Universidade do Minho
Escola de Engenharia

Diogo Manuel Rodrigues Santos Silva

ETL Systems Modelling with Coloured Petri Nets

Janeiro de 2013

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo Manuel Rodrigues Santos Silva

ETL Systems Modelling with Coloured Petri Nets

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor Doutor Orlando Manuel de Oliveira Belo
Professor Doutor João Miguel Fernandes

Janeiro de 2013

Modelling ETL Systems with Coloured Petri Nets

Diogo Manuel Rodrigues Santos Silva

Dissertação apresentada à Universidade do Minho para obtenção do grau de Mestre em Engenharia Informática, elaborada sob orientação do Professor Doutor Orlando Manuel de Oliveira Belo e do Professor Doutor João Miguel Fernandes.

2013

Agradecimentos

A todas as pessoas que, de alguma maneira, me influenciaram positivamente durante o período de criação desta dissertação, tanto a nível pessoal como académico. Aos meus orientadores e professores, Orlando Belo e João Miguel Fernandes. A António Santos Silva, Helena Rodrigues, Daniel Silva, Mónica Casado e a todos os amigos e colegas que me acompanharam no percurso académico que agora termina.

List of Acronyms

CDC – Change Data Capture

CPN – Coloured Petri Net(s)

DBMS – Data Base Management System

DML – Data Manipulation Language

DSA – Data Staging Area

DW – Data Warehouse

DWS – Data Warehousing Systems

ETL – Extract - Transform - Load

GUI – Graphical User Interface

LSN – Log Sequence Number

PN – Petri Nets(s)

SCD – Slowly Changing Dimension(s)

SCD-H – Slowly Changing Dimension(s) with maintenance of Historic records

SK – Surrogate Key

Resumo

Modelação de sistemas de ETL com Redes de Petri Coloridas

Os sistemas de ETL (Extract-Transform-Load) são formados por processos responsáveis pela extração de dados de diversas fontes, pela sua limpeza e transformação de acordo com os pré-requisitos de um data warehouse, e finalmente pelo seu carregamento em estruturas multidimensionais. Os processos de ETL são as tarefas mais complexas no desenvolvimento de um sistema de data warehousing, sendo essencial modelar tais tarefas para que, durante a fase de implementação, sejam considerados os requisitos certos do sistema. As Redes de Petri Coloridas são uma linguagem de modelação gráfica usada no desenho, especificação, simulação e validação de sistemas concorrentes.

Nesta dissertação é apresentado o estudo relativo à aplicação das Redes de Petri Coloridas (RPC) no desenho conceptual e validação de sistemas de ETL (Extract-Transform-Load). Para o iniciar foi feita uma pesquisa das abordagens já existentes, no que diz respeito à modelação conceptual deste tipo de sistemas, para determinar se o seu desenho conceptual é normalmente efetuado durante a implementação de Sistemas de Data Warehouse e também para determinar quais as linguagens de modelação adotadas neste tipo de tarefa. Esta pesquisa confirmou que, para além de não ser usual modelar conceptualmente sistemas de ETL antes da sua implementação, as RPC nunca foram usadas para tal.

Para usar as RPC na modelação de sistemas de ETL foi feito um estudo mais aprofundado dos conceitos desta linguagem; ao mesmo tempo foram selecionados para uma primeira abordagem dois processos de ETL que, embora simples, são determinantes neste tipo de sistemas: Surrogate Key Pipeline (SKP) e Surrogate Key Generation. Antes destes processos serem

modelados foi efetuado um estudo teórico relativo ao comportamento e estrutura de cada um, de acordo com a metodologia proposta por Ralph Kimball (Kimball and Caserta, 2004). Os primeiros modelos implementados são bastante acessíveis pois, para além de os processos correspondentes serem relativamente simples, não necessitam de utilizar conceitos hierárquicos. Depois de testada a capacidade das RPC no desenho e validação de tarefas mais simples, foi selecionado um processo padrão nos sistemas de ETL – Slowly Changing Dimensions (SCD) – para dar continuidade a este estudo, pois é uma tarefa mais complexa composta por operações mais pequenas (ou subprocessos) e é por isso adequada para a introdução de conceitos hierárquicos.

