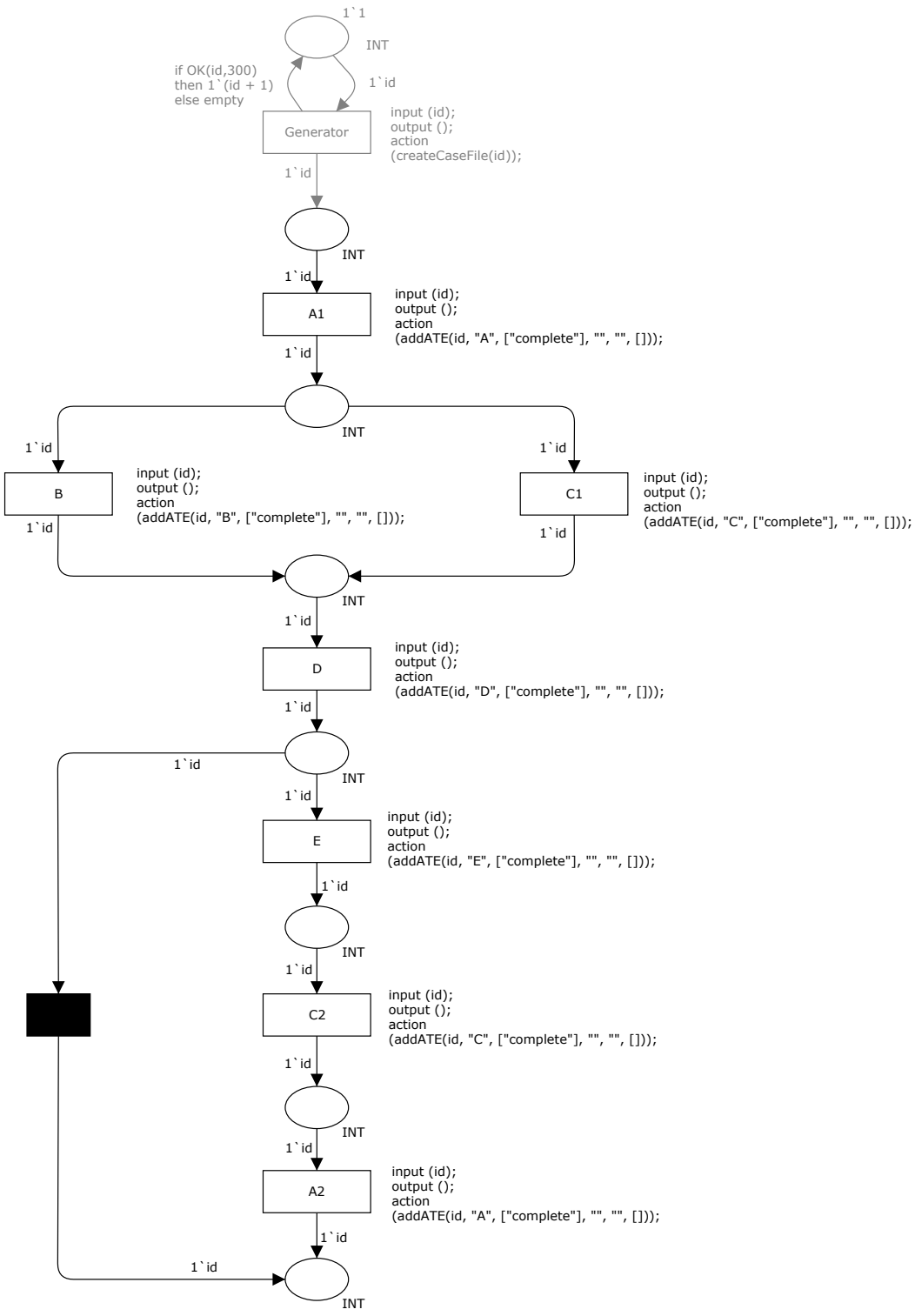


Figure B.65: CPN model for net herbstFig6p9.

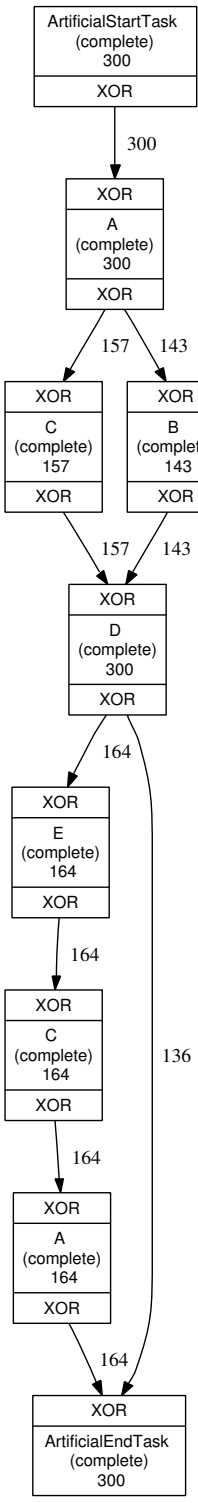


Figure B.66: Heuristic net for herbstFig6p9.

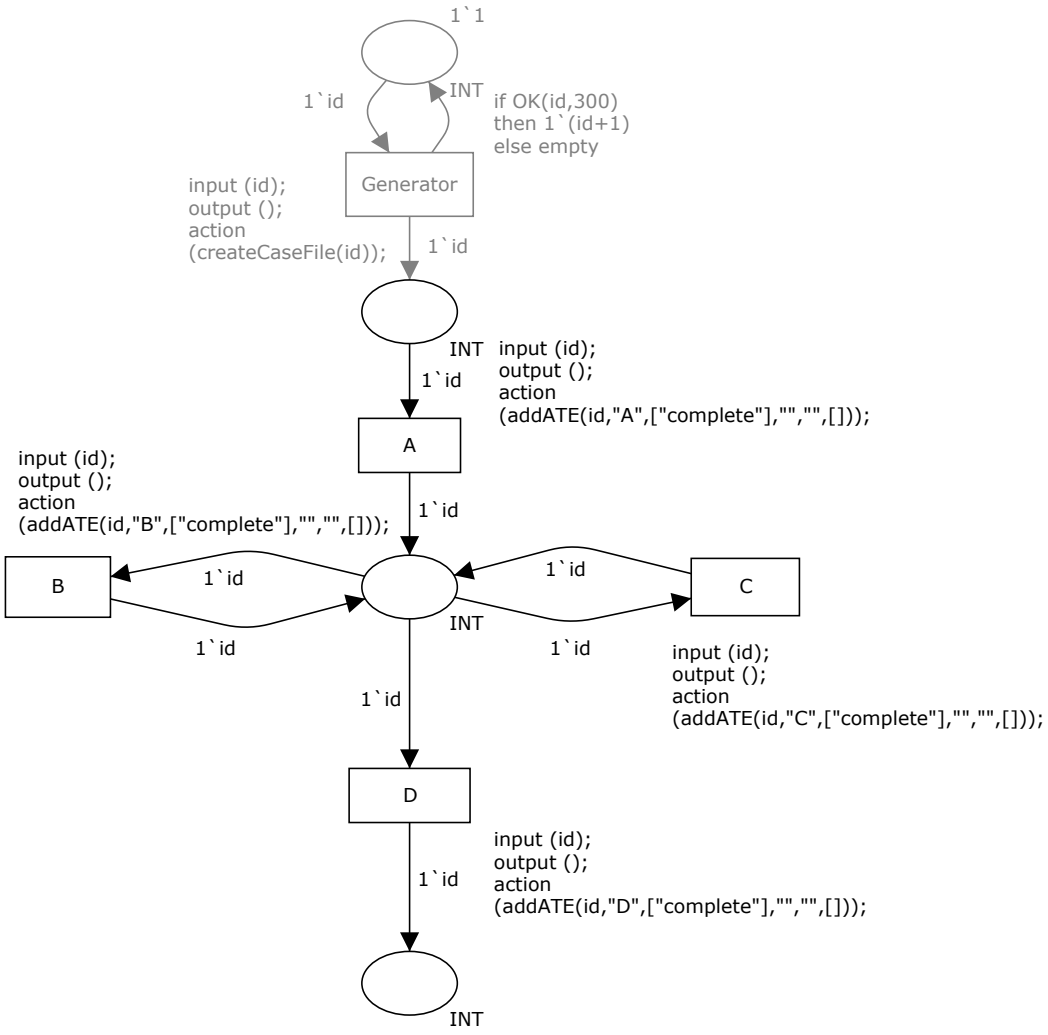


Figure B.67: CPN model for net |1|.

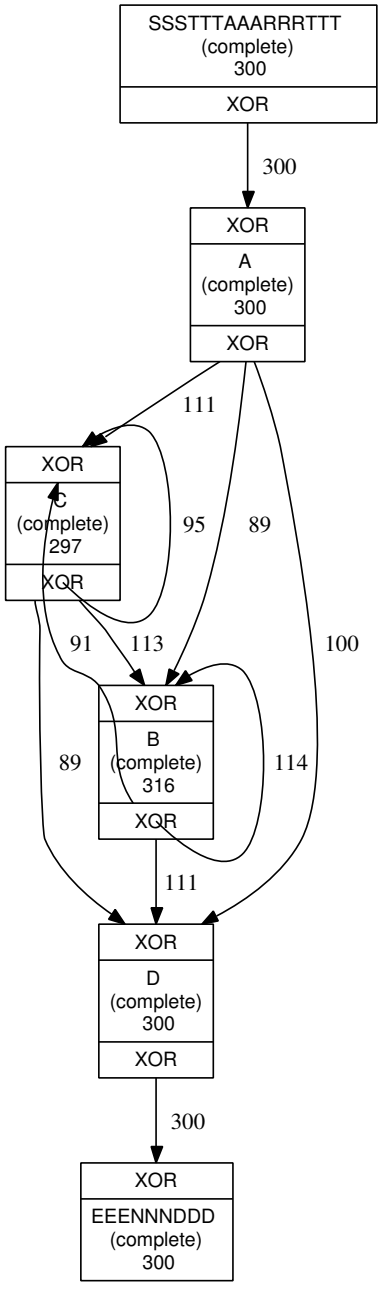


Figure B.68: Heuristic net for l1l.

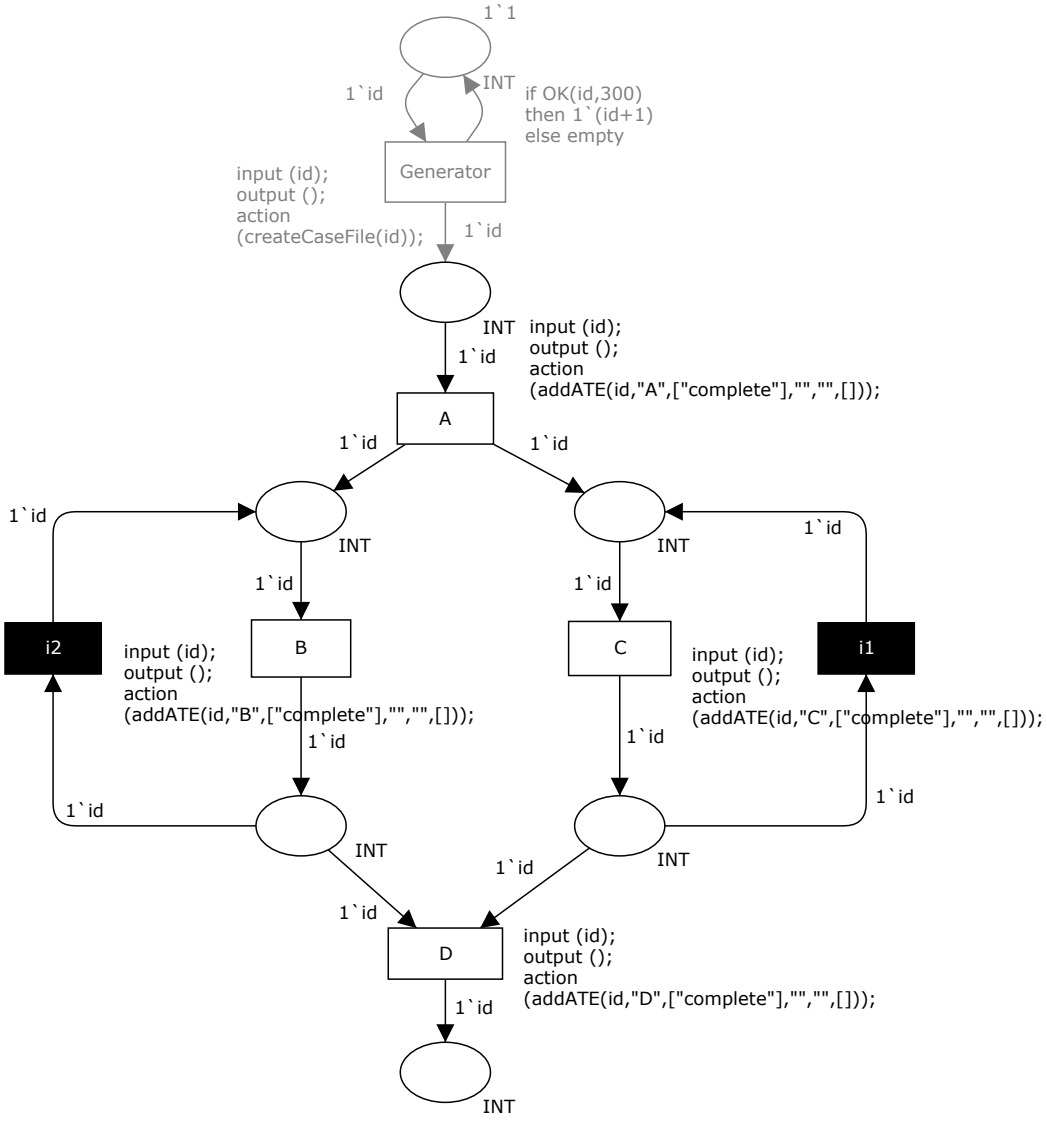


Figure B.69: CPN model for net l1lSkip.

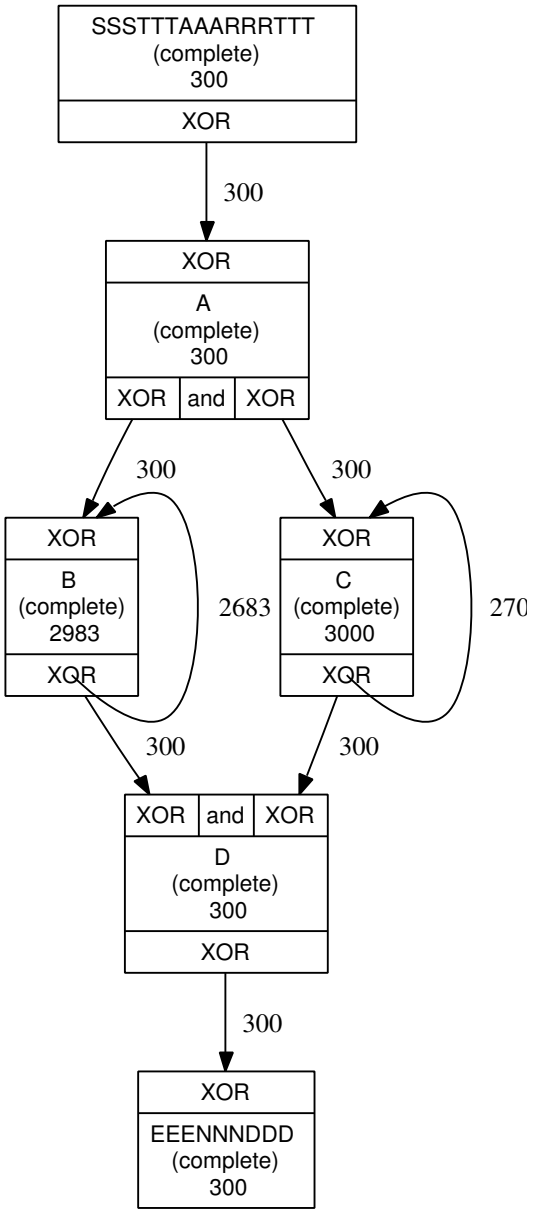


Figure B.70: Heuristic net for l1lSkip.

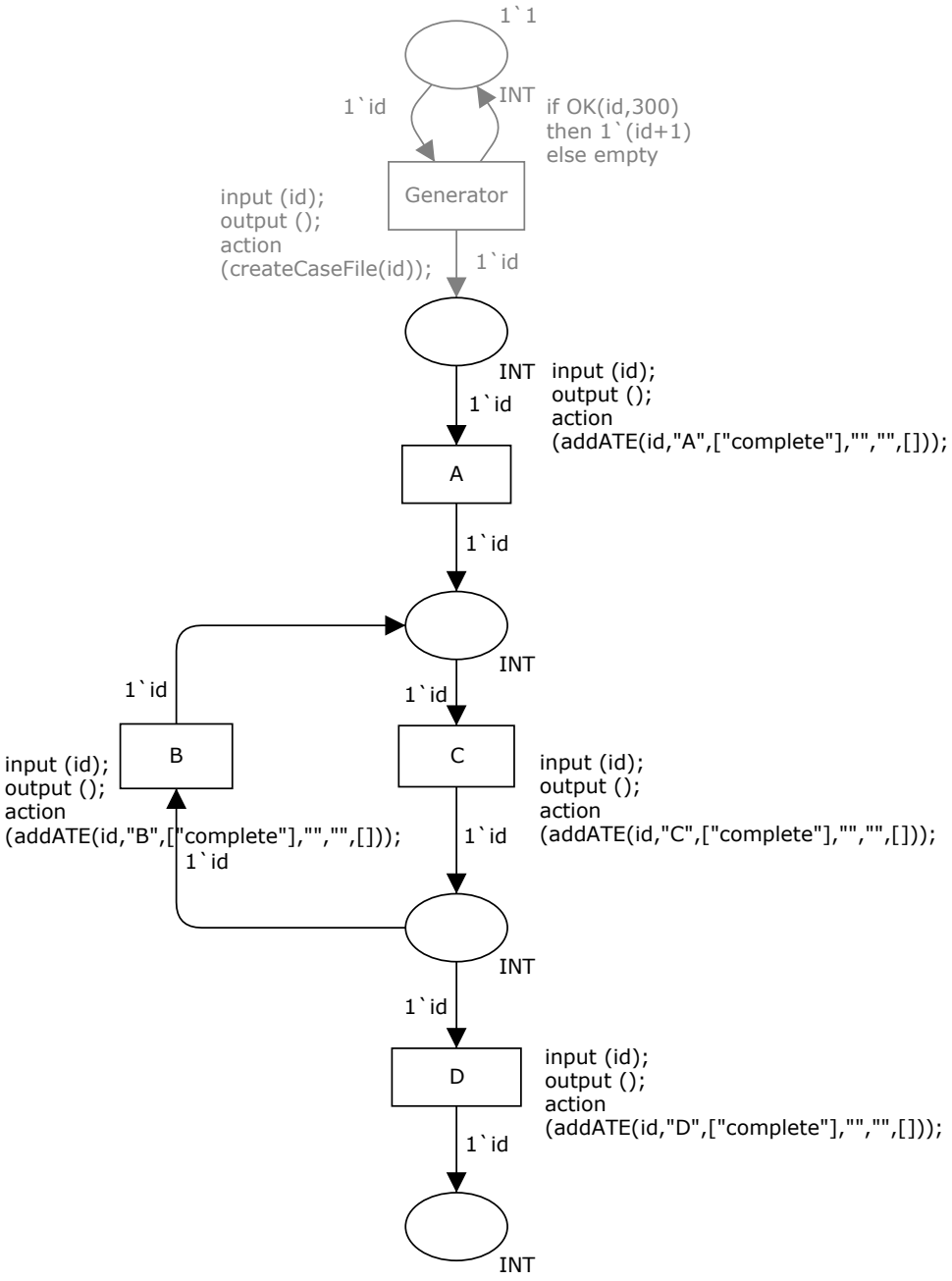


Figure B.71: CPN model for net l2l.