O último processo de ETL a ser considerado neste estudo é o Change Data Capture (CDC). Existem variadas maneiras de implementar este processo; os métodos mais avançados são automáticos e dependem do Sistema de Gestão de Base de Dados (SGBD) usado nas fontes operacionais. Para implementar o modelo correspondente foi selecionado um SGBD e o seu método de CDC analisado para que este fosse modelado de acordo com o seu comportamento. Depois dos três módulos estarem completos, cada um deles representando uma processo padrão de ETL, é possível modelar sistemas de ETL que sejam formados por estas operações. Para terminar este trabalho de dissertação foi implementado um modelo, com RPC, para um sistema de ETL baseado num data mart dado como exemplo, com o objetivo de demonstrar a aplicação prática dos módulos implementados no desenho conceptual de ETL.

Abstract

Modelling ETL Systems with Coloured Petri Nets

ETL (Extract-Transform-Load) systems are formed by processes responsible for the extraction of data from several sources, cleaning and transforming it in accordance with some prerequisites of a data warehouse, and finally loading it in its multidimensional structures. ETL processes are the most complex tasks involved with a Data Warehousing System, being crucial to model them previously so that, during the implementation stage, the correct set of requirements is considered. Coloured Petri Nets (CPN) are a graphical modelling language used in the design, specification, simulation and validation of large systems, characterized as being strongly concurrent.

In this dissertation the carried out study concerning the application of CPN in the conceptual design and validation of ETL systems is presented. Initially, a brief research on the existing approaches for the conceptual modelling of ETL systems was made, in order to determine if the conceptual design is usually adopted during the implementation of DWS (Data Warehousing Systems) and also to find out which modelling languages are mostly used in this kind of task. As it was confirmed by this initial research, the conceptual modelling of ETL is not common and the CPN is not one of the languages that have already been used to design and validate this type of systems.

In order to use the CPN in ETL system design an in-depth study of this language's concepts was made; at the same time two of the most simple, yet very important, processes were selected as a first approach and study cases: the Surrogate Key Pipelining (SKP) and the Surrogate Key Generation. Prior to the design of each process, a theoretical study concerning its behaviour and structure was carried out, according to the methodology proposed by Ralph Kimball (Kimball and

Caserta, 2004). The first two designed models are the smallest presented here, as they model relatively simple processes and, as such, do not need hierarchy concepts to be applied. After testing the CPN modelling language in the design and validation of these simple tasks, the modelling process moved on to more complex ETL standard tasks. The Slowly Changing Dimension (SCD) was chosen to continue this study as it is composed of smaller operations (or sub-processes) and presented itself as a great study case for the introduction of hierarchical concepts.

The last ETL task to be considered in this study is the Change Data Capture (CDC). There are many different ways to implement this process; the more advanced ones are automatic and depend on the DBMS used by the operational sources. To implement the corresponding CPN model a DBMS (Data Base Management System) was selected, and its CDC process analysed, so that it could be modelled accordingly. With these three complete CPN modules (or packages), each of them representing an ETL task, it is possible to model ETL systems that are composed by these operations. For this dissertation work to be completed a CPN model for an entire ETL system, based on an example data mart, was designed with the objective of demonstrating the application of the already defined modules in the conceptual design of ETL.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Modelling ETL Processes

The volume of information generated by organizations has been growing exponentially in the last decade, due to the advances in the information technologies, which made it easier to store, query and manage large volumes of data. Today business activities are supported by the information stored in organizational data repositories and used to simplify and assist decision-makers, namely through the use of *Data Warehousing Systems* (DWS) facilities.

ETL (Extract-Transform-Load) systems are one of the most important components of a DWS (Kimball and Caserta, 2004), as they are responsible to feed the data warehouses, ensuing high levels of data quality and, consequently, adding value to decision making processes. They are formed by specific processes of extraction, cleaning and integration of data, usually taking place in a *Data Staging Area* (DSA), that is responsible for adjusting, correcting and structuring data coming from disparate information sources so that decision makers can exploit them The result is a highly specialized single repository, containing high quality data that are detailed, historic, subject oriented and non volatile (Inmon, 1996; 2004).

Usually, an ETL process comprises three main stages, extraction, transformation and loading. The first consists on the extraction of relevant information from its operational sources; in most cases the collected data has poor quality and errors that makes it inadequate to be directly used for populating a *Data Warehouse* (DW) (e.g. duplicate data, impossible or wrong values, inconsistent values due to typing errors). In the following stage – transformation – a series of rules are applied in order to increase the quality of the extracted data. This is done by correcting several errors through rectification and homogenization processes, and through the conversion of the data's format to the one used in the data warehouse, like measure units conversion, derived attributes calculation, surrogate key generation, matching of data from different sources, among other things. Finally, the populating process can be done in two distinct ways; the refresh method rewrites all the information stored in the DW and the update method updates it with new data.

The ETL is the most complex and technically challenging process among all of the data warehouse process phases (Golfarelli and Rizzi, 2009), as it easily consumes about 70% of the resources needed for its implementation and maintenance (Kimball and Caserta, 2004), as well as a big slice of the project time and budget. Technically, this is a difficult process to implement due to the high learning curve presented by a lot of the ETL tools available on the market, which don't offer the possibility to model the system conceptually, forcing the design and development to be made in an ad-hoc fashion by many organizations (Vassiliadis et al., 2002). Furthermore, a poor implementation of an ETL process, which may result in low quality information, can entirely undertake a DWS (English, 1999), bringing unbearable additional costs. For these reasons, it becomes necessary to adopt some means for conceptual modelling, design and validation methodologies for the development and implementation of ETL systems, as well as proper tools to model these systems. Despite being already the centre of many research efforts (Vassiliadis et al., 2002) (Simitsis, 2003) (Abelló et al., 2006) (Golfarelli, 2008), ETL conceptual modelling still remains almost as an island in the entire ETL life-cycle development, remaining a significant hole between the conceptual modelling and the next ETL stages, namely the ones related to logical modelling and, of course, physical implementation.

1.2. Motivation and Goals

The main objective of this dissertation work is to study and discuss the application of Coloured Petri Nets to the specification and validation of ETL systems. By proposing this new approach we hope to reduce implementation and maintenance costs, as well as the development time and risk of failure of a final DWS, which are frequently associated with a poor implementation and validation of an ETL process. As a first approach to this problem, *Petri Nets* (PN) (Petri, 1966) come into attention, being a mathematical modelling language applied to a wide variety of systems. PN are very adequate to describe and study information processing systems that are characterized as being concurrent, asynchronous, distributed, non-deterministic and stochastic (Murata, 1989). Being these some of the characteristics that are present in ETL systems, using PN to model them seems to be advantageous, once it is possible to graphically represent them and subsequently simulate the concurrent activities of the system. *Coloured Petri Nets* (CPNs) constitute a discrete event modelling language that combines the capabilities of the PN to those of a high-level programming language (Jensen and Kristensen, 2009). Additionally, they introduce hierarchic and data concepts, making them ideal for the modelling of ETL systems.

In other way, this work provides a formal and well-sustained model to specify and model the behaviour of ETL systems using the CPN modelling language. Thus, in this dissertation the carried out studies about modelling and validation of three of the most important and commonly used ETL processes are presented: the Surrogate Key Pipeline, the Slowly Changing Dimensions with Historic and the Change Data Capture cases. To conclude, each of the modelled processes is to be used as an independent package to build and model an ETL scenario that is composed of one or more instances of each of the available modules.

1.3. Dissertation Structure

In addition to this chapter, this dissertation integrates three more chapters, organized according to the various topics that were addressed in this dissertation work. In chapter 2, an introduction to the CPN modelling language is presented, starting with the characteristics, formal definition and examples of the original concept of PN, followed by the presentation of one of its high-level extensions, the CPNs - what they are, what are their characteristics -, followed by some examples of their industrial application and the advantages of using them. In this chapter, a brief introduction to the CPN modelling language is also presented by the means of an example of a CPN that models the transmission of packets through the TCP protocol between an operational source and an ETL process. The chapter is concluded demonstrating the simulation of this model in the CPN Tools environment. The CPN models created for each of the selected ETL processes are presented in chapters 3 and 4. In the beginning of each section a description of the corresponding ETL process is given prior to the presentation of the implemented CPN model and its characterization. Each section is concluded by the simulation carry out of the execution of the model produced. Finally, in chapter 5, the conclusions drawn from this study are presented, as well as some future research lines.