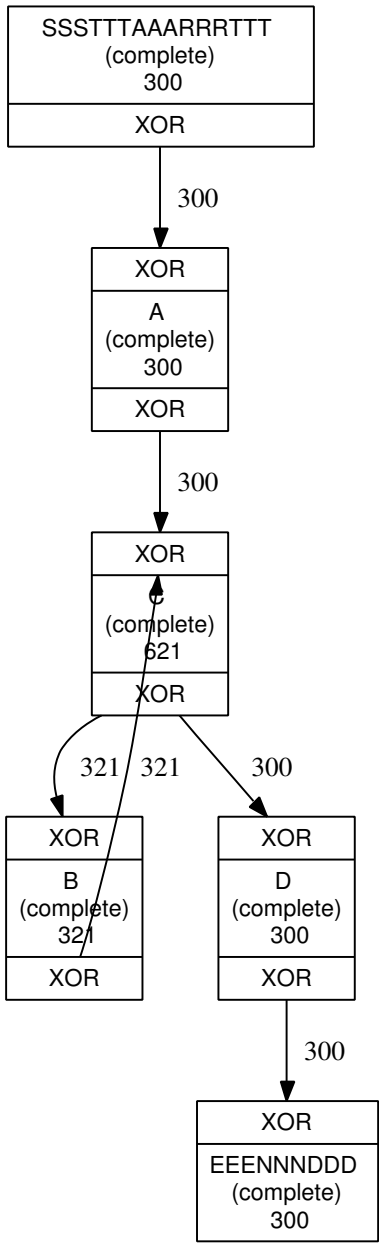


Figure B.72: Heuristic net for l2l.

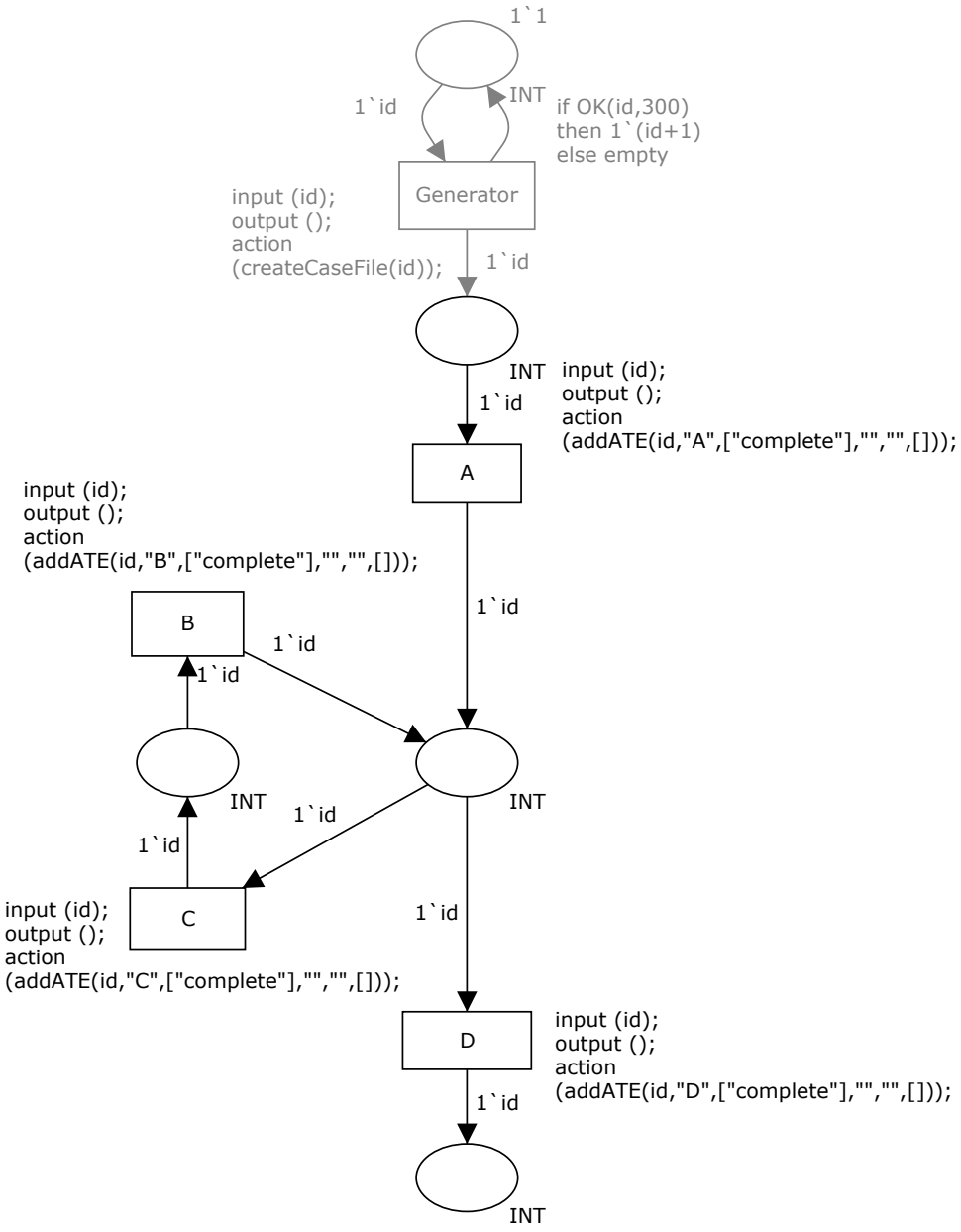


Figure B.73: CPN model for net l2lOptional.

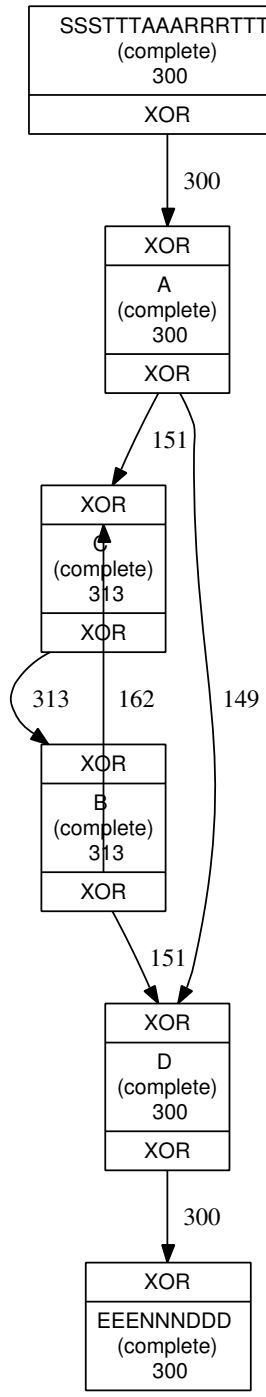


Figure B.74: Heuristic net for l2lOptional.

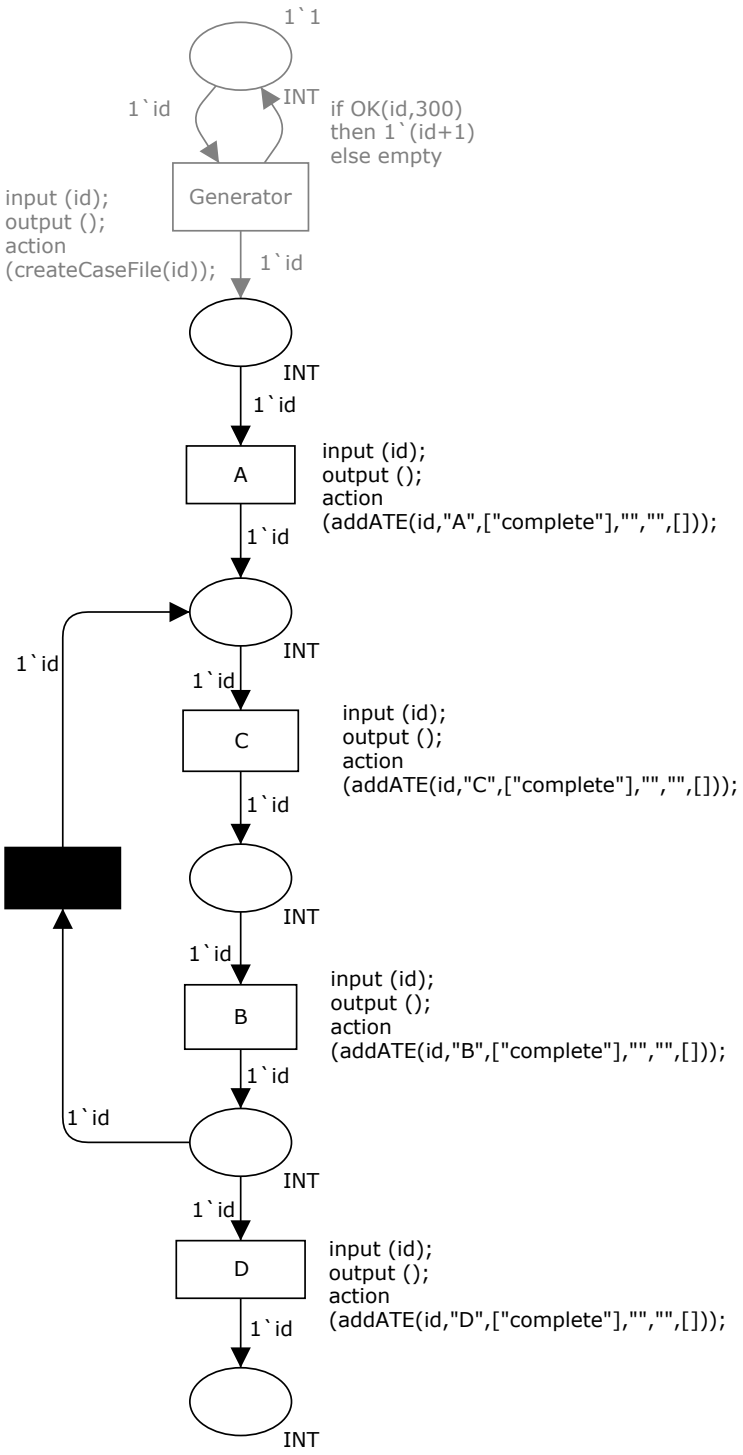


Figure B.75: CPN model for net l2lSkip.

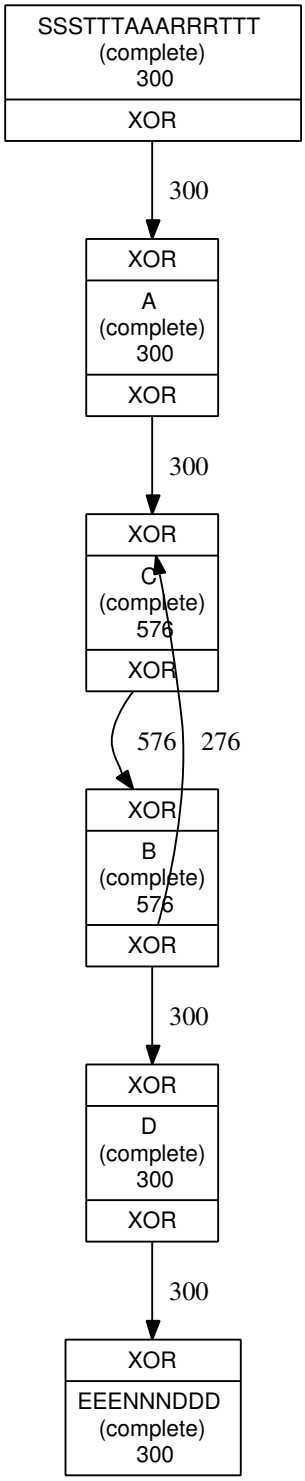


Figure B.76: Heuristic net for l2lSkip.

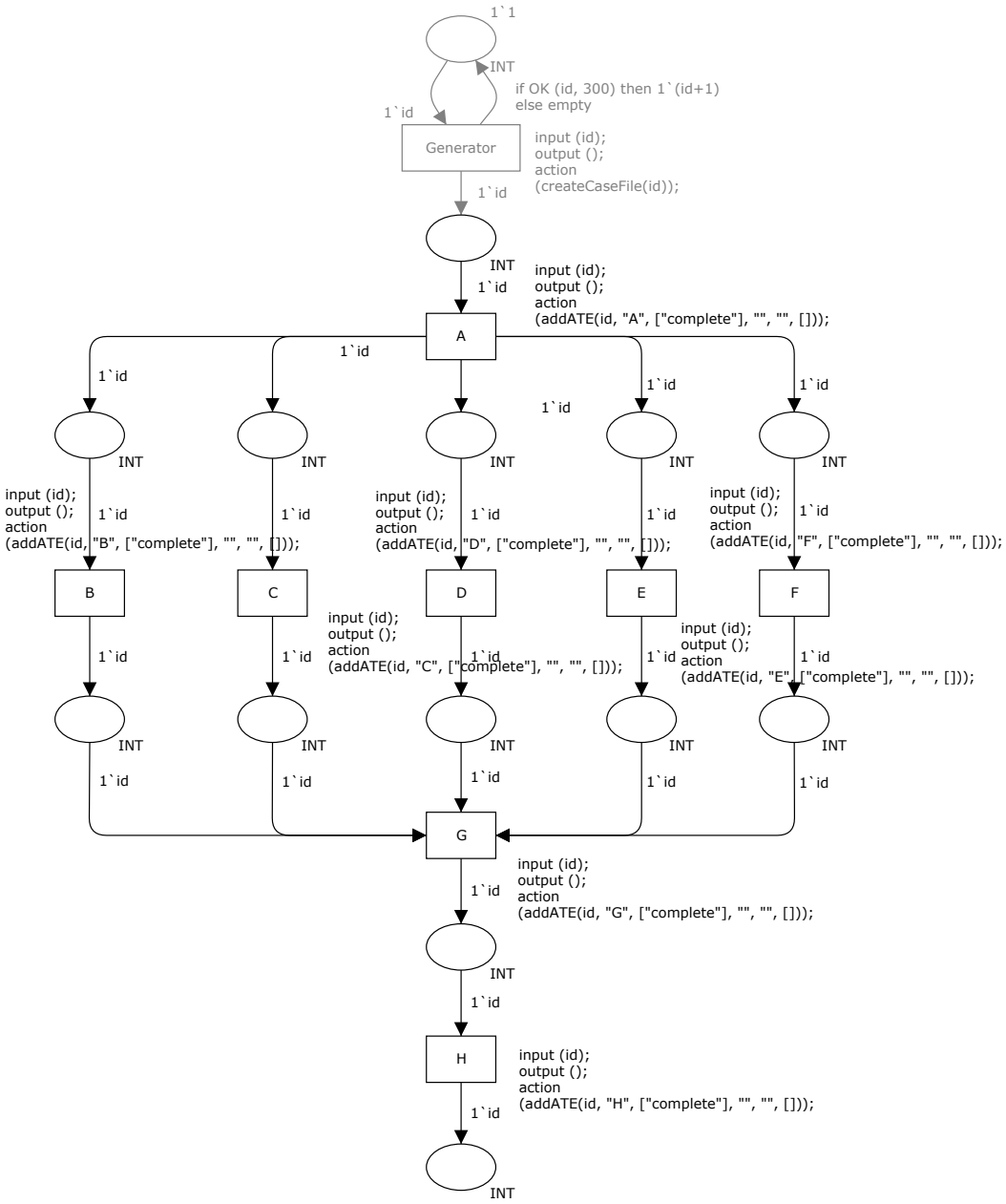


Figure B.77: CPN model for net parallel5.

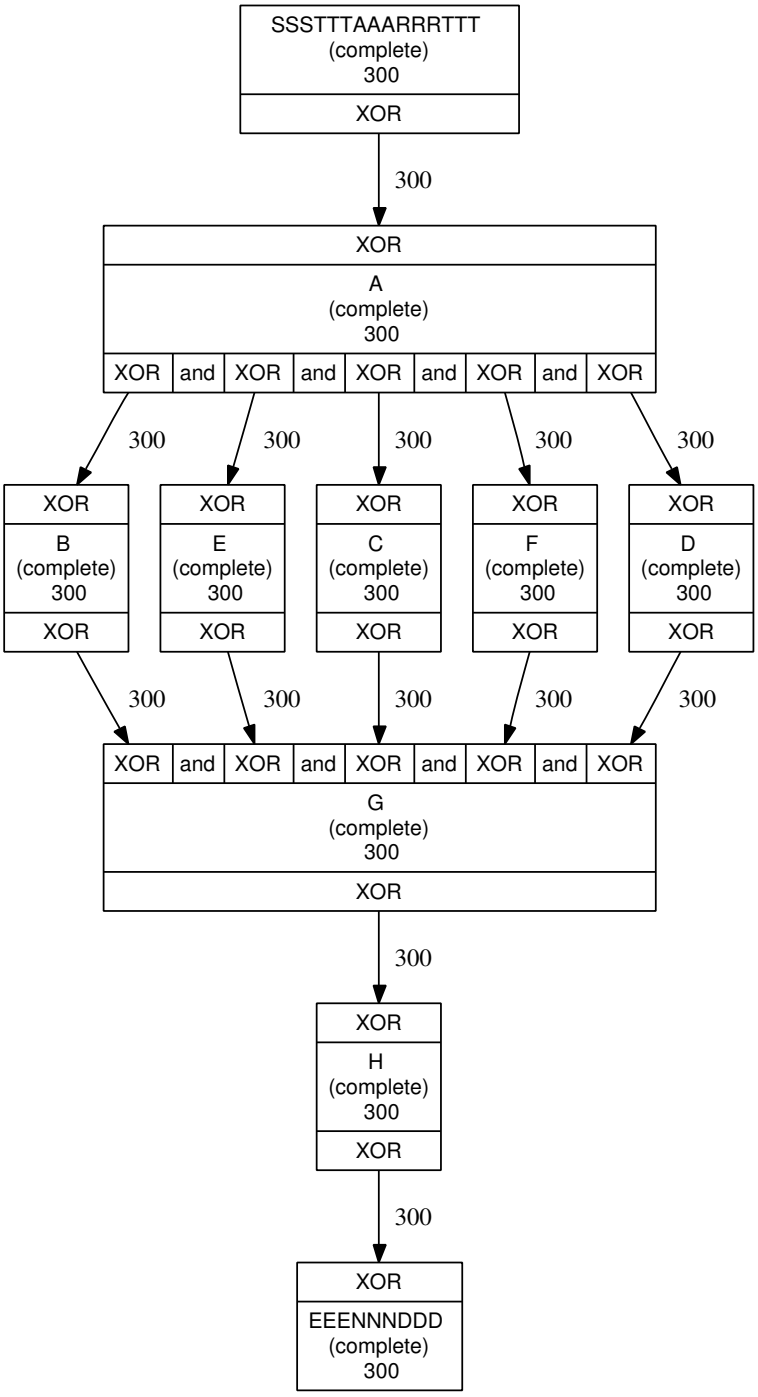


Figure B.78: Heuristic net for parallel5.

Appendix C

All Models Used during the Single-Blind Experiments

This appendix contains the heuristic net (or causal matrix) representation of the original and mined models for the single-blind experiments (cf. Section 8.2). The main characteristics of the original models are summarized in Table C.1. Note that, unlike the experiments with known models (cf. Appendix B), the models presented here also contain unbalanced AND-split/join points. This means that, for some of these models, there is not a one-to-one correspondence between the AND-split points and the AND-join ones.