Chapter 2

Coloured Petri Nets

2.1. Petri Nets

PN are a mathematical and graphical modelling language applied in the description of a wide variety of systems. They are used to represent logical interactions between system components and, therefore, are very fit to model concurrent, sequential or synchronized processes, among others, in distributed systems. PN can be used as a graphical tool, allowing for a system to be designed, and its dynamic operations analysed in a visual manner. It is also possible to define the behaviour of the system through a set of algebraic equations and mathematical models.

The origin of the PN dates back to 1962, when Carl Adam Petri defended his PhD thesis "Communication with Automata", submitted to the Science Faculty of Darmstadt Technical University in Germany, in July 1961 (Petri, 1966). Petri's work had very little application in the information technologies during the early 60's. Then, computers consisted of giant mainframes that could take an entire building's room, while computer networks consisted of telephony services used for the communication of the mainframe's terminals. However, Petri, together with his staff and numerous other partners, continued his work and research on PN, until in 1970 they were featured in MIT's Project MAC Conference on Concurrent Systems and Parallel Computation (Project MAC, 1970). This paved the way for further research on PN by different entities, resulting in an accentuated growth in the number of organized workshops and released publications, such as conference proceedings on PN, reports and thesis. Since then, the PN have been applied in projects from different areas, such as Communication Protocols, Performance Evaluation, Software Design and Process Modelling, among others.

A PN is a directed bipartite graph consisting of two types of nodes – *places* and *transitions* – connected by *directed arcs*, with an initial state that is designated as *initial marking M0*. One arc can connect a place to a transition and a transition to a place, but never two nodes of the same type; a place connected to a transition is called the *input place* of that transition and a place connected by an arc from a transition is called the *output place*. In a graphical PN model, places are represented by circles, transitions by rectangles and arcs by directional arrows, which can be labelled with an integer representing their weight (omitted if the weight is one), i.e., the number of parallel arcs represented by a single arc. Each place may contain a discrete number of marks, called *tokens*, consumed or passed by transitions once they are *fired*. In order for a transition to *activate*, all of its input places need to possess at least one token that will be consumed and passed to its output place(s). Tokens are graphically represented by black dots inside the respective places and the distribution of such tokens by the existing places is called the marking of a PN model. In systems modelling, transitions represent the events or processes that cause the system's state to change while the places represent the necessary conditions for the event to take place; the tokens in a place can be looked at as resources or data items that are available, in the case of input places, or released, in the case of output places.

A PN is formally defined as a 5-tuple $PN = (P, T, I, O, M)$ where:

- $P = \{P0, P1, P2, \dots, Pn\}$ is the set of places in the net;
- $T = \{T0, T1, T2, \dots, Tn\}$ is the set of transitions is the net;
- $I(Tn)$ is the input function of transition n, indicating its input place(s);
- $O(Tn)$ is the output function of transition n, indicating its output place(s);
- $Mn = \{M_{P0}, M_{P1}, M_{P2}, \dots, M_{Pm}\}$ is the marking of the model;

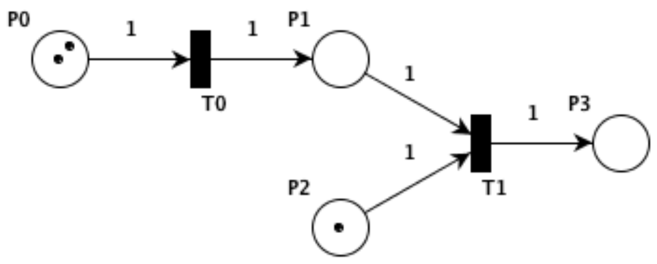


Figure 1 – Example of a PN model

- $P = \{P0, P1, P2, P3\}$
- $T = \{T0, T1\}$
- $I(T0) = \{P0\},\ I(T1) = \{P1, P2\}$
- $O(T0) = \{P1\},\ O(T1) = \{P3\}$
- $M_0 = \{2, 0, 1, 0\}$

In Figure 1 a simple PN is represented, followed by its characterization according to the previously presented definition. This PN is composed by four places, two transitions and five arcs connecting these nodes. $P0$ is the input place of the transition $T0$ and its output place is $P1$; relatively to transition $T1$, it has two input places, $P1$ and $P2$, and one output place $P3$. In the model's initial marking $M0$ there are two tokens in $P0$ and a single token in $P1$. As it has been referred, a PN model is executed by firing its transitions, which remove tokens from their input places and places them in their output places; the execution of the presented PN model is shown below.

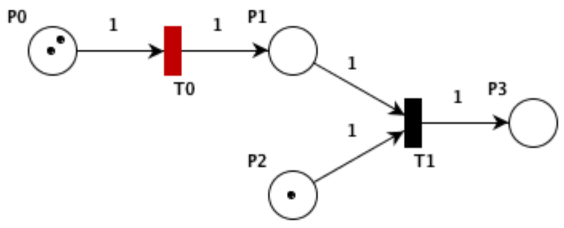


Figure 2 – Marking $M0$

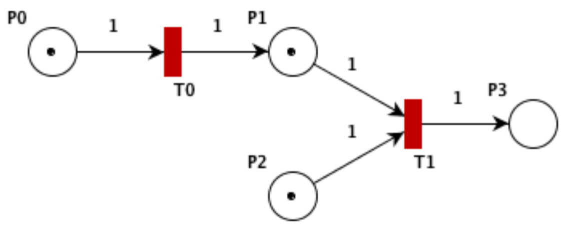


Figure 3 – Marking *M1*

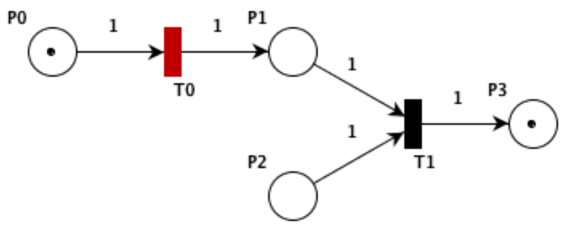


Figure 4 – Marking *M2*

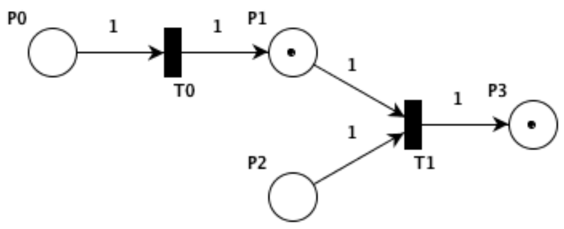


Figure 5 – Marking *M3*

In the initial marking – *M0* – displayed in Figure 2, there is a single enabled transition – *T0* – with two tokens in its input place. For a transition to be enabled there must be at least one token in all of its input places, which is not the case for transition *T1* since there is one token in its input place *P2*, but none in its input place *P1*. When the transition *T0* is fired one of the tokens is removed from its input place *P0*, passed to its output place *P1*, and the next marking – *M1* – is reached (Figure 3). In this marking transition *T1* is enabled, since there is now one token in both its input places, as well as transition *T0*, since there is a remaining token in *P0*. In this stage, as both transitions are enabled, either of them can be fired. The next marking – *M2* – is reached after transition *T1* is fired, where both of the tokens in the places *P1* and *P2* are consumed and the resulting token is passed to the output place *P3* (Figure 4). The remaining token resides in *P0* since transition *T0* wasn't fired in the previous marking and is still enabled. The final marking *M4* is displayed in Figure 5, and has no enabled transitions, since the conditions for such are not met. There is one token in *P1*, resulting from the firing of transition *T0*, which is not enough to enable transition *T1* since one of its input places – *P2* – has no tokens, and one token in the place *P3*.