Net	Figure	Sequence	Choice	Parallelism	Length-One Loop	Length-Two Loop	Structured Loop	Arbitrary Loop	Invisible Tasks	Unbalanced AND-split/join
g2	C.1	✓	✓	✓		✓	✓		✓	
g3	C.2	✓	✓	✓			✓	✓	✓	
g4	C.4	✓	✓	✓		✓				✓
g5	C.6	✓	✓	✓				✓	✓	
g6	C.7	✓	✓	✓		✓			✓	
g7	C.8	✓	✓	✓			✓		✓	
g8	C.9	✓	✓	✓		✓	✓		✓	✓
g9	C.11	✓	✓	✓		✓	✓		✓	
g10	C.13	✓	✓	✓				✓	✓	
g12	C.15	✓	✓	✓		✓		✓	✓	
g13	C.16	✓	✓	✓	✓	✓			✓	✓
g14	C.18	✓	✓	✓			✓		✓	✓
g15	C.20	✓	✓		✓	✓			✓	
g19	C.22	✓	✓	✓		✓			✓	✓
g20	C.24	✓	✓		✓	✓		✓	✓	
g21	C.25	✓	✓					✓	✓	
g22	C.26	✓	✓	✓			✓		✓	✓
g23	C.28	✓	✓	✓		✓				✓
g24	C.30	✓	✓	✓				✓	✓	✓
g25	C.32	✓	✓	✓	✓				✓	

Table C.1: Nets for the single-blind experiments.

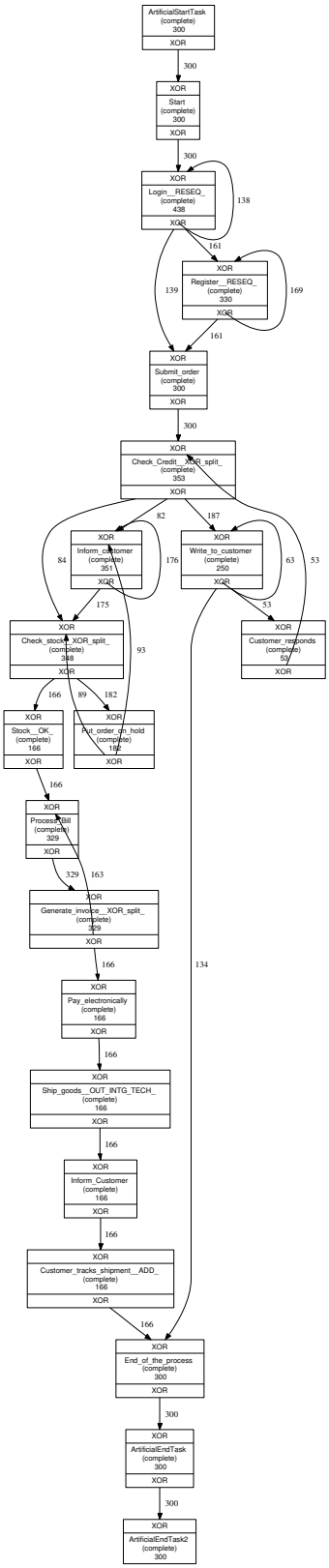


Figure C.1: Original model for net g2. The mined model is identical to the original model.

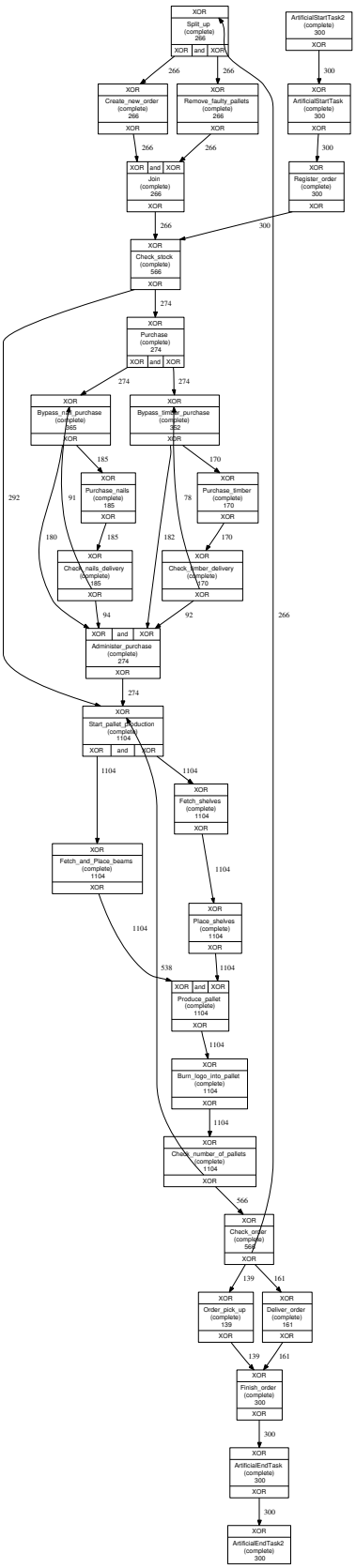


Figure C.2: Original model for net g3.

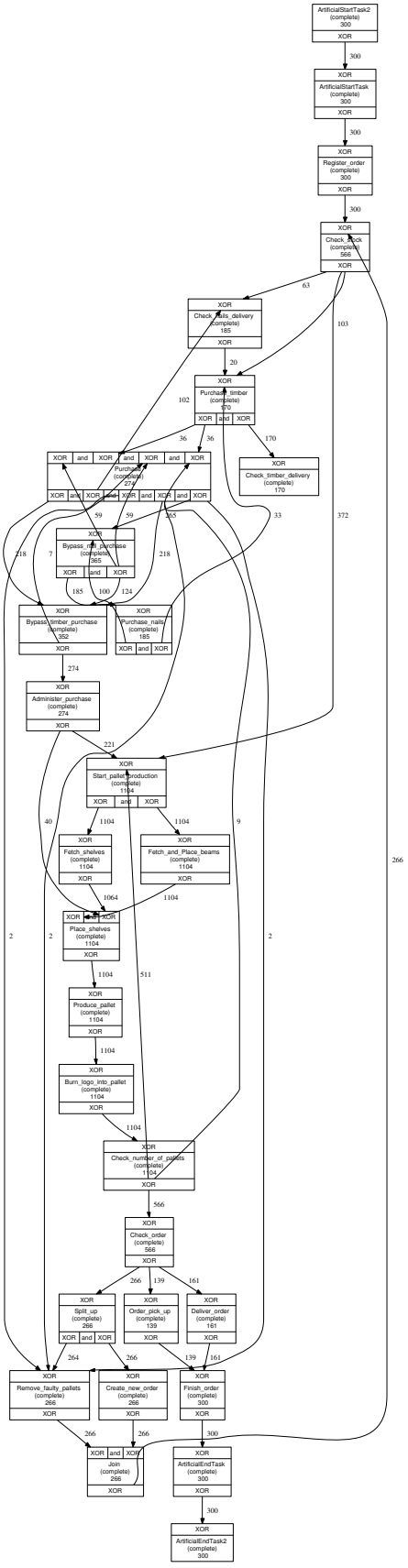


Figure C.3: Mined model for net g3.

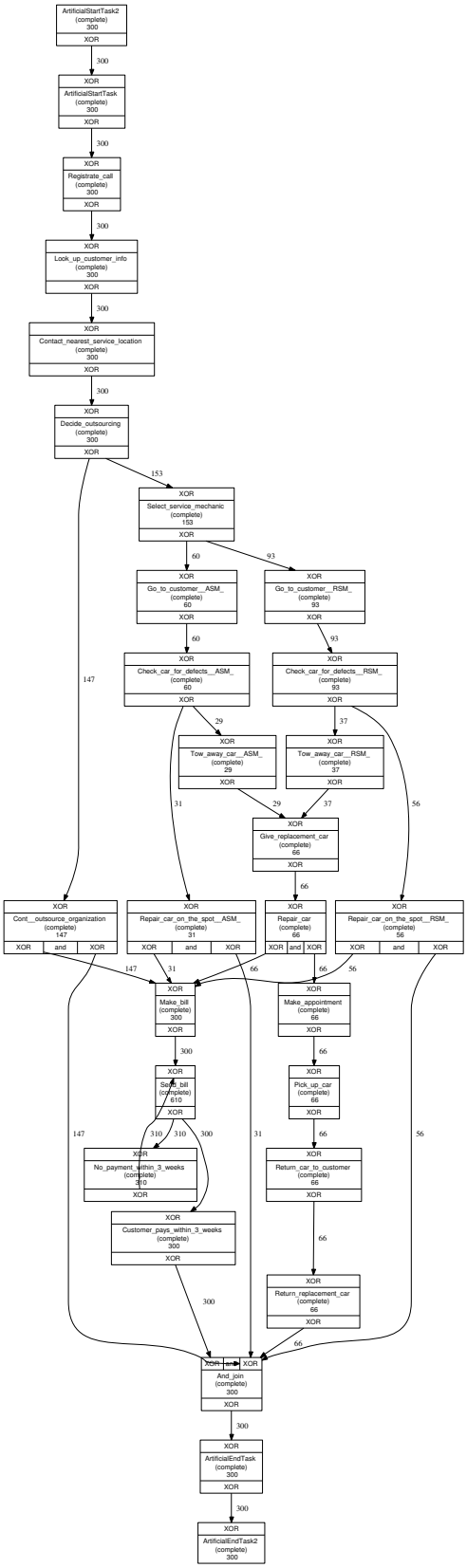


Figure C.4: Original model for net g4.

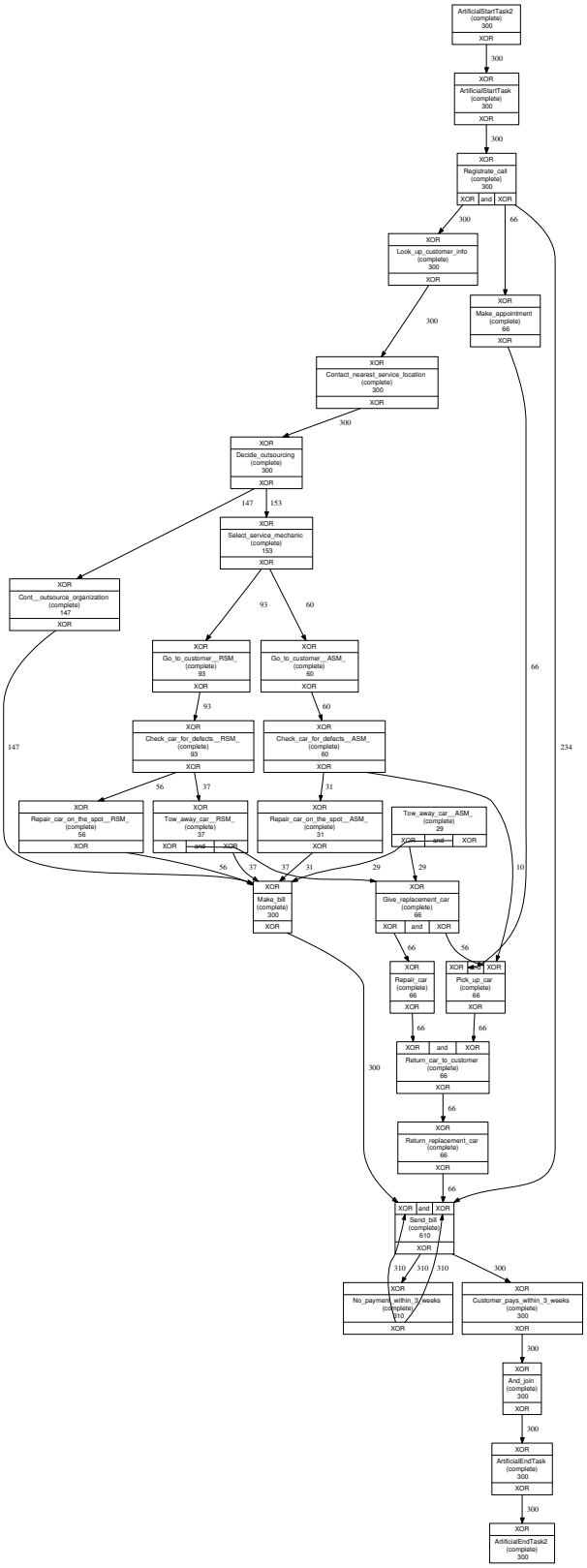


Figure C.5: Mined model for net **g4**.

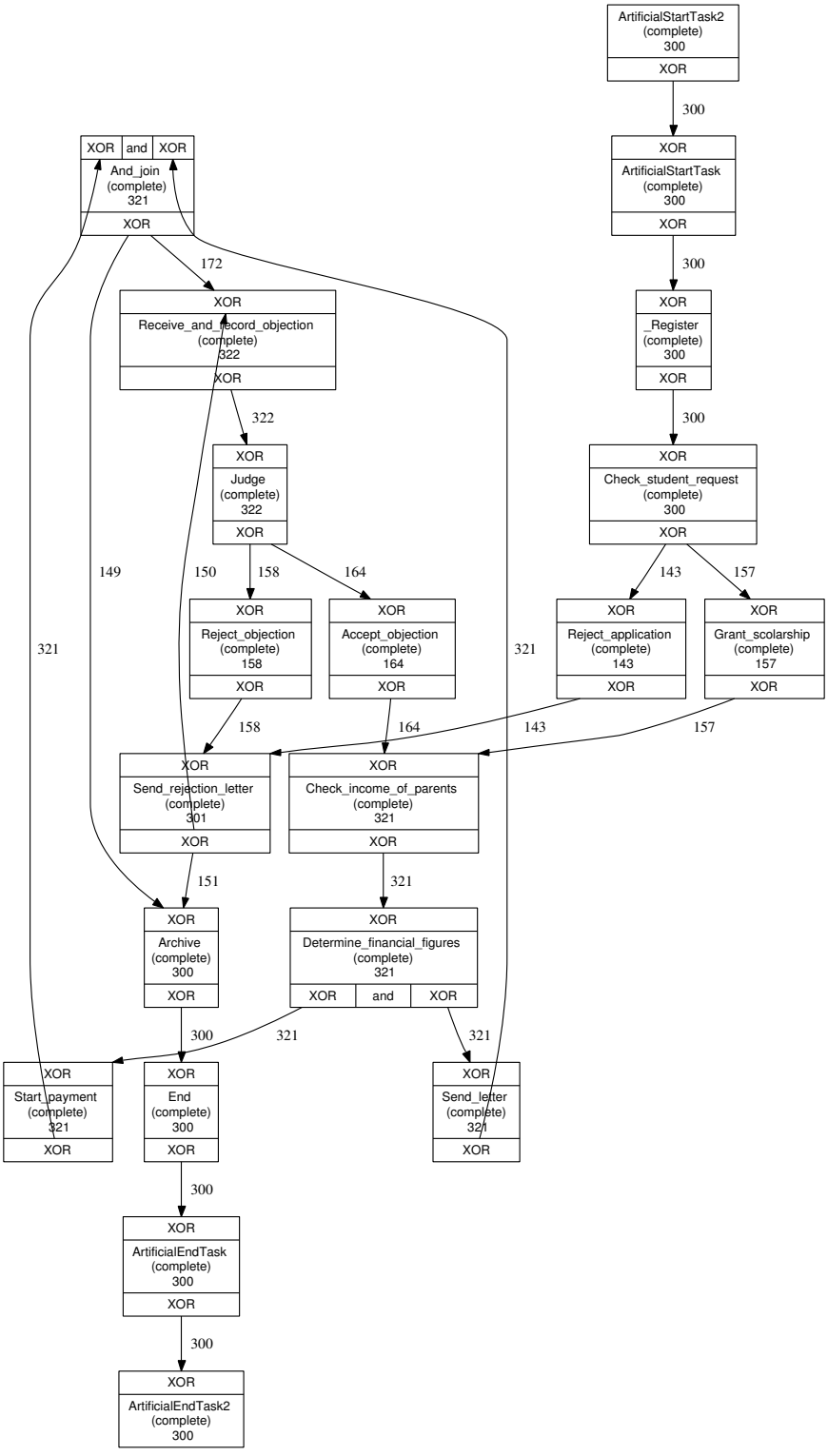


Figure C.6: Original model for net g5. The mined model is identical to the original model.

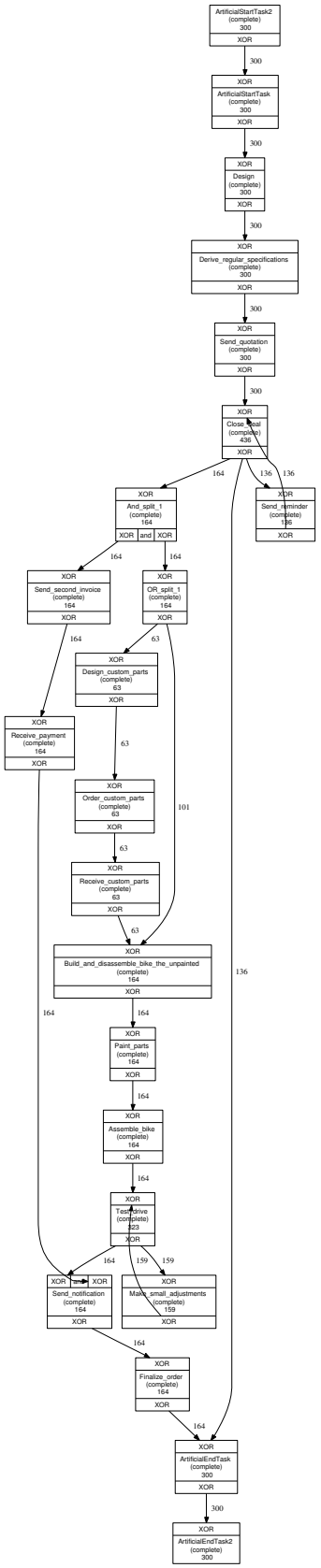


Figure C.7: Original model for net g6. The mined model is identical to the original model.

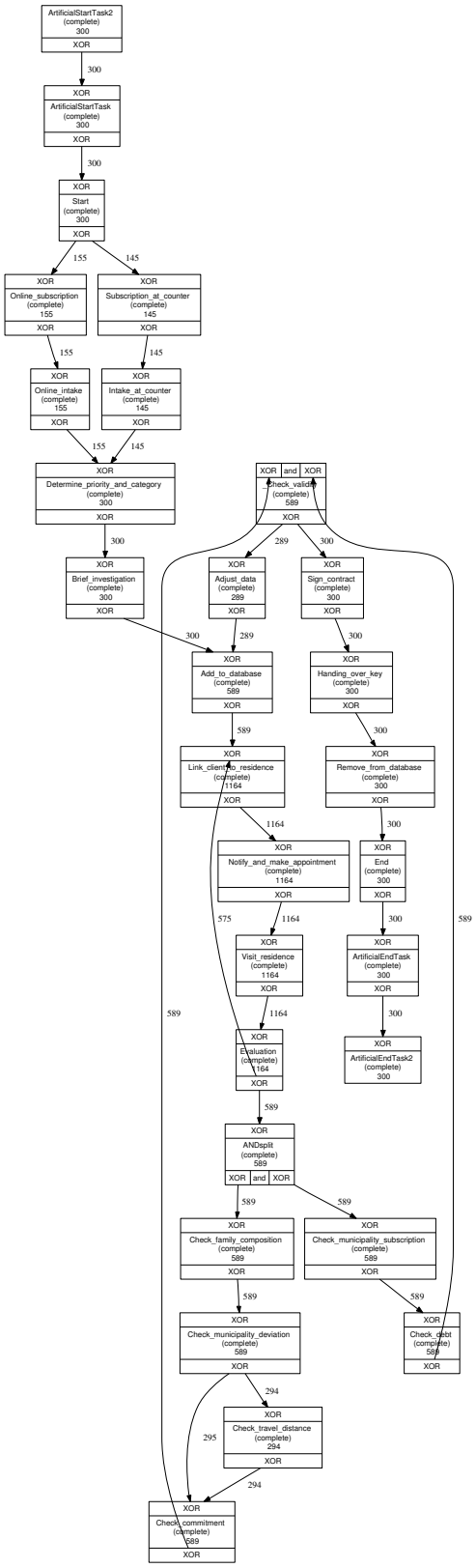


Figure C.8: Original model for net g7. The mined model is identical to the original model.

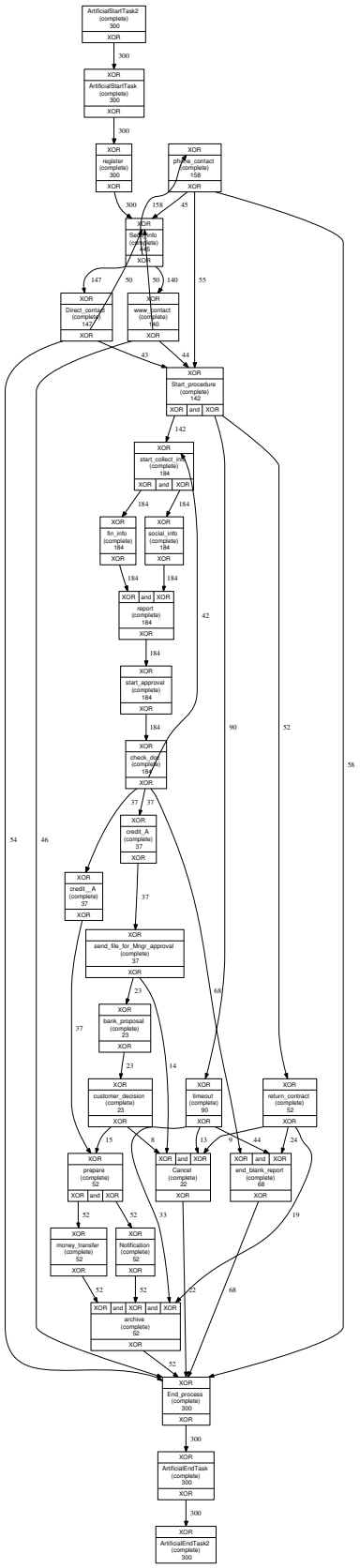


Figure C.9: Original model for net g8.

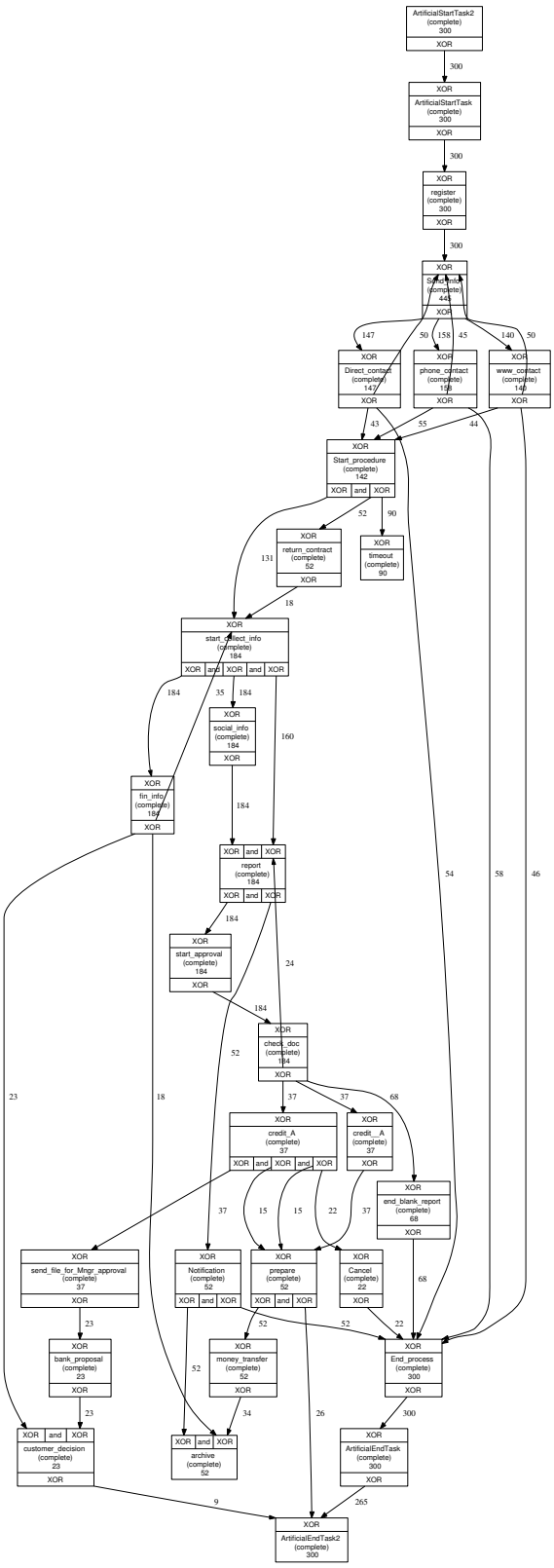


Figure C.10: Mined model for net g8.

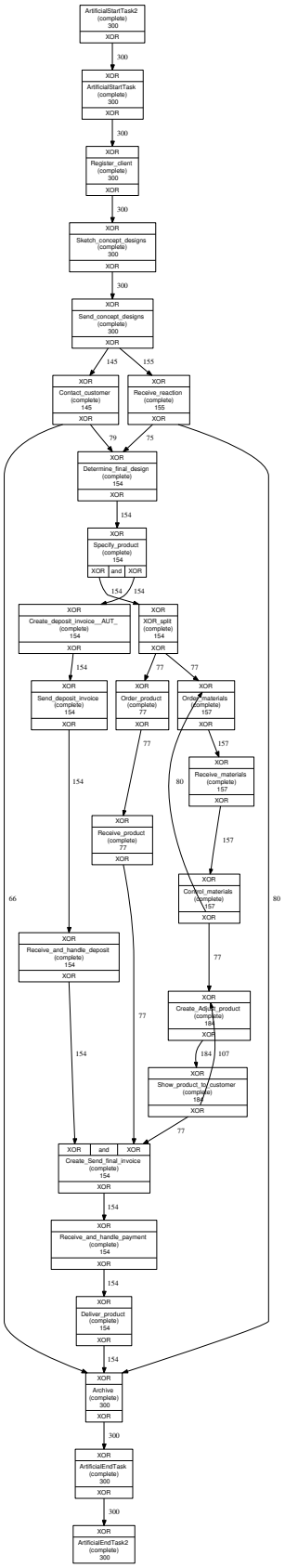


Figure C.11: Original model for net g9.

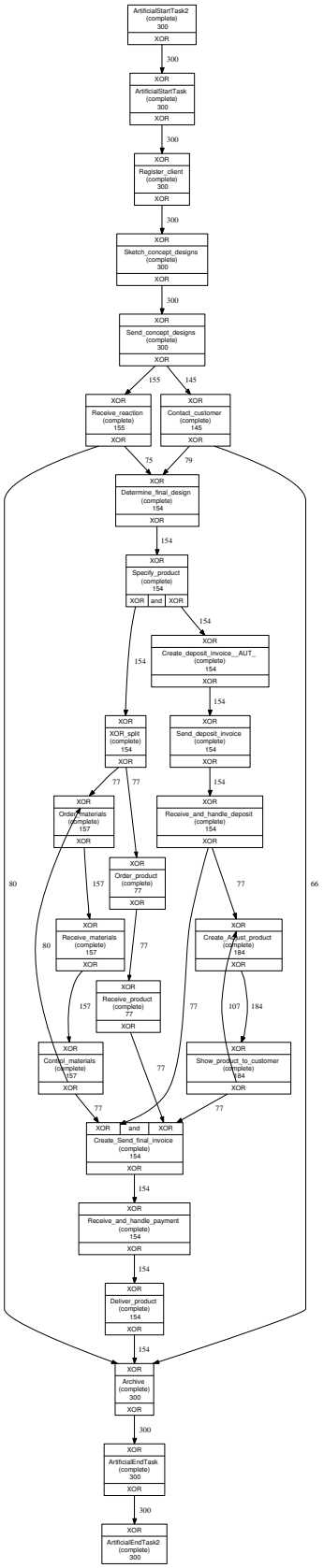


Figure C.12: Mined model for net g9.

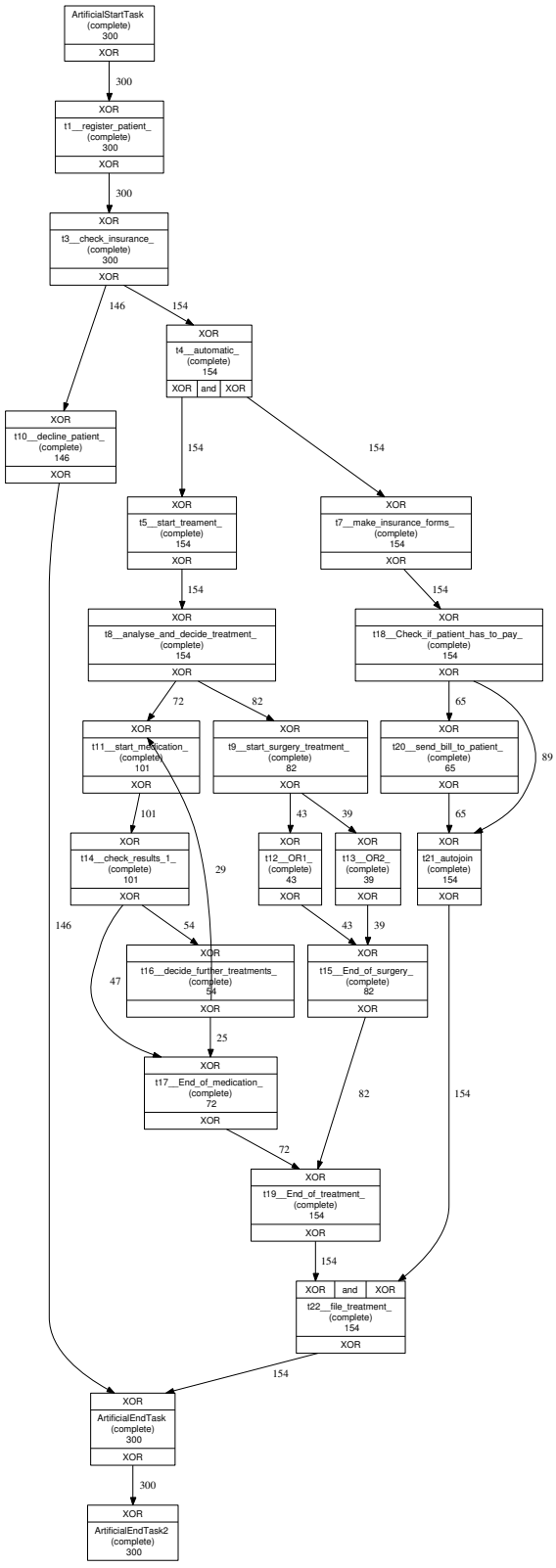


Figure C.13: Original model for net g10.

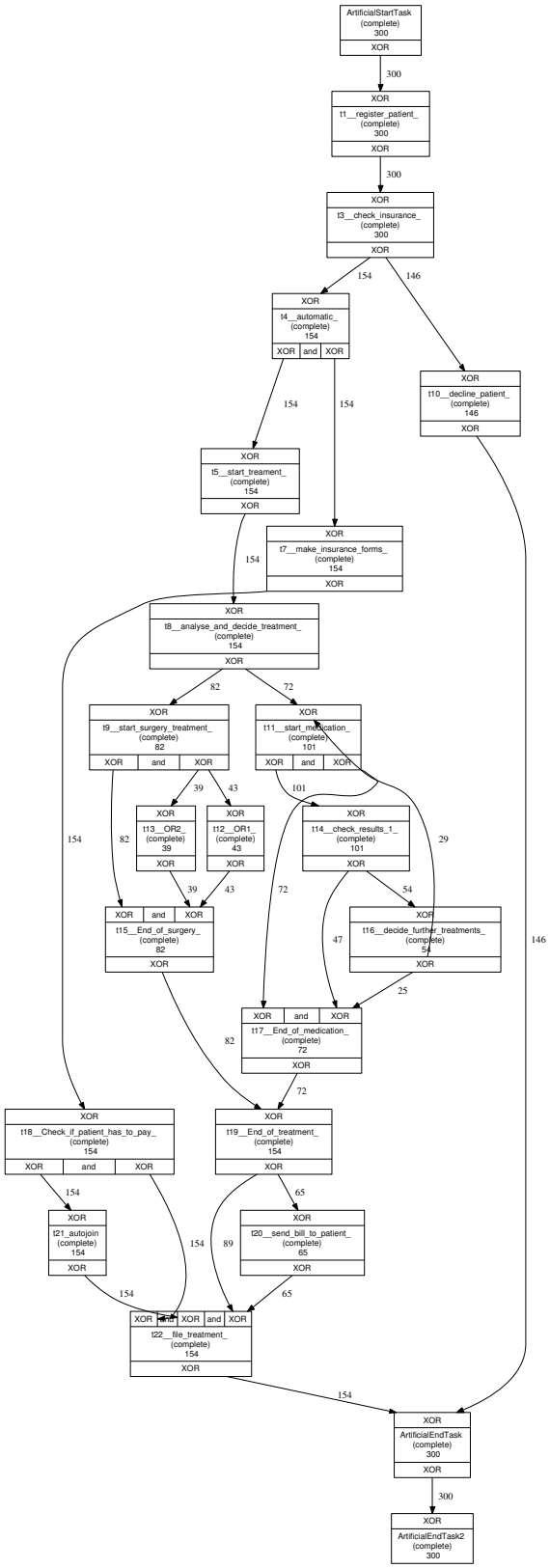


Figure C.14: Mined model for net g10.

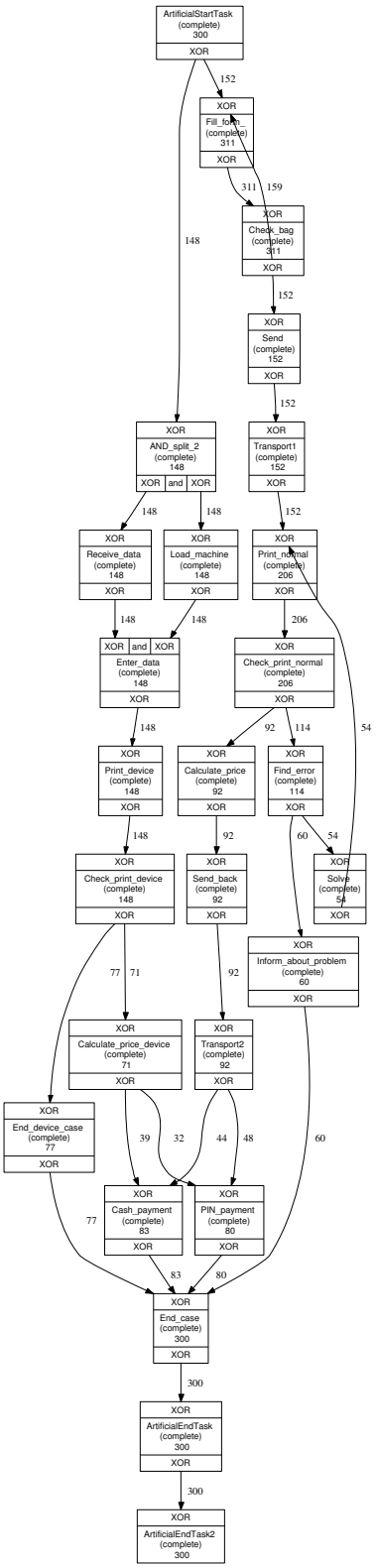


Figure C.15: Original model for net g12. The mined model is identical to the original model.

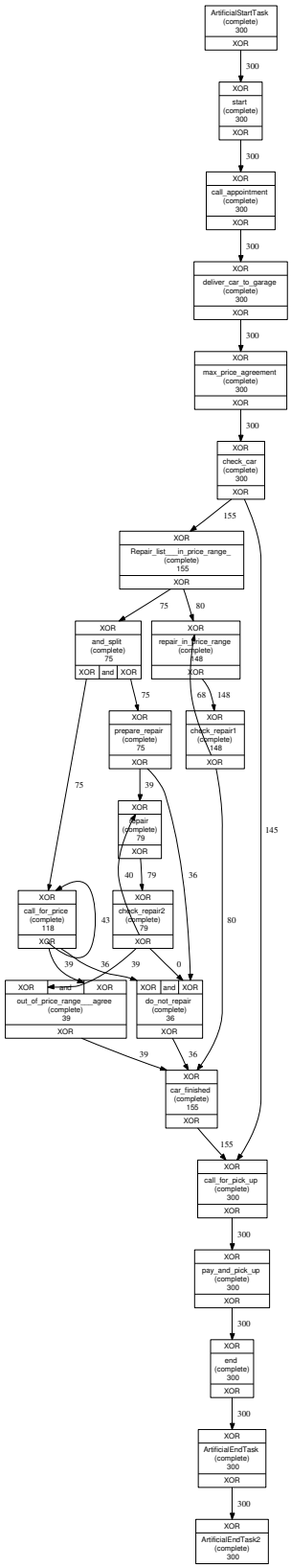


Figure C.16: Original model for net g13.

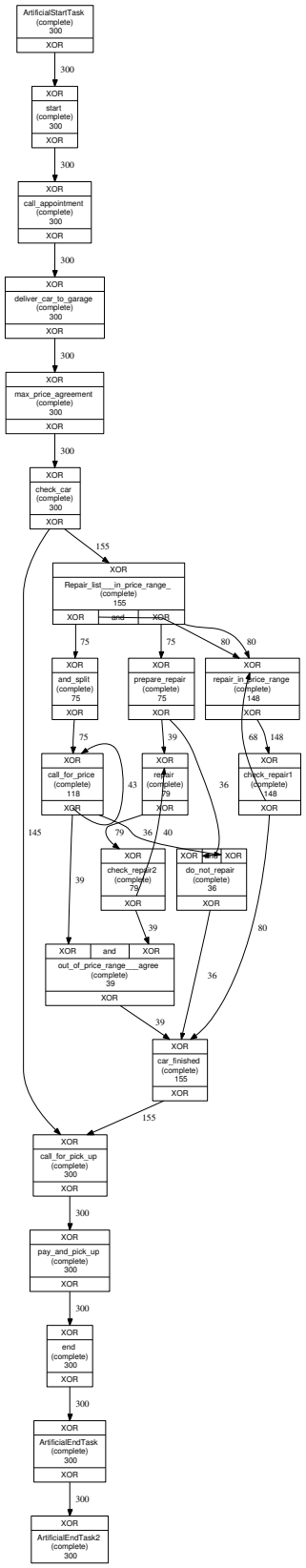


Figure C.17: Mined model for net g13.