As it has been referred previously, PN are adequate to describe concurrent, sequential and synchronization processes, but they can also be used to effectively model and manage conflict and mutual exclusion processes, which are very common in real systems. The next presented examples illustrate the use of PN to model these activities.

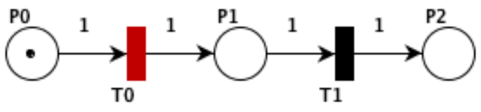


Figure 6 – Sequential execution

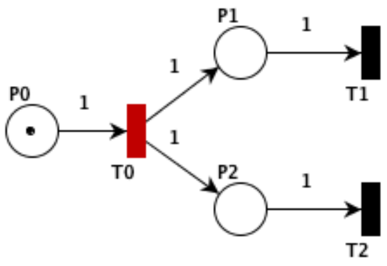


Figure 7 – Concurrency

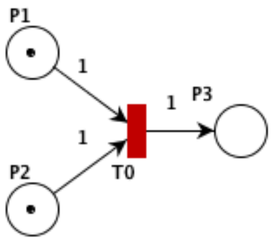


Figure 8 – Synchronization

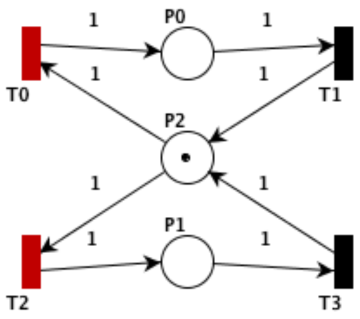


Figure 9 – Mutual exclusion

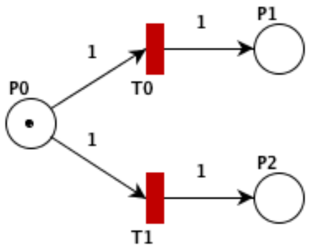


Figure 10 – Conflict

The modelling of a process's *sequential execution* is represented in Figure 6. In this net transition *T1* can only be fired after transition *T0*, which is the one with consumable tokens in its input place (i.e., *P0*), hence the precedence constraint '*T0 → T1'* is enforced. In Figure 7, we can see a *concurrent* process represented by transitions *T1* and *T2*. When transaction *T0* is fired, the token removed from *P0* is passed to both of the transition's output places – *P1* and *P2* – simulating a concurrent (or parallel) execution. The model presented nest (Figure 8) represents the *synchronization* between two concurrent processes. In this example the input places of the transition represent the resources freed by each process. In this case, transition *T0* activates only when it has enough resources from its concurrent processes, that it, when there is at least one token in each of its input places. Two *mutual exclusive* processes are represented in Figure 9. These kinds of processes cannot be executed at the same time if they have shared resources that are consumed by a single process at a time. In this example the tokens in the place *P2* (i.e., the input place of transitions *T0* and *T2*, and the output place of transitions *T1* and *T3*) represent the shared resources used to fire either transition *T0* or transition *T2*. The final example (Figure 10) represents two processes in *conflict*. Transition *T0*, representing the activation of the first process, and transition *T1*, representing the activation of the second process, are both enabled but only one of them can be fired, leading to the disabling of the other transition.

The use of PN is very advantageous in systems modelling. The examples presented previously clearly illustrate the simplicity of these nets, being their creation and manipulation quite easy and straightforward. They are also very adequate to describe common system activities, e.g., concurrency, synchronization, or conflict. This modelling tool is also very scalable as the created models can grow and be adapted as the system's size and complexity increases. For instance, if the number of resources increases, one can simply increase the number of tokens in certain places without any other modification in the entire model. The PN, as a graphical tool, provides a way of visualising the implemented models, which is very useful for the system modeller as the modelling can be done exclusively in a GUI, but also for the users of the system that can easily analyse and study its behaviour.

One disadvantage of the PN is that their lack to support time, making it impossible to model systems where time plays a major role, which is the case for many real systems. Another disadvantage of these nets is that there is no distinction is made between the existing tokens, and the lack of hierarchy concepts. Because of this, the created models tend to grow exponentially as the modelled system's size increases, since it is impossible to structure the net into different modules, making it difficult to graphically analyse and manage.