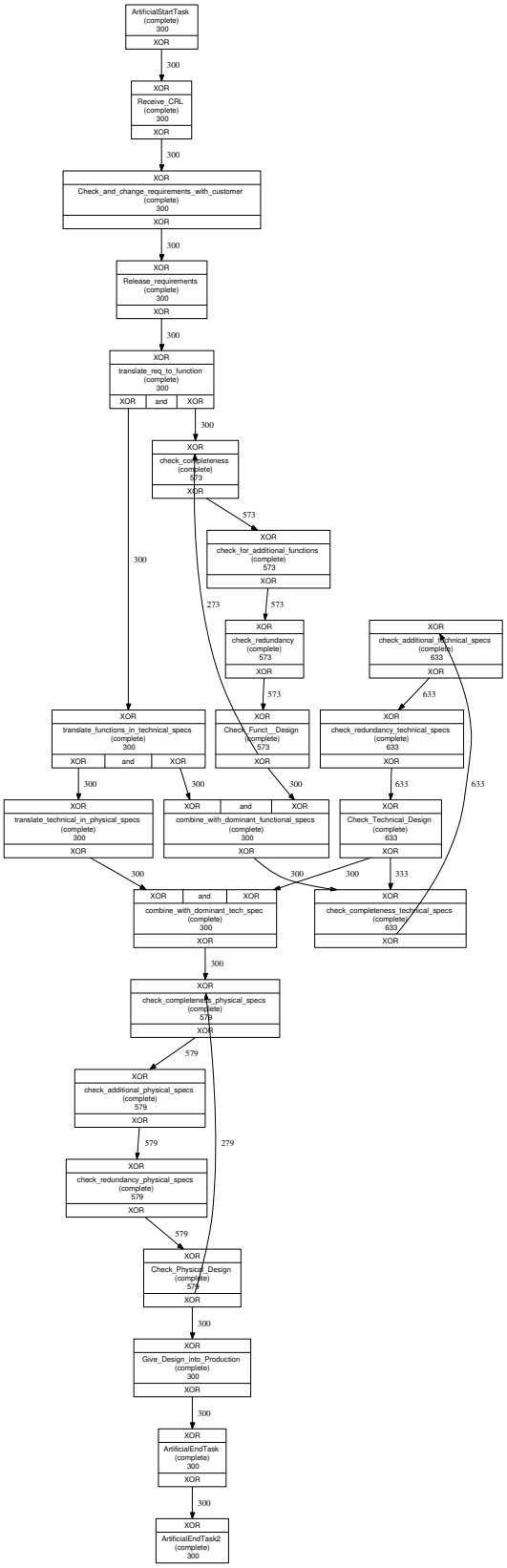


Figure C.18: Original model for net g14.

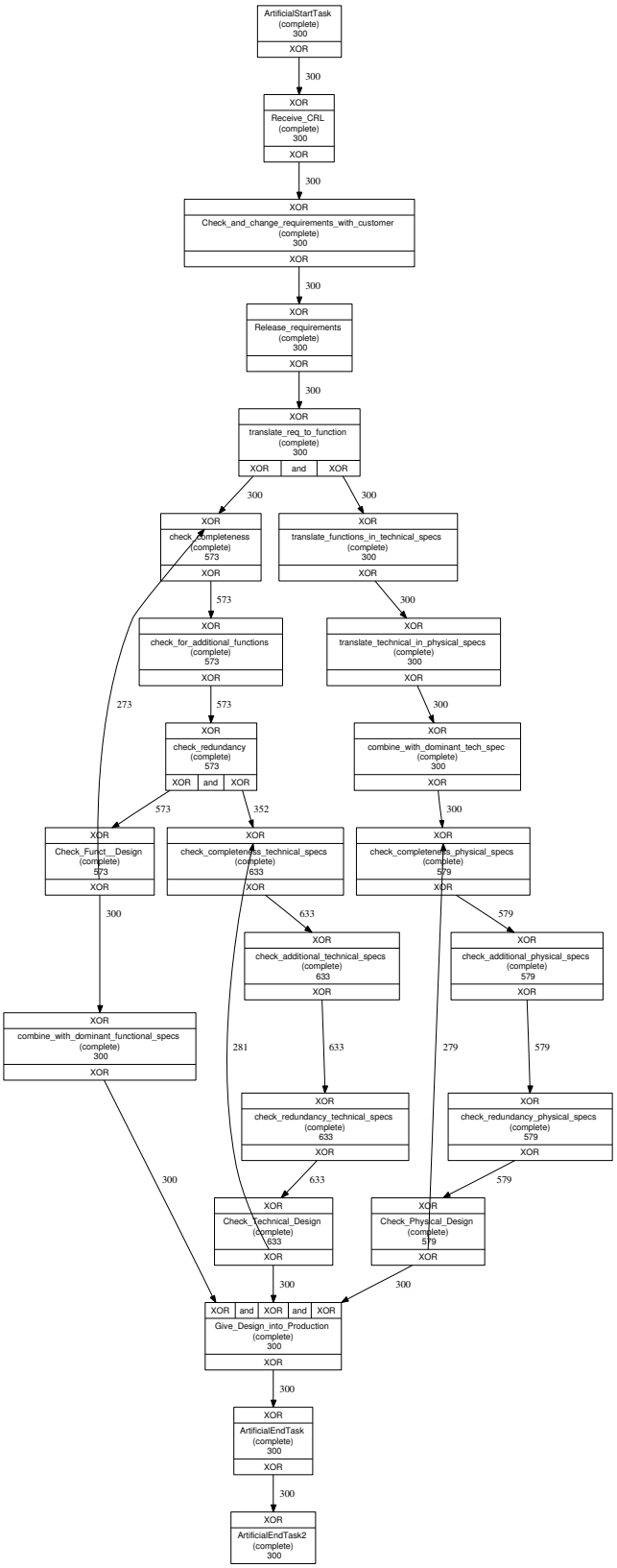


Figure C.19: Mined model for net **g14**.

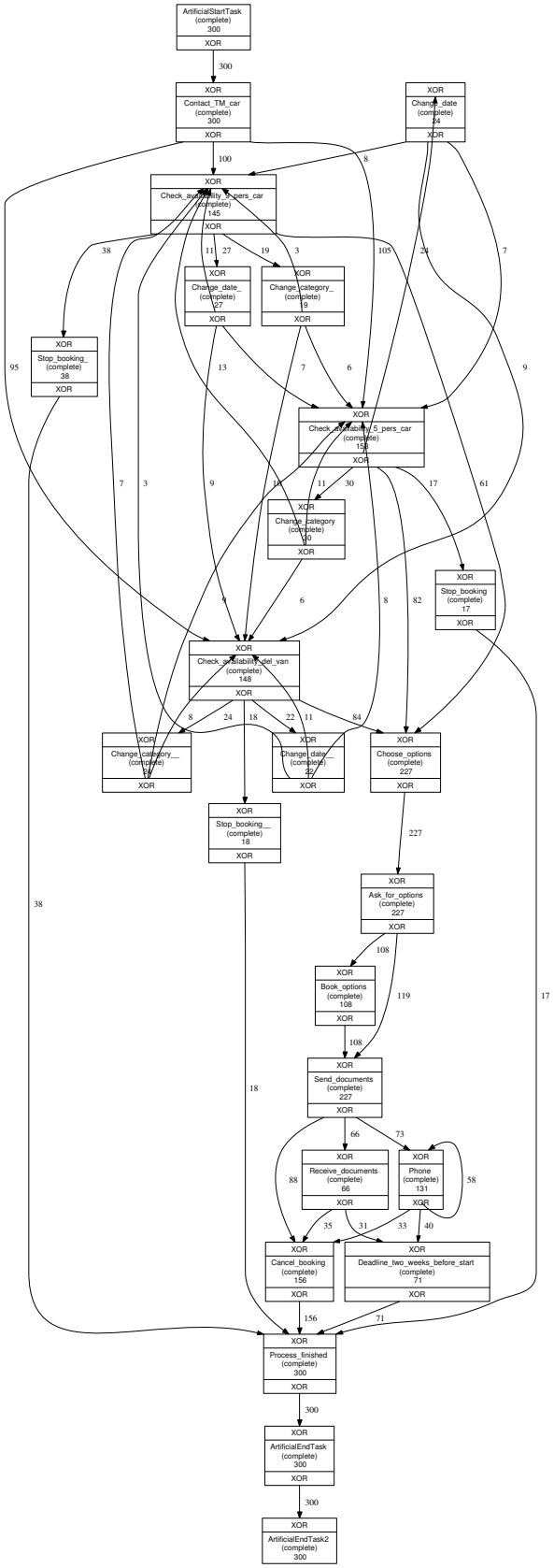


Figure C.20: Original model for net g15.

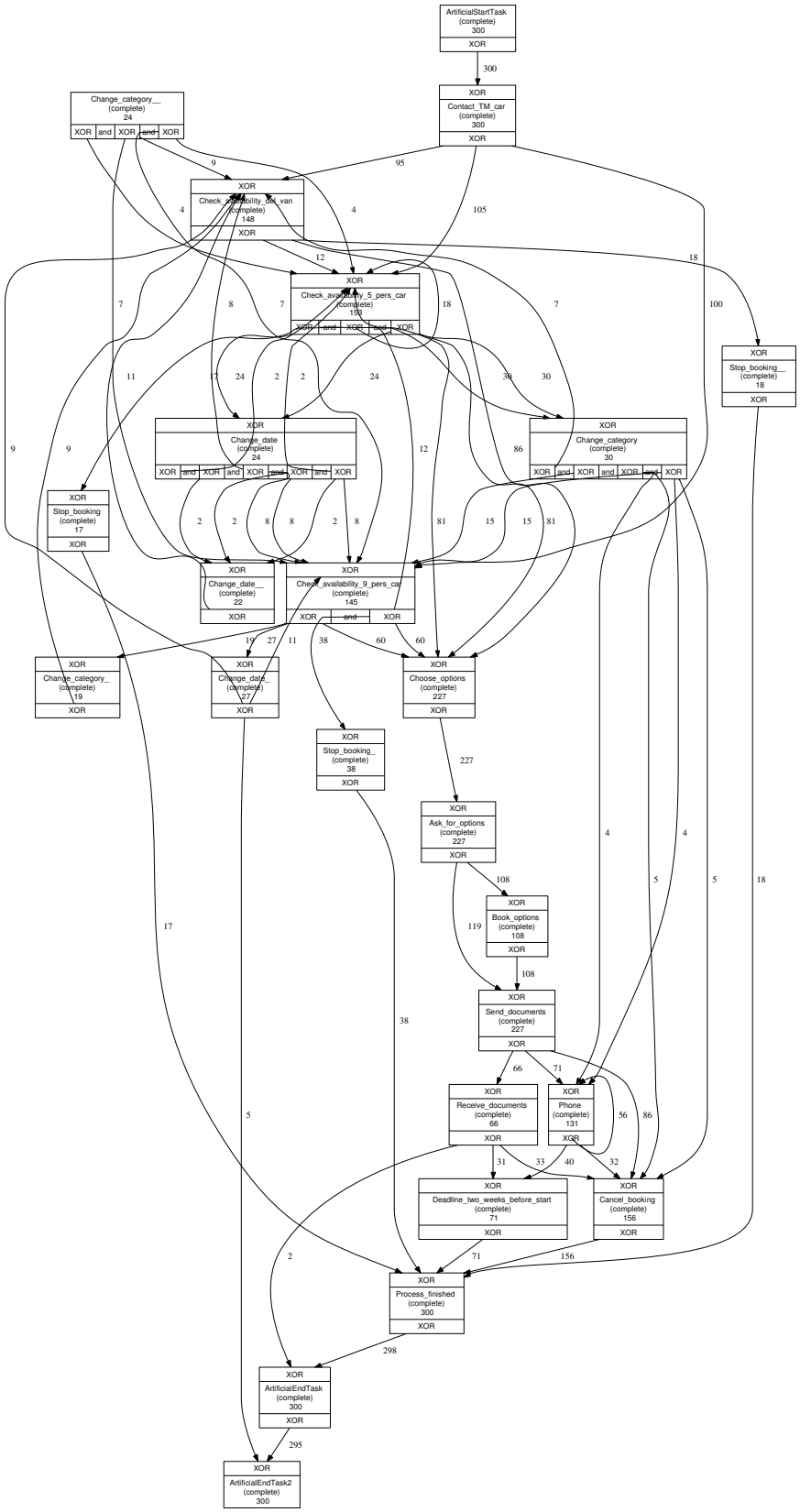


Figure C.21: Mined model for net g15.

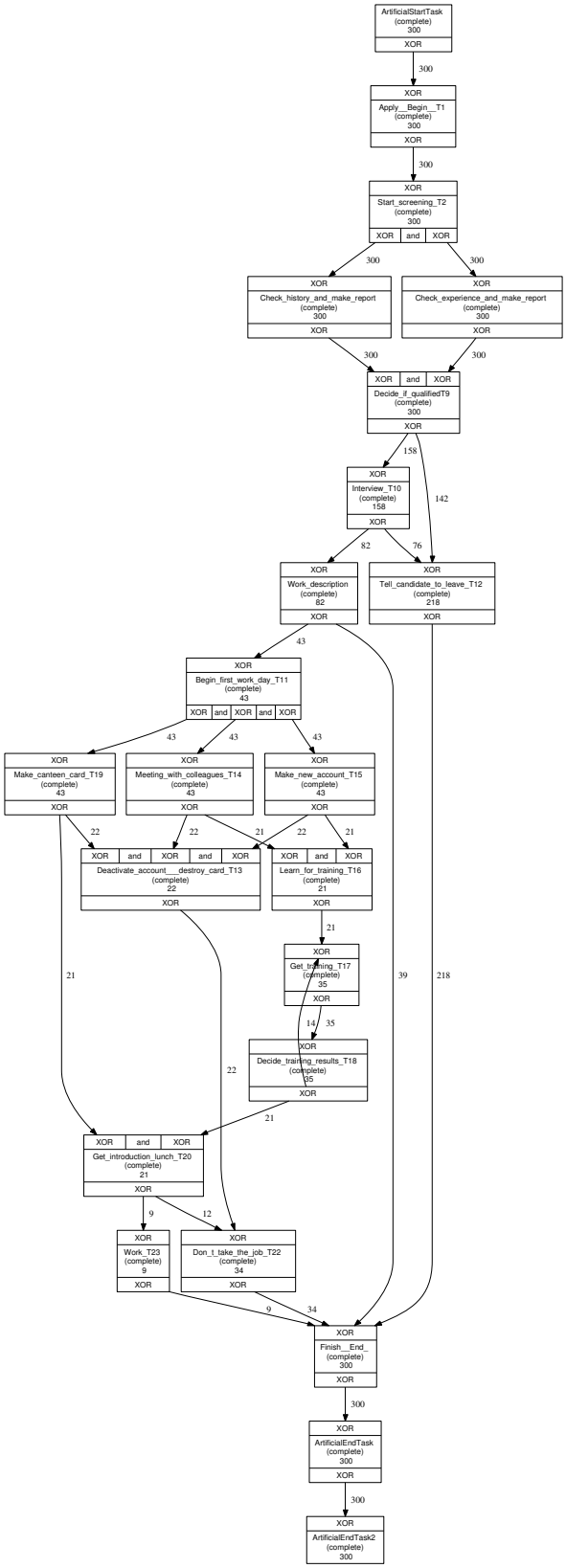


Figure C.22: Original model for net g19.

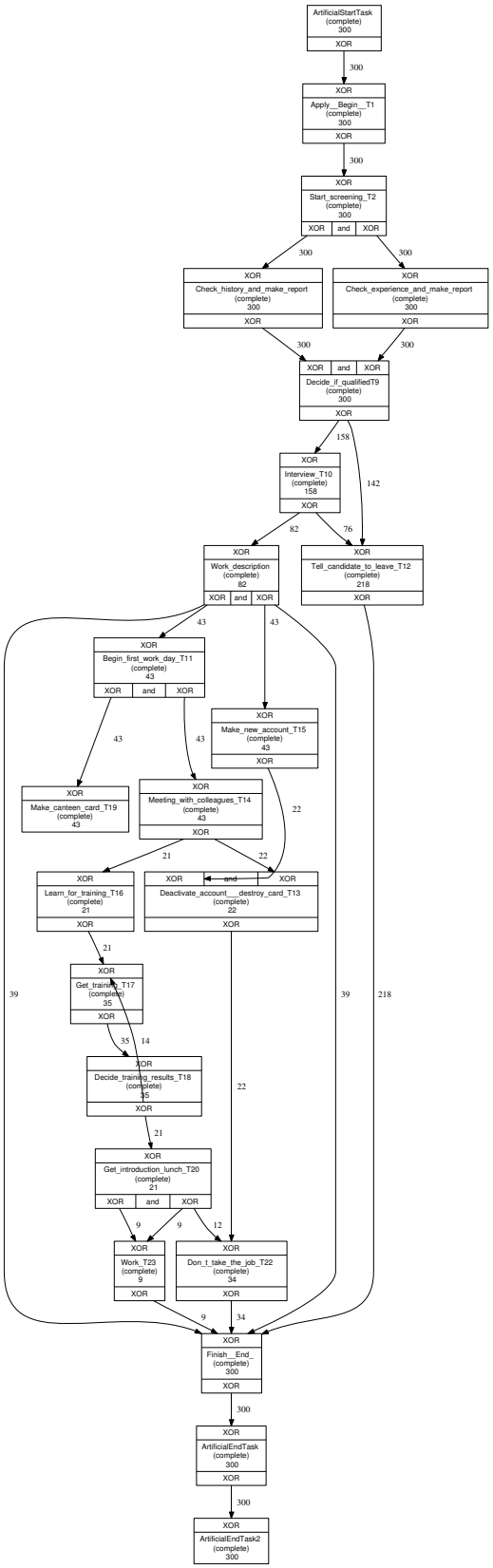


Figure C.23: Mined model for net g19.

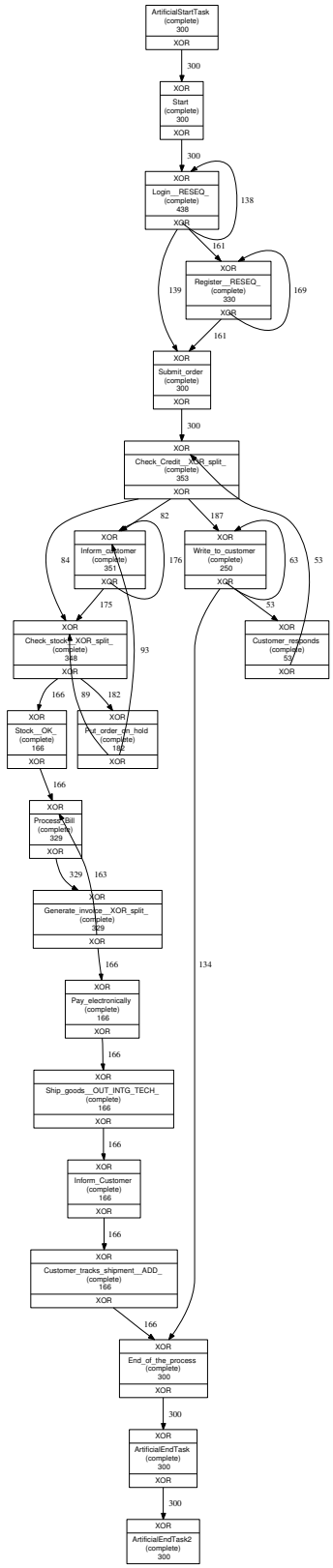


Figure C.24: Original model for net g20. The mined model is identical to the original model.

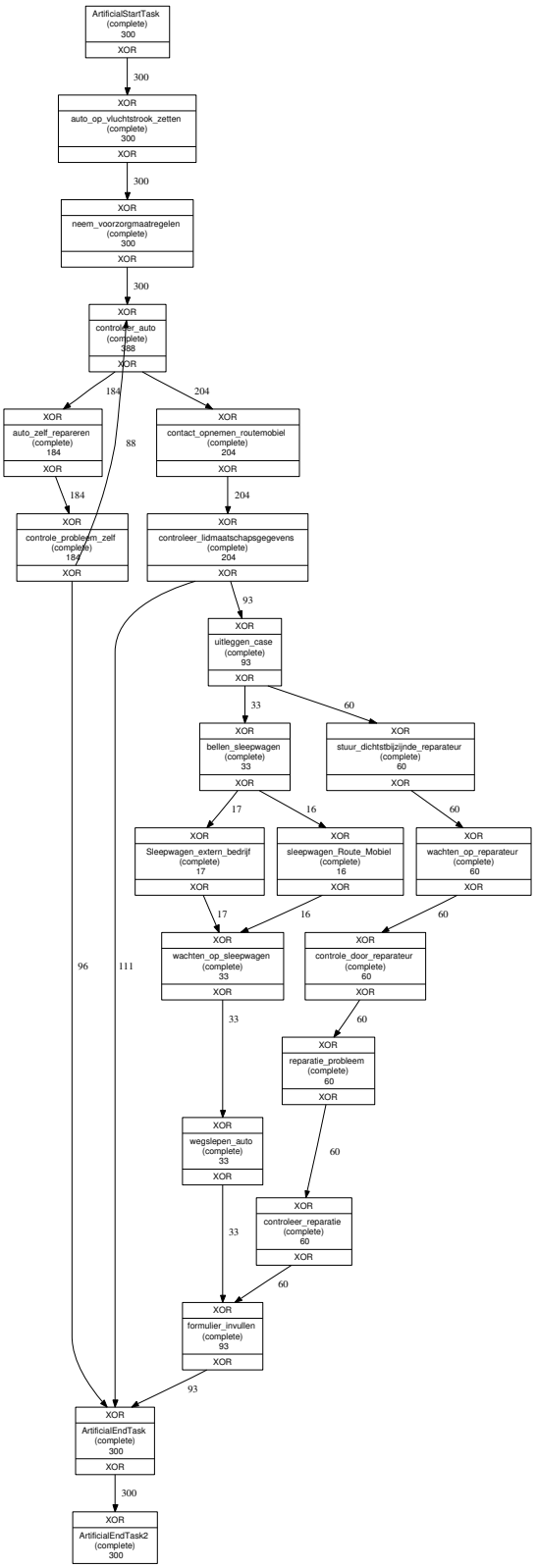


Figure C.25: Original model for net **g21**. The mined model is identical to the original model.

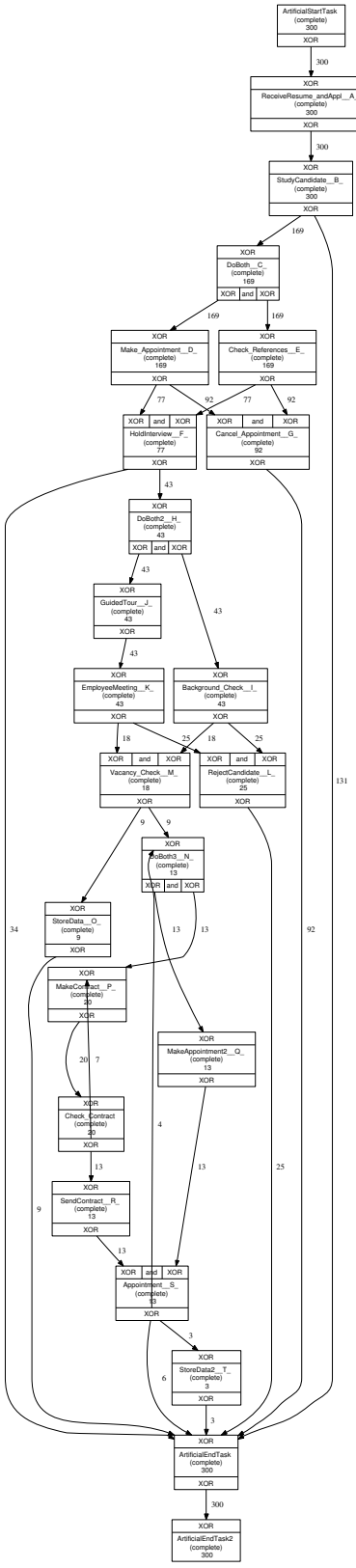


Figure C.26: Original model for net g22.

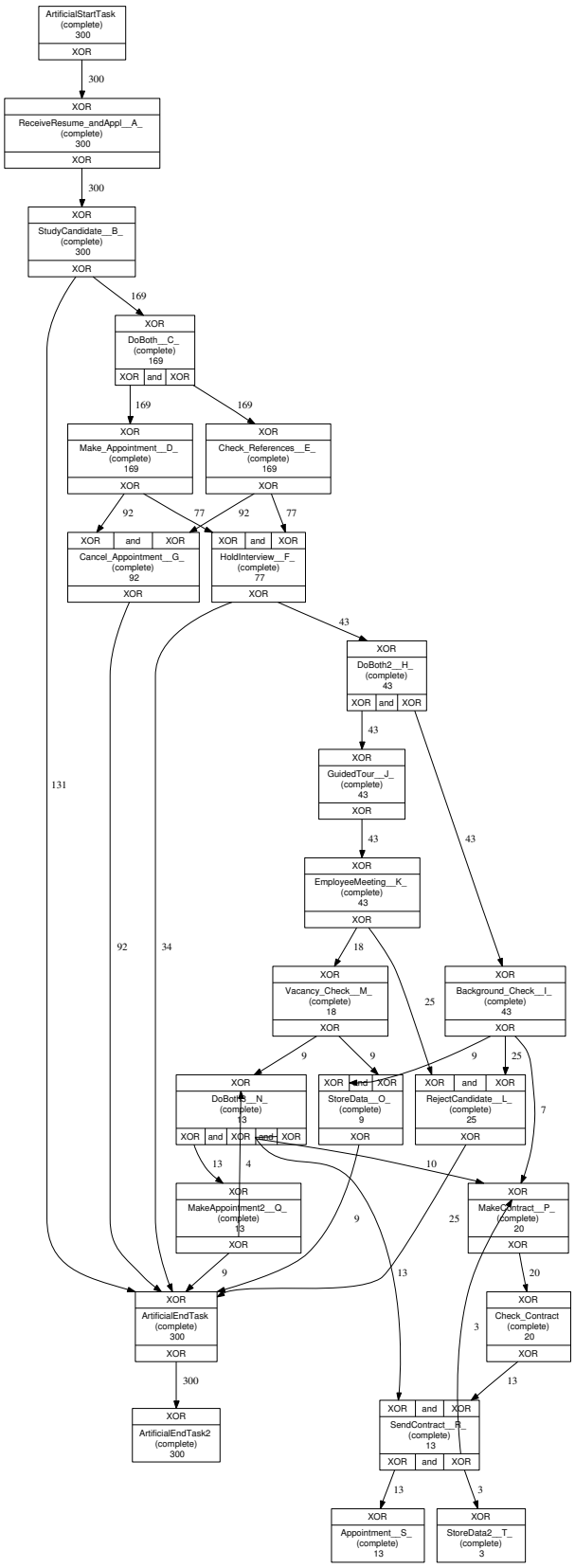


Figure C.27: Mined model for net g22.

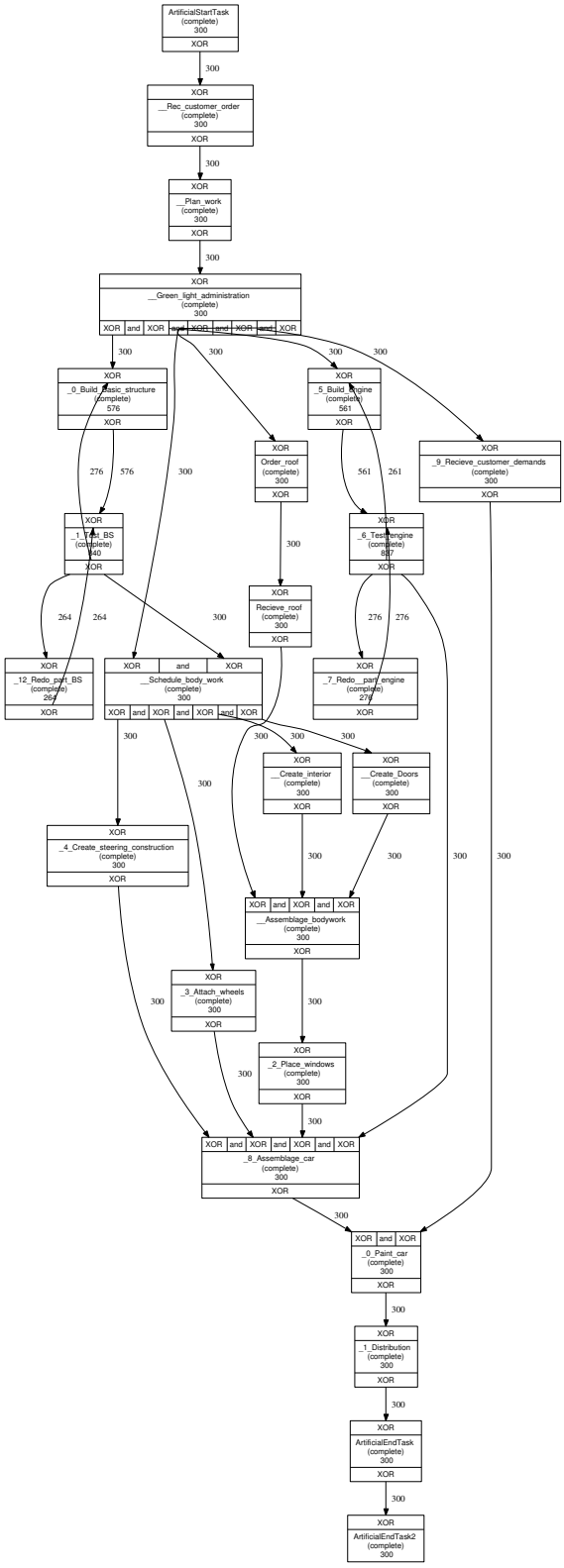


Figure C.28: Original model for net g23.

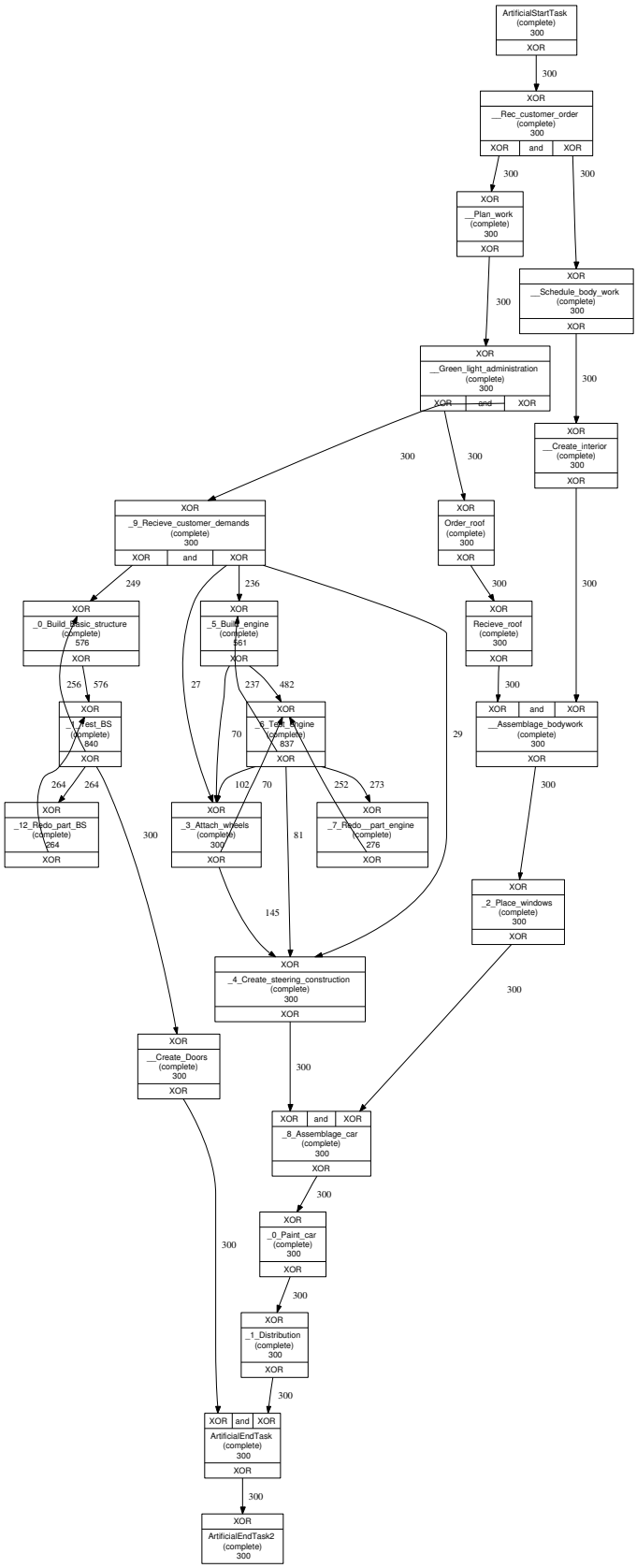


Figure C.29: Mined model for net g23.

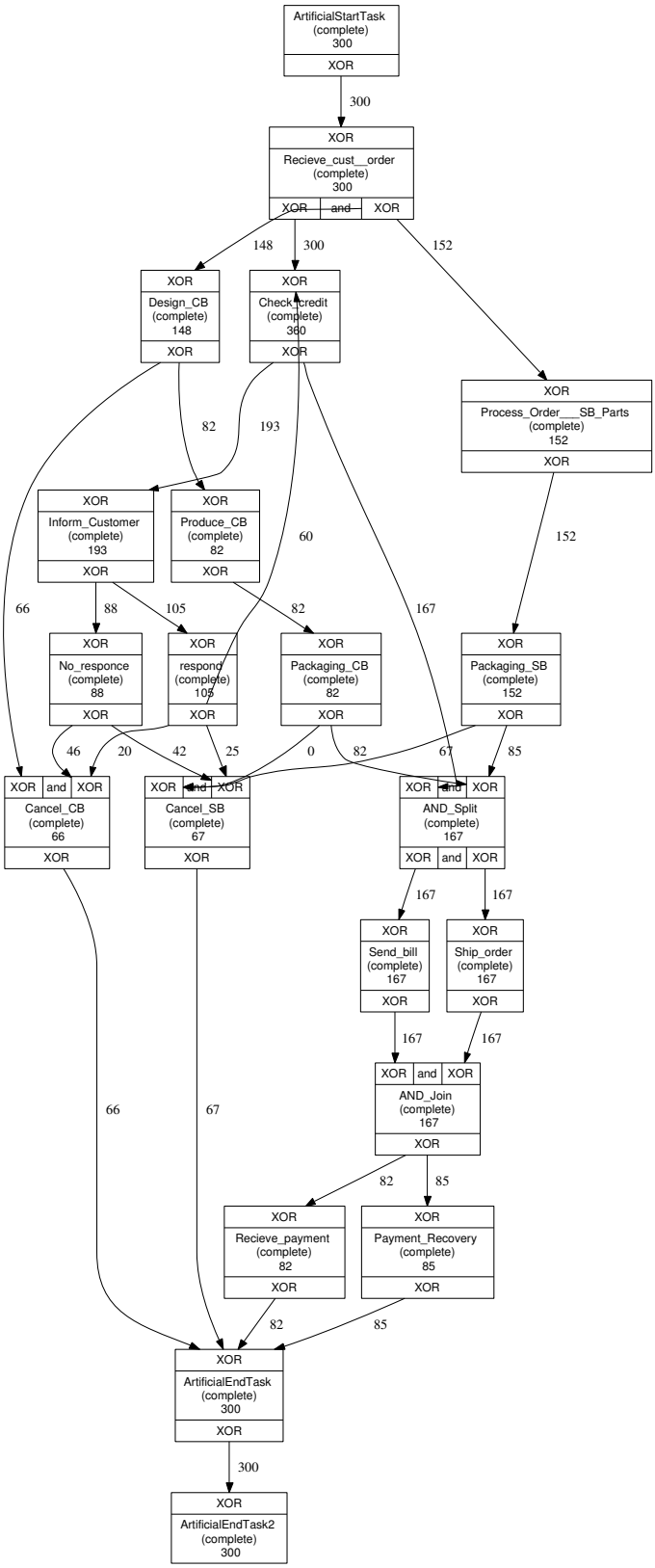


Figure C.30: Original model for net g24.

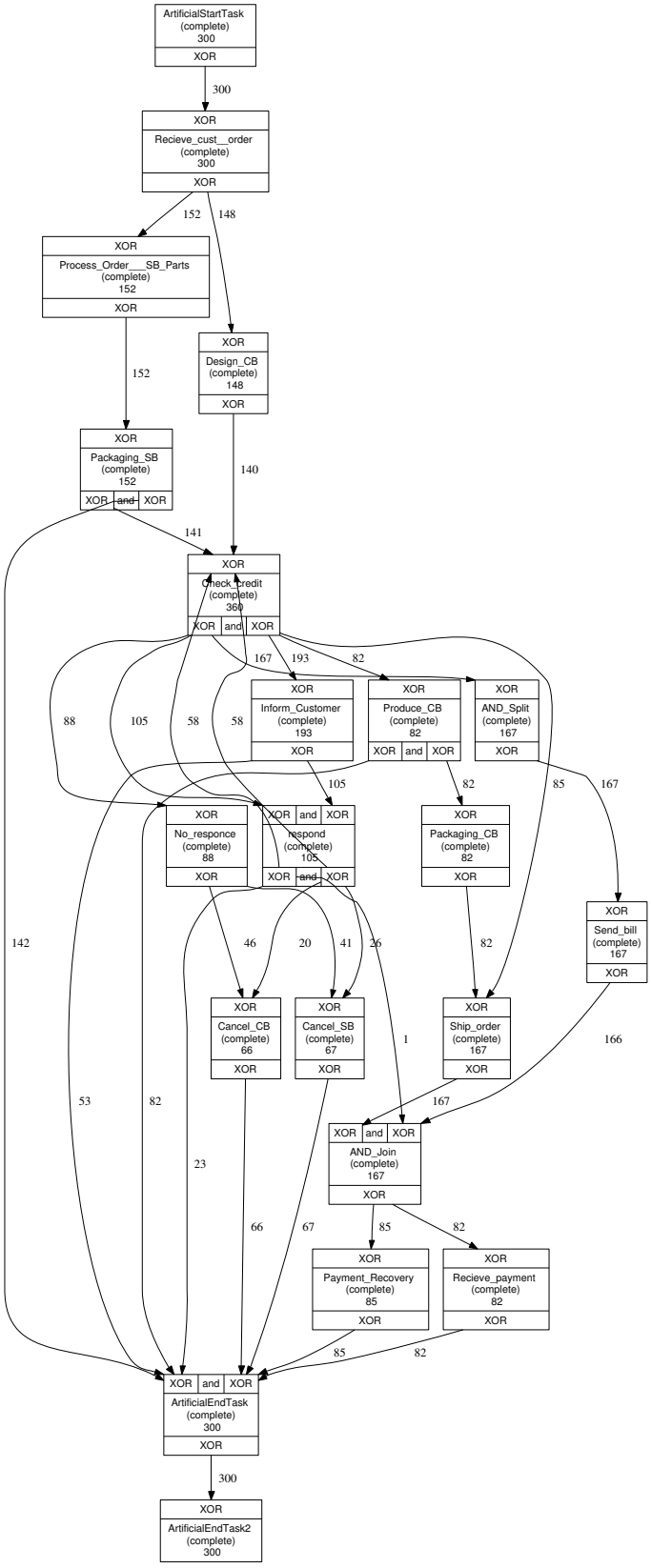


Figure C.31: Mined model for net g24.