To overcome these weaknesses, as well as to enhance their modelling power, several extensions to the original PN have been developed and new properties added, such is the case of the Time Petri Nets (Popova, 1991), where each transition is associated with a *firing time*. Hierarchy concepts have also been added to these nets, allowing for different abstraction levels to be created and analysed independently. The assignment of a data type and a value to each token led to the creation of the commonly called high-level Petri Nets, such as the Coloured Petri Nets. There are many more extensions to the original PN. In this work the Coloured Petri Nets will be used for the ETL system modelling.

2.2. Coloured Petri Nets

Coloured Petri Nets (CPN) (Jensen, 1994; 1997; 1998), an extension to the original Petri Nets, is a graphical language that, much like is predecessor, is very adequate for design, specification, simulation and validation of a wide variety of systems with concurrency, communication and synchronization as their main characteristics. It was created in 1979 by the CPN group – Aarhus University, Denmark – to fill two existing gaps in the original PN that hampered their practical application in systems of the size and complexity normally found in industrial projects: the inexistence of data and hierarchical concepts (Jensen and Kristensen, 2009).

The foundations of the CPN graphical notation come from the low-level PN, and they are also governed by the same set of basic rules, with some minor alterations. A CPN is also represented by a directed bipartite graph with *places* and *transitions* (i.e., nodes) connected by *directed arcs*, and an initial state called *initial marking*. The arcs connect places to transitions and vice-versa. A place is designated *input place* if it directed, by an arc, to a transition, or *output place* if a transition is directed to that place. These objects are graphically represented in the very same way as in the low-level PN: *circles* or *ellipses* for places, *rectangles* or *squares* for transitions and *directed arrows* for arcs. The places and transitions, however, can now be labelled with a unique name and the arcs can possess an *arc expression* instead of its weight. Once again, places may have a discrete number of *tokens*, consumed when a transition is *fired*. A transition may fire if it is *enabled*, which only happens when all of its input places possess at least one token.

In contrast to what happens in the original PN, where tokens, simply drawn as black dots, were indistinguishable, in CPN each token has a data value, i.e., a *colour*, of a predefined type, i.e., a *colour set*. The primitives used for data types definition, as well as the manipulation of data, are provided by CPN ML (CPN ML, 2012), a high-level functional programming language based on Standard ML (Standard ML of New Jersey, 2012); with the introduction of the *colour* concept every token is differentiated from each other, allowing for a more precise and more detailed description of the modelled system. For this reason CPNs are referred to as high-level PNs, but they also support the valuable concepts of hierarchy and time.

With the CPN modelling language, it is possible and convenient to build models for large and complex systems, which are composed by several smaller nets. The idea is to use a number of modules that are related and communicate to each other through a well-defined interface, in order to construct a large model. The main advantage in building a hierarchical CPN is that each of the modules can be designed, tested and validated independently, allowing the system modeller to work both bottom-up or top-down, and also allowing for a system to be built and visualised with several levels of abstraction, resulting consequently in smaller and more compact models. The time taken to execute certain system activities can also be captured through the time concept, also included in the CPN modelling language, allowing it to be applied in the modelling and validation of real-time systems.

A given CPN model is executable and is both state- and action-oriented, i.e., it describes the events that make the state of the system to change. Using CPN Tools, a tool that supports the construction and manipulation of CPNs, it is possible to simulate the behaviour of the modelled systems. Two kinds of simulations are available: *automatic* and *interactive*. In an automatic simulation there is no user interaction and the model is executed as fast as possible, much like a program execution, in order to test its overall speed and performance. An interactive simulation consists on a step-by-step execution of the model where the user determines the next step, by choosing between the enabled transitions, and observes the effects of each individual step graphically. This kind of simulation is very useful for a detailed analysis of the model, for investigating the different existing scenarios in the net and to check the behaviour of the model.