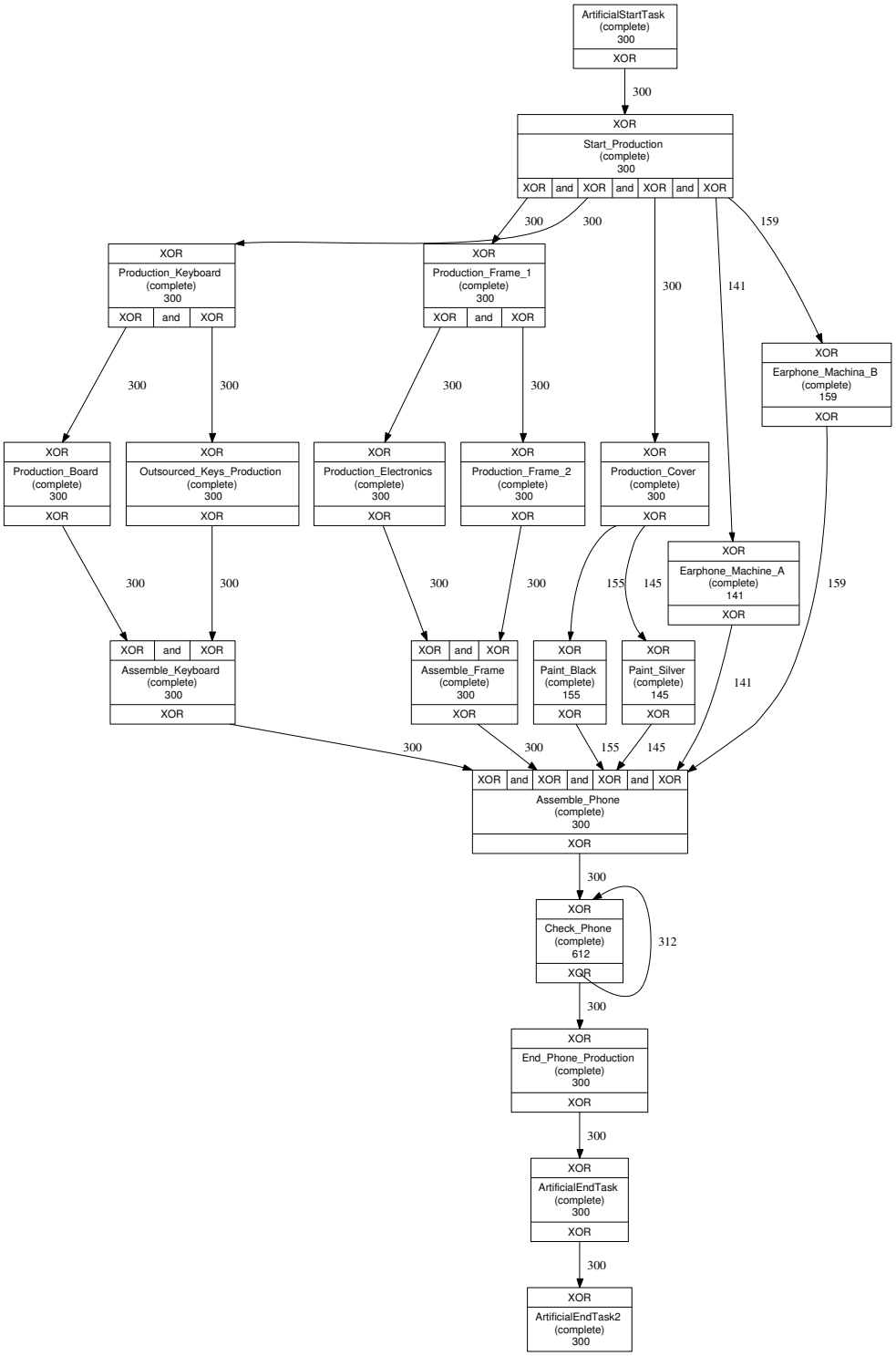


Figure C.32: Original model for net g25.

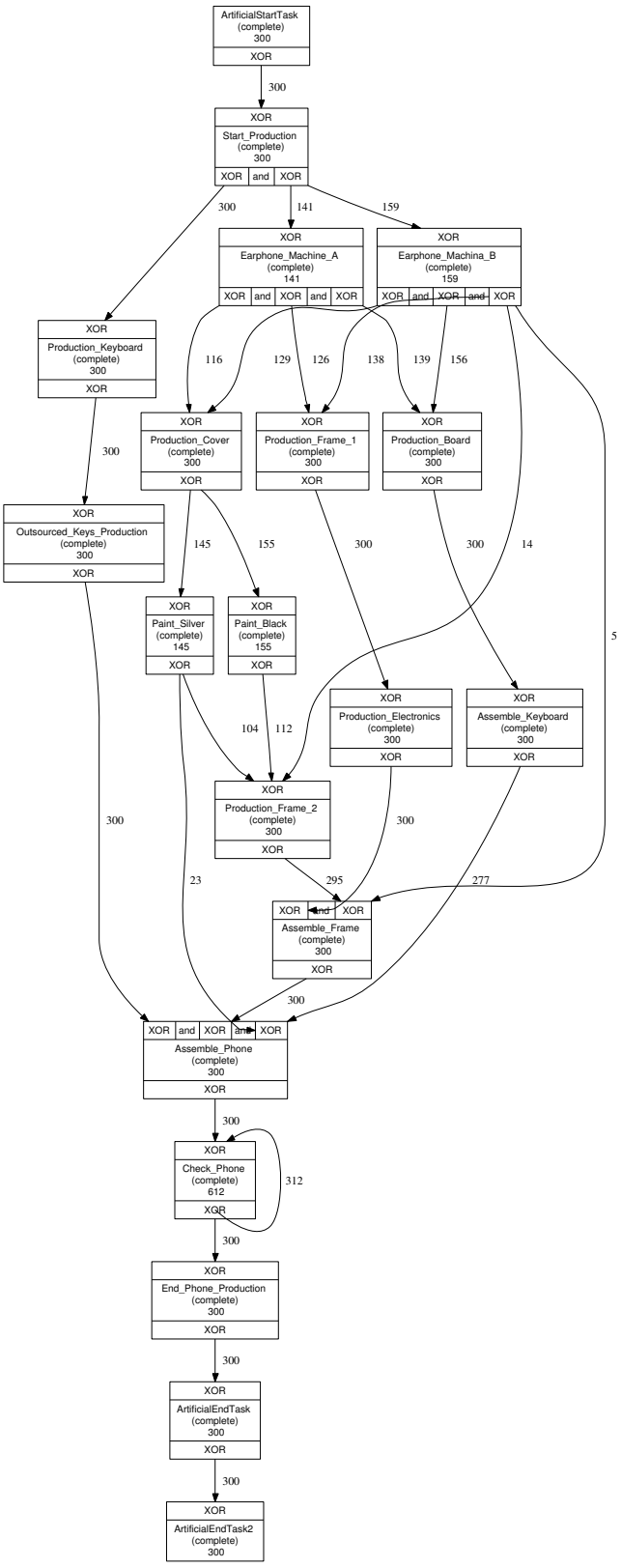


Figure C.33: Mined model for net g25.

Bibliography

[1] COSA Business Process Management. *http://www.cosa-bpm.com/*.

[2] CPN Tools. *http://wiki.daimi.au.dk/cpntools/*.

[3] Extensible Markup Language (XML). http://www.w3.org/XML/.

[4] Process Mining Website. *http://www.processmining.org/*.

[5] SAP. *http://www.sap.com/*.

[6] Staffware Process Suite. *http://www.staffware.com/*.

[7] W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, Berlin, 1997.

[8] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[9] W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.

[10] W.M.P. van der Aalst, H.T. de Beer, and B.F. van Dongen. Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In Robert Meersman, Zahir Tari, Mohand-Said Hacid, John Mylopoulos, Barbara Pernici, Özalp Babaoglu, Hans-Arno Jacobsen, Joseph P. Loyall, Michael Kifer, and Stefano Spaccapietra, editors, *OTM Conferences (1)*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2005.

[11] W.M.P. van der Aalst and B.F. van Dongen. Discovering Workflow Performance Models from Timed Logs. In Y. Han, S. Tai, and D. Wikarski, editors, *International Conference on Engineering and Deployment of*

Cooperative Information Systems (EDCIS 2002), volume 2480 of *Lecture Notes in Computer Science*, pages 45–63. Springer-Verlag, Berlin, 2002.

[12] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.

[13] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[14] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.

[15] W.M.P. van der Aalst and A.J.M.M. Weijters, editors. *Process Mining*, volume 53 of *Special Issue of Computers in Industry*. Elsevier Science Publishers, Amsterdam, 2004.

[16] W.M.P. van der Aalst and A.J.M.M. Weijters. Chapter 10: Process Mining. In M. Dumas, W.M.P. van der Aalst, and A.H. ter Hofstede, editors, *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons Inc, 2005.

[17] W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[18] R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In I. Ramos G. Alonso H.-J. Schek, F. Saltor, editor, *Advances in Database Technology - EDBT'98: Sixth International Conference on Extending Database Technology*, volume 1377 of *Lecture Notes in Computer Science*, pages 469–483, 1998.

[19] Pallas Athena. *Case Handling with FLOWer: Beyond workflow*. Pallas Athena BV, Apeldoorn, The Netherlands, 2001.

[20] Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2001.

[21] Workflow Management Coalition. WFMC Home Page. http://www.wfmc.org.

[22] J.E. Cook. *Process Discovery and Validation Through Event-Data Analysis*. PhD thesis, 1996.

[23] J.E. Cook, Z. Du, C. Liu, and A.L. Wolf. Discovering Models of Behavior for Concurrent Workflows. *Computers in Industry*, 53(3):297–319, 2004.

[24] J.E. Cook and A.L. Wolf. Automating Process Discovery Through Event-Data Analysis. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 73–82, New York, NY, USA, 1995. ACM Press.

[25] J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[26] J.E. Cook and A.L. Wolf. Event-Based Detection of Concurrency. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, pages 35–45, New York, NY, USA, 1998. ACM Press.

[27] A.K. Alves de Medeiros and C.W. Guenther. Process Mining: Using CPN Tools to Create Test Logs for Mining Algorithms. In K. Jensen, editor, *Proceedings of the Sixth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2005)*, volume 576 of *DAIMI*, pages 177–190, Aarhus, Denmark, October 2005. University of Aarhus.

[28] A.K. Alves de Medeiros, B.F. van Dongen, W.M.P. van der Aalst, and A.J.M.M. Weijters. Process Mining for Ubiquitous Mobile Systems: An Overview and a Concrete Algorithm. In L. Baresi, S. Dustdar, H. Gall, and M. Matera, editors, *Ubiquitous Mobile Information and Collaboration Systems (UMICS 2004)*, volume 3272 of *Lecture Notes in Computer Science*, pages 154–168. Springer-Verlag, Berlin, 2004.

[29] A.K. Alves de Medeiros, B.F. van Dongen, W.M.P. van der Aalst, and A.J.M.M. Weijters. Process Mining: Extending the α-algorithm to Mine Short Loops. BETA Working Paper Series, WP 113, Eindhoven University of Technology, Eindhoven, 2004.

[30] J. Dehnert and W.M.P. van der Aalst. Bridging the Gap Between Business Models and Workflow Specifications. *International Journal of Cooperative Information Systems*, 13(3):289–332, 2004.

[31] J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.

[32] B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM Framework: A New Era in Process Mining Tool Support. In G. C. and P. Darondeau, editors, *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2005.

[33] B.F. van Dongen and W.M.P. van der Aalst. EMiT: A Process Mining Tool. In Jordi Cortadella and Wolfgang Reisig, editors, *ICATPN*, volume

3099 of *Lecture Notes in Computer Science*, pages 454–463. Springer, 2004.

[34] B.F. van Dongen and W.M.P. van der Aalst. Multi-phase Process Mining: Building Instance Graphs. In Paolo Atzeni, Wesley W. Chu, Hongjun Lu, Shuigeng Zhou, and Tok Wang Ling, editors, *ER*, volume 3288 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2004.

[35] B.F. van Dongen and W.M.P. van der Aalst. Multi-phase Process mining: Aggregating Instance Graphs into EPCs and Petri Nets. In *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management (PNCWB)*, 2005.

[36] B.F. van Dongen, W.M.P. van der Aalst, and H.M.W. Verbeek. Verification of EPCs: Using Reduction Rules and Petri Nets. In O. Pastor and J. Falcão e Cunha, editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2005.

[37] M. Dumas, W.M.P. van der Aalst, and A.H. ter Hofstede, editors. *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons Inc, 2005.

[38] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Natural Computing. Springer-Verlag, Berlin, 2003.

[39] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, 17:37–54, 1996.

[40] L. Fischer, editor. *Workflow Handbook 2001, Workflow Management Coalition*. Future Strategies, Lighthouse Point, Florida, 2001.

[41] R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[42] M. Golani and S.S. Pinter. Generating a Process Model from a Process Audit Log. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 136 – 151, 2003.

[43] G. Greco, A. Guzzo, and L. Pontieri. Mining Hierarchies of Models: From Abstract Views to Concrete Specifications. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management*, volume 3649, pages 32–47, 2005.

[44] G. Greco, A. Guzzo, L. Pontieri, and D. Saccà. Mining Expressive Process Models by Clustering Workflow Traces. In H. Dai, R. Srikant, and

C. Zhang, editors, *PAKDD*, volume 3056 of *Lecture Notes in Computer Science*, pages 52–62. Springer, 2004.

[45] P. D. Grunwald, I. J. Myung, and M. Pitt, editors. *Advances in Minimum Description Length Theory and Applications*. The MIT Press, 2005.

[46] C.W. Guenther and W.M.P. van der Aalst. A Generic Import Framework for Process Event Logs. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, volume 4103, pages 81–92, 2006.

[47] M. Hammori, J. Herbst, and N. Kleiner. Interactive Workflow Mining. In B. Pernici J. Desel and M. Weske, editors, *Second International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 211 – 226. Springer-Verlag, Berlin, 2004.

[48] K. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In W.M.P. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 335–354. Springer-Verlag, Berlin, 2003.

[49] J. Herbst. A Machine Learning Approach to Workflow Management. In R.L. de Mntaras and E. Plaza, editors, *Proceedings 11th European Conference on Machine Learning*.

[50] J. Herbst. *Ein induktiver Ansatz zur Akquisition und Adaption von Workflow-Modellen*. PhD thesis, Universität Ulm, November 2001.

[51] J. Herbst and D. Karagiannis. Integrating Machine Learning and Workflow Management to Support Acquisition and Adaptation of Workflow Models. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 9:67–92, 2000.

[52] J. Herbst and D. Karagiannis. Workflow Mining with InWoLvE. *Computers in Industry*, 53(3):245–264, 2004.

[53] IBM. *IBM MQSeries Workflow - Getting Started With Buildtime*. IBM Deutschland Entwicklung GmbH, Boeblingen, Germany, 1999.

[54] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.

[55] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.

[56] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.

[57] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.

[58] L. Maruster. *A Machine Learning Approach to Understand Business Processes*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2003.

[59] J. Mendling and M. Nüttgens. Transformation of ARIS Markup Language to EPML. In *Proceedings of the 3rd GI Workshop on Business Process Management with Event-Driven Process Chains (EPK 2004)*, pages 27–38, 2004.

[60] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

[61] M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 1996.

[62] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[63] Pallas Athena. *Protos User Manual*. Pallas Athena BV, Plasmolen, The Netherlands, 1999.

[64] S.S. Pinter and M. Golani. Discovering Workflow Models from Activities Lifespans. *Computers in Industry*, 53(3):283–296, 2004.

[65] H.A. Reijers and S. Limam Mansar. Best Practices in Business Process Redesign: An Overview and Qualitative Evaluation of Successful Redesign Heuristics. *Omega: The International Journal of Management Science*, 33(4):283–306, 2005.

[66] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[67] A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Alignment Between Event Logs and Process Models. BETA Working Paper Series, WP 144, Eindhoven University of Technology, Eindhoven, 2005.

[68] A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In Christoph Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812, pages 163–176, 2005.

[69] G. Schimm. Process Mining. http://www.processmining.de/.

[70] G. Schimm. Generic Linear Business Process Modeling. In S.W. Liddle, H.C. Mayr, and B. Thalheim, editors, *Proceedings of the ER 2000 Workshop on Conceptual Approaches for E-Business and The World Wide Web and Conceptual Modeling*, volume 1921 of *Lecture Notes in Computer Science*, pages 31–39. Springer-Verlag, Berlin, 2000.

[71] G. Schimm. Process Miner - A Tool for Mining Process Schemes from Event-based Data. In S. Flesca and G. Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA)*, volume 2424 of *Lecture Notes in Computer Science*, pages 525–528. Springer-Verlag, Berlin, 2002.

[72] G. Schimm. Mining Most Specific Workflow Models from Event-Based Data. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 25 – 40, 2003.

[73] G. Schimm. Mining Exact Models of Concurrent Workflows. *Computers in Industry*, 53(3):265–281, 2004.

[74] Eastman Software. *RouteBuilder Tool User's Guide*. Eastman Software, Inc, Billerica, MA, USA, 1998.

[75] Staffware. *Staffware 2000 / GWD User Manual*. Staffware plc, Berkshire, United Kingdom, 2000.

[76] H.M.W. Verbeek. *Verification of WF-nets*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2004.

[77] H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.

[78] H.M.W. Verbeek, B.F. van Dongen, J. Mendling, and W.M.P. van der Aalst. Interoperability in the ProM Framework. In T. Latour and M. Petit, editors, *Proceedings of the CAiSE'06 Workshops and Doctoral Consortium*, pages 619–630, Luxembourg, June 2006. Presses Universitaires de Namur.

[79] A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

[80] L. Wen, J. Wang, W.M.P. van der Aalst, Z. Wang, and J. Sun. A Novel Approach for Process Mining Based on Event Types. BETA Working Paper Series, WP 118, Eindhoven University of Technology, Eindhoven, 2004.

[81] L. Wen, J. Wang, and J. Sun. Detecting Implicit Dependencies Between Tasks from Event Logs. In Xiaofang Zhou, Jianzhong Li, Heng Tao Shen, Masaru Kitsuregawa, and Yanchun Zhang, editors, *APWeb*, volume 3841 of *Lecture Notes in Computer Science*, pages 591–603. Springer, 2006.

Index

Summary

Nowadays, most organizations use information systems to support the execution of their business processes. These information systems may contain an explicit model of the business processes (e.g. workflow management systems), may support the tasks involved in the processes without necessarily defining explicit process models (e.g. ERP systems), or may simply keep track (for auditing purposes) of the tasks that have been performed without providing any support for the actual execution of those tasks (e.g. custom made systems in hospitals). Either way, these information systems typically support logging capabilities that register what has been executed in the organization: general data about cases (i.e. process instances), times at which tasks were executed, persons or systems that performed the tasks, and so on. Such logs, called *event logs*, are the starting point for process mining.

Process mining targets the *automatic* discovery of information from an event log. This discovered information can be used to deploy new systems that support the execution of business processes or as a feedback tool that helps in auditing, analyzing and improving already enacted business processes. The main benefit of process mining techniques is that information is *objectively* compiled. Depending on the type of data present in an event log, three different *perspectives* of process mining can be discovered. The *control-flow* perspective relates to the "How?" question (e.g. "How are the processes actually been executed?"), the *organizational* perspective to the "Who?" question (e.g. "Who is handing over work to whom?"), and the case perspective to the "What?" question (e.g. "What is the average throughput time for cases of a certain process?"). All these three perspectives are complementary and relevant for process mining. However, *this thesis focusses on the control-flow perspective of process mining.*

Control-flow mining techniques discover a process model that specifies the relations between tasks in an event log. This mined process model is an *objective picture* that depicts possible flows that were followed by the cases in the log (assuming that the events were correctly logged). Because the flow of tasks is to be portrayed, control-flow mining techniques need to support

the correct mining of the *common control-flow constructs* that appear in process models. These constructs are: *sequences*, *parallelism*, *choices*, *loops*, and *non-free-choice*, *invisible tasks* and *duplicate tasks*. In fact, there has been quite a lot of work on mining the control-flow perspective of process models. However, *none of the current control-flow process mining techniques is able to mine all constructs at once*. The *first reason* why these techniques have problems to handle all the constructs is that they are based on *local* information in the log. In other words, they use the information about what tasks directly precede or directly follow each other in a log to set the dependencies between these tasks. A *second reason* why some of the techniques cannot mine certain constructs is because the notation they use to model the processes does not support these constructs. Furthermore, many of the current techniques have problems while dealing with another factor that is common in real-life logs: *the presence of noise*. Noise is low frequent behavior that can appear in two situations: event traces were somehow incorrectly logged (for instance, due to temporary system misconfiguration) or event traces reflect exceptional situations. Either way, most of the techniques will try to find a process model that can parse all the traces in the log. The problem here is that, for many of these approaches, the number of times a relation holds in the log is irrelevant. Thus, these approaches are very vulnerable to noise because they are unable to distinguish between high frequent and low frequent behavior.

Given all these reasons, we decided to investigate if *it is possible to develop a control-flow process mining algorithm that can discover all the common control-flow structures while being robust to noisy logs*. We did so by applying genetic algorithms to perform process mining, and we call it *genetic process mining*. The choice for using genetic algorithms was mainly motivated by the absence of good heuristics that can tackle all the constructs, and by the fact that genetic algorithms are intrinsically robust to noise. This investigation resulted in two main contributions: (i) the genetic process mining algorithms themselves and (ii) the analysis metrics that quantify the quality of the mined models.

Genetic algorithms are a search technique that mimics the process of evolution in biological systems. In this thesis, two genetic algorithms have been defined: GA and DGA. The GA (basic Genetic Algorithm) can mine all structural constructs, except for duplicate tasks. The DGA (Duplicates Genetic Algorithm) is an extension of the GA that is able to also discover duplicate tasks. Both algorithms are robust to noise because they benefit the mining of models that correctly portrait the *most frequent behavior* in the log. Any genetic algorithm has three main building blocks: the internal representation of individuals, the fitness measure and the genetic operators.

For both the GA and the DGA, *individuals* are represented as *causal matrices*. The main strength of causal matrices is that they support the modelling of all common control-flow constructs in process models. The *fitness measure* assesses the quality of individuals by replaying the log traces into these individuals. The main lesson learned from this replaying process is that the use of a *continuous parsing semantics* is better than the use of a blocking semantics. The fitness measure guides the search towards individuals that are *complete*, *precise* and *folded*. An individual is complete when it can successfully replay all the traces in an event log. An individual is precise when it does not allow for much more behavior than the one that can be derived from the log. Therefore, the preciseness requirement is based on the amount of tasks of an individual that are simultaneously enabled while replaying the log. The purpose of this requirement is to punish individuals that tend to be over-general. An individual is folded when none of its duplicates have input/output elements in common. The folding requirement is based only on the structure of an individual. This requirement punishes the individuals that have too many duplicates and, therefore, tend to be over-specific. *Crossover* and *mutation* are the two genetic operators used in the GA and the DGA. The core concept behind both operators is that the *causality relations* (i.e. dependencies) among tasks are the genetic material to be manipulated.

In total, seven analysis metrics have been developed to quantify how complete, precise and folded the mined models are. These metrics are: the partial fitness for the completeness requirement ($PF_{complete}$), the behavioral precision (B_P) and recall (B_R), the structural precision (S_P) and recall (S_R), and the duplicates precision (D_P) and recall (D_R). From all these metrics, the more elaborate ones are the behavioral precision and recall, which quantify how much behavior two models have in common while parsing an event log. We had to develop these analysis metrics because two individuals could model the same behavior in the log, but have completely different structures. Furthermore, because it is unrealistic to assume that event logs are exhaustive (i.e. contain all the possible behavior that can be generated by original models), metrics that would compare the coverability graphs of individuals or metrics based on branching bisimilarity were not applicable anymore. Thus, the defined metrics can detect differences in the individuals, but can also quantify how much behavior they have in common regardless of the log being exhaustive or not. The concepts captured by these analysis metrics are applicable to situations in which two models need to be compared with respect to some exemplary behavior. In other words, these analysis metrics are useful also beyond the scope of our genetic process mining approach.

The results of our experiments and case study show that the mined models tend to be precise, complete and folded in many situations. However, the

algorithms have a drawback that cannot be neglected: the computational time. The situation is more critical for the DGA than for the GA because the DGA allows for duplicate tasks and, therefore, often uses a bigger search space. As a consequence, the DGA usually needs more iterations to converge to good solutions (i.e. mined process models that are complete, precise and folded).

Finally, all the algorithms described in this thesis have been implemented as plug-ins in the ProM framework. ProM is an open-source tool that is available at www.processmining.org.

Samenvatting

Tegenwoordig gebruiken de meeste organisaties informatiesystemen om de uitvoering van hun bedrijfsprocessen te ondersteunen. Deze informatiesystemen bevatten een expliciet model van de bedrijfsprocessen (bijv. workflow management systems), ondersteunen de taken in het proces zonder een expliciet model te definiëren (bijv. ERP systemen), of houden simpelweg bij welke taken er uitgevoerd zijn (voor auditing doeleinden) zonder ondersteuning te bieden bij het uitvoeren van deze taken (bijv. op maat gemaakte systemen in ziekenhuizen). Doorgaans bieden deze informatiesystemen de mogelijkheid om bij te houden wat er uitgevoerd is in een organisatie: algemene gegevens over cases ("process instances"), wanneer taken zijn uitgevoerd, personen die de taak hebben verricht, etc. Deze databestanden, genaamd *event logs*, zijn het startpunt voor process mining.

Het doel van process mining is het *automatisch* ontdekken van informatie in een event log. Deze informatie kan gebruikt worden om nieuwe systemen te implementeren die de uitvoering van bedrijfsprocessen ondersteunen of als een manier om feedback te krijgen over het auditen, analyseren en verbeteren van reeds geïmplementeerde bedrijfsprocessen. Het belangrijkste voordeel van process mining is dat de informatie op een *objectieve* wijze verkregen wordt. Afhankelijk van de gegevens in de event log kunnen drie verschillende *perspectieven* ontdekt worden. Het *control-flow-perspectief* gaat over de "Hoe?" vraag (bijv. "Hoe worden de processen echt uitgevoerd?"), het *organisatieperspectief* over de "Wie?" vraag (bijv. "Wie draagt werk over aan wie?") en het *caseperspectief* over de "What?" vraag (bijv. "Wat is de gemiddelde doorlooptijd van bepaalde cases?"). Deze drie perspectieven vullen elkaar aan en zijn alle relevant voor process mining. *In dit proefschrift ligt de nadruk echter op het control-flow-perspectief van process mining.*

Control-flow mining technieken ontdekken een procesmodel dat de onderlinge verbanden tussen taken in een event log weergeeft. Dit ontdekte procesmodel is een *objectieve weergave* van mogelijke opeenvolgende stappen in het proces die gevolgd zijn ("flow") door een bepaalde case in de event log (onder de aanname dat de event log correct is). Omdat de flow van

taken weergegeven dient te worden, moeten control-flow mining technieken de meestvoorkomende control-flow elementen kunnen herkennen. Deze elementen zijn: *opeenvolging*, *parallellisme*, *keuze*, *herhaling*, *non-free-choice*, *onzichtbare taken* en *gedupliceerde taken*. Hoewel er al behoorlijk wat werk is gedaan op het gebied van control-flow process mining, *kan geen enkele bestaande techniek al deze elementen tegelijkertijd herkennen*. De *eerste reden* waarom bestaande technieken hier problemen mee hebben is dat ze gebaseerd zijn op *lokale* informatie in de event log. Dat wil zeggen, ze gebruiken informatie over welke taken direct voorafgegaan of gevolgd worden door elkaar om de afhankelijkheden te ontdekken. Een *tweede reden* waarom sommige technieken sommige elementen niet kunnen herkennen is dat deze elementen niet uit te drukken zijn in de notatie die de technieken gebruiken. Bovendien hebben veel bestaande technieken problemen met een ander veelvoorkomende eigenschap van event logs: *ruis*. Ruis is laagfrequent gedrag dat in twee gevallen voor kan komen: data in de event log is fout opgeslagen (bijv. als het system tijdelijk niet goed geconfigureerd is), of er zijn flows die uitzonderlijke situaties beschrijven. De meeste technieken zullen een procesmodel proberen te vinden dat alle stappen in de event log kan naspelen. Het probleem is dat, voor veel van deze aanpakken, het aantal keer dat een bepaalde relatie tussen taken voorkomt in de log er niet toe doet. Dit maakt de huidige technieken erg gevoelig voor ruis omdat ze niet in staat zijn om onderscheid te maken tussen hoogfrequent en laagfrequent gedrag.

Gegeven deze redenen, hebben we besloten om te onderzoeken of *het mogelijk is om een control-flow mining algoritme te ontwikkelen dat alle veelvoorkomende control-flow elementen kan ontdekken en bovendien om kan gaan met ruis*. We hebben dit gedaan door genetische algoritmes te gebruiken voor process mining en we hebben het *genetic process mining* genoemd. De motivatie om genetische algoritmes te gebruiken was vooral de afwezigheid van goede heuristieken die alle elementen aan kunnen en het feit dat genetische algoritmes goed met ruis om kunnen gaan. Dit onderzoek heeft geleid tot twee belangrijke bijdragen: (i) de genetische process mining algoritmes zelf en (ii) de analysemetrieken die de kwaliteit van ontdekte procesmodellen kwantificeren.

Genetische algoritmes zijn een zoektechniek die het proces van evolutie in biologische systemen volgt. In dit proefschrift zijn twee genetische algoritmes gedefiniëerd: GA en DGA. GA (eenvoudig Genetic Algorithm) kan alle control-flow elementen herkennen, behalve gedupliceerde taken. DGA (Duplicates Genetic Algorithm) is een uitbreiding van GA die ook gedupliceerde taken kan ontdekken. Beide algoritmes kunnen goed omgaan met ruis omdat ze zoeken naar procesmodellen die het *meestvoorkomende gedrag* in de event log weergeven. Elk genetisch algoritme bestaat uit drie hoofdbestanddelen:

de interne representatie van de individuen, de fitness measure en de genetische operatoren. Voor zowel GA als DGA worden *individuen* gerepresenteerd als *causale matrices*. Het belangrijkste voordeel van causale matrices is dat ze alle control-flow elementen ondersteunen. De *fitness measure* beoordeelt de kwaliteit van individuen door de event log na te spelen aan de hand van het procesmodel wat gedefiniëerd wordt door deze individuen. De belangrijkste les met betrekking tot deze simulatie is dat zogenaamde *continuous parsing semantics* beter zijn dan blocking semantics. De fitness measure legt de nadruk op individuen die *compleet*, *precies* en *folded* zijn. Een individu is compleet wanneer het de event log helemaal kan naspelen. Een individu is precies wanneer het niet veel meer gedrag toestaat dan wat er uit de event log afgeleid kan worden. Daarom is de precisie-eis gebaseerd op het aantal taken dat tegelijkertijd mogelijk uitgevoerd zouden kunnen worden tijdens het naspelen van de event log met een individu. Het doel van deze eis is om individuen te straffen die te algemeen zijn. Een individu is folded wanneer geen enkele van zijn gedupliceerde taken input/output elementen gemeen hebben. Deze eis straft individuen die te veel gedupliceerde taken hebben, en daardoor te specifiek zijn. *Crossover* en *mutatie* zijn de twee genetische operatoren die gebruikt zijn in GA en DGA. Het basisconcept achter beide operatoren is dat de *causale relaties* (de afhankelijkheden) tussen taken het genetisch materiaal zijn dat gemanipuleerd dient te worden.

In totaal zijn er zeven analysemetrieken ontwikkeld om te kwantificeren hoe compleet, precies en folded de ontdekte modellen zijn. Deze metrieken zijn: de partiële fitness voor de compleetheidseis ($PF_{complete}$), de precisie van het gedragsaspect (B_P) en de recall ervan (B_R), de precisie van de structuur (S_P) en de recall ervan (S_R), en de precisie van de gedupliceerde taken (D_P) en de recall ervan (D_R). De meest uitgebreide metrieken zijn de precisie en recall van het gedragsaspect; zij kwantificeren hoeveel gedrag twee modellen gemeen hebben wanneer een event log nagespeeld wordt. We hebben deze analysemetrieken moeten ontwikkelen omdat two individuen hetzelfde gedrag kunnen modelleren, maar tegelijkertijd een compleet verschillende structuur kunnen hebben. Bovendien, omdat het niet realistisch is om aan te nemen dat event logs compleet zijn (d.w.z. alle mogelijke gedrag bevatten van de oorspronkelijke procesmodellen), waren metrieken die de coverability-graaf van individuen vergelijken of metrieken op basis van branching bisimilarity niet meer van toepassing. De gedefinierde metrieken kunnen dus verschillen detecteren tussen individuen, maar zij kunnen ook kwantificeren hoeveel gedrag ze gemeen hebben ongeacht of de event log compleet is of niet. De concepten achter deze analysemetrieken zijn van toepassing op situaties waar twee modellen vergeleken moeten worden ten opzichte van een bepaald “goed” gedrag. Met andere woorden, deze metrieken zijn ook nuttig buiten het gebied van

genetische process mining.

De resultaten van onze experimenten en case study laten zien dat de ontdekte modellen vaak precies, compleet en folded zijn. Echter, de algoritmes hebben een nadeel dat niet verwaarloosd kan worden: rekentijd. De situatie is kritischer voor DGA dan voor GA omdat DGA gedupliceerde taken toestaat en daarom vaak een grotere zoekruimte heeft. Het gevolg hiervan is dat DGA meestal meer iteraties nodig heeft om te convergeren naar goede oplossingen (d.w.z. procesmodellen die compleet, precies en folded zijn).

Tot slot, alle algoritmes beschreven in dit proefschrift zijn geïmplementeerd als plug-ins in het ProM-framework. ProM is een open-source programma en is beschikbaar op www.processmining.org.

Acknowledgements

Although the tangible and final product of a PhD is a thesis, there is a lot more to it than the written document. Actually, dedicating four years of your life to a project is much more worthwhile if you have interesting and nice people around you. When I decided to come to The Netherlands to do my PhD, I was also motivated by the fact that this would be a nice opportunity to be in touch with a different culture. I believe experiences abroad help one in breaking old paradigms. Actually, my adaptation here was very smooth and pleasant. In part because, in my opinion, the Dutch culture is not that different from the Brazilian one, but also because I was lucky enough to come to a nice work environment and to make some very nice friends during these years. I try to acknowledge all of them here, but I apologize in advance if I forgot someone.

(Still in Brazil)

The first person I want to thank is Jorge C.A. de Figueiredo because he was the one who told me about the PhD position in Eindhoven. Without his hint, I would have never applied for the position. Thanks, Jorge! Two other important people were Péricles and Heliante Barros. They have both lived in Eindhoven before (Heliante is Dutch) and they provided me with the most important facts that Brazilians should know about Dutch people. Furthermore, they introduced me (via e-mail) to a family (de Paiva Verheijden) that would play a very important role in my stay in Eindhoven. Thanks, Péricles and Heliante!

(While in The Netherlands)

My stay in The Netherlands has been a great opportunity for both my professional and personal development. First of all, I was very lucky to have Wil van der Aalst and Ton Weijters as my advisors. Their passion for good research and their different (but complementary!) leadership styles inspired me to always try my best. Wil and Ton, thank you very much for believing in me and providing such a rewarding work environment! I have learned a lot in these four years! Second, I would like to thank my colleagues and friends from the Information Systems (IS) group for the nice discussions and

social activities. More specifically, I would like to thank: Ineke Withagen and Ada Rijnberg for providing their services as secretaries and translating some Dutch documents for me in the first years in Eindhoven; Boudewijn van Dongen, my officemate, for the discussions about how to express my ideas in a nice mathematical way; Laura Maruster (former PhD in our group) for having helped me in getting acquainted with life in Eindhoven and for becoming such a great friend; Nick Szirbik for helping me in registering for a course to learn Dutch at ROC; Monique Jansen-Vullers for introducing me to her Brazilian neighbor already in the first months after my arrival in Eindhoven and for helping me in getting the data for the case study conducted in this thesis; Eric Verbeek for his technical support and also for allowing me to run my experiments in his computer as well; Maurice Loosschilder for generating the logs for the blind experiments; Alex Norta, Anke Hutzschenreuter, Anne Rozinat, Christian Günther, Florian Gottschalk, Irene Vanderfeesten, Jeroen van Luin, Hajo Reijers, Maja Pesic, Mariska Netjes, Minseok Song, Nataliya Mulyar, Samuil Angelov, Sven Till and Ting Wang for the nice times while playing sports, chatting, having lunch and/or going out together.

Other people who were not working in the IS group but also played an important role in these years are: Peter van den Brand (thanks for being such a special partner!) and his family (thanks for accepting me even when I could not speak Dutch yet!); the employees of the Dutch municipality (thanks for collaborating with us during the case study!); Cristina Ivanescu (thanks for introducing me to Peter and for being a great friend!); Ulaz Ozen and Baris Selcuk (thanks for helping with the Queuing Theory course!); Onno ter Haar and Geri van den Berg (thanks for being so motivating while teaching Dutch!); Family de Paiva Verheijden, especially Ignez and Jan (thank you very much for welcoming me in Eindhoven and having acted as my "second family" in many moments here!); Karina and Jack Nijssen (thanks for the nice walks, celebrations and cheerful moments!); Tânia Fernandez (thanks for the nice chats and for introducing Karina and Jack to me!); Adriana Stout (thanks for putting me in touch with many of the Brazilians I know nowadays in Eindhoven!); Nancy Horning (thanks for the nice times we had together!); Ana Rita and Martin Roders, Fabio Ferreira, Javier Aprea, Keila and Hermes Cordoba, Mônica Melhado, Sara Barros, Viviane Soares and Cícero Vaucher (thank you all for the nice moments in Eindhoven!); Franklin Ramalho, Genaína Rodrigues, Gilberto Cysneiros, Juliano Iyoda, Lívia Sampaio and Natasha Lino (my Brazilian friends also doing (part of) a PhD abroad, thanks for the nice trips in Europe and for your friendship!); and my other friends in Brazil, especially Andréa Lima and Andréa Cavalcanti (thanks for the encouraging e-mails and phone calls!).

(Always)

Last, but definitely not least, I want to thank my family for all their constant support, love, care and stimulus. Thanks, mom and dad! Thanks, Karina and Leonardo!

Curriculum Vitae

Ana Karla Alves de Medeiros was born on June 18, 1976 in Campina Grande, Paraíba, Brazil, where she also grew up. From 1991 to 1993 she went to high school at the Colégio Imaculada Conceição (Damas) in Campina Grande.

After finishing high school, she studied from 1994 to 1997 at the Universidade Federal da Paraíba (Campus II)[1], in Campina Grande, Brazil, to get a Bachelor of Science degree in Computing Science (Ciência da Computação). From 1998 to 2000, Ana Karla studied to get a Master of Science degree in Computing Science, at this same university. The title of her dissertation is "Mecanismos de Interação para um Modelo de Redes de Petri Orientado a Objetos" (Interaction Mechanisms for an Object-Oriented Petri Net Model).

From May/2000 until July/2002, Ana Karla worked as a software engineer for a software company in Recife, Pernambuco, Brazil. The company was called Radix and its main product was a search engine especially tailored for the Brazilian web.

In September 2002, Ana Karla moved to Eindhoven, The Netherlands, to start her doctoral studies under the supervision of Prof.dr.ir. W.M.P. van der Aalst and the co-supervision of Dr. A.J.M.M. Weijters in the area of process mining, at the Technische Universiteit Eindhoven. Her doctoral studies were completed in November 2006. The title of her doctoral thesis is "Genetic Process Mining".

At the moment (2006), Ana Karla continues to live in Eindhoven and since April she is working as a researcher for the SUPER European project (www.ip-super.org). Her main research area is process mining, but she is also interested in semantic web, data mining, business process management, genetic algorithms and Petri nets. Ana Karla can be reached at a.k.medeiros@tm.tue.nl.

[1] Nowadays, this university is called Universidade Federal de Campina Grande.