As far as we know, CPNs haven't yet been applied in the design and validation of ETL systems; they have, however, been successfully applied in numerous industrial and academia projects worldwide from distinctive areas, such as software development, workflow and business processes, protocols and networks, and even military systems. Concerning the Protocols and Networks area, the CPN have been used to model the different versions of the TCP protocol and to study and analyse its behaviour and performance (Figueiredo and Kristensen, 1999). They have also been used to model a switched LAN network and its response time evaluated through CPN Tools (Zaitsev, 2004). In an industrial project at Ericsson Telebit, the CPNs were applied in the development of the edge router discovery protocol (ERDP) for mobile ad-hoc networks (Kristensen and Jensen, 2004). The CPNs were used for the design and specification of this protocol, allowing for the investigation of its behaviour, as well as the detection of several design errors in an early stage of the development.

Other major companies, although in different areas, have also used the CPN modelling language and its supporting computer tools in industrial projects development. Nokia used them to model and analyse the interactions between the user and the interface (UI) of a new family of mobile phone software (Xu and Kuusela, 2001) and, in a different project, they were used to model and simulate the features of the company's mobile phones (e.g., voice calls, text messages, games) interaction patterns (Lorentsen et al., 2001). Hewlett-Packard has also used the CPN in industrial projects at the HP Laboratories in Palo Alto, USA. In one of these projects the CPN were used to model and analyse their on-line transaction processing system, allowing for key decisions about the system's structure to be evaluated from the simulation of the constructed model (Cherkasova et al., 1993). The CPN have also been applied in other kinds of systems, familiar to everyone in the every-day life, such as in the modelling of traffic control systems (Perkusich et al., 1999), for the visualization and analysis of such system's behaviour, and also in the analysis of the

planning process for airplane arrivals in air traffic control (Oberheid and Soffker, 2008). A complete list of industrial applications of CPNs can be consulted in (AarhusUniversity, 2012).
The advantages of using CPN in systems modelling are quite evident and some of them have already been described, such as the existence of an appealing graphical notation, the possibility to carry out simulations, in order to test and validate wide variety of systems, and a hierarchic representation that, together with CPN ML, makes it possible to create compact models. Many of the CPN concepts are also present in other programing languages that system modellers are certainly familiar with, lowering its learning curve; it is also possible to add time concepts to models and to verify the systems' properties through the model's formal representation. The CPN's supporting tools, such as the Design/CPN and more recently CPN Tools, allow for the implementation, simulation, analysis and validation of the system's models, paving the way for the practical application of the CPN in large industrial projects.

2.3. The CPN Modelling Language

To introduce CPN modelling language, as well as the functional language used in CPN based models, an example that represents the transmission of data from an operational source (OS) to the corresponding ETL process, using the *Transmission Control Protocol* (TCP), is used (figure 11). The implemented model represents the *establishment of a connection* between two hosts, an OS and an ETL process, so that some segments of data can be transferred from an OS to a *Data Staging Area* (DSA) of a Data Warehouse, and then used by an ETL process. After the transference of data is completed the *connection is terminated*. A practical example of such a system is, for instance, the request of the audited data in an operational source by a *Change Data Capture* (CDC) ETL process. This example focuses exclusively on the establishment and termination of the connection between the two hosts. The data transfer process is represented in this example by the simple transmission of segments, and the respective acknowledgments, over the network and no data is loss during its transmission to keep the model simpler. The modelling of a communications protocol over an unreliable network, using acknowledgments, sequence numbers and retransmissions to assure that data packets are delivered in order is described in detail in (Jensen and Kristensen, 2009).

In TCP, the process for establishing a connection between two hosts in a IP based network is often called the three-way handshake, TCP-handshake or SYN-SYN-ACK, and is designed so that the network and port parameters can be negotiated between the two computers trying to communicate. The host trying to establish a connection sends a SYN (synchronize) packet to the second host, upon receiving it, a SYN-ACK (synchronize-acknowledgement) packet is sent back; at this stage the second host has acknowledged that the SYN packet was received and sends its own SYN packet. Finally, the first host replies with an ACK packet and, when received by the second host, the connection is established. For the connection to terminate, each side has to send a FIN packet and await the reception of an ACK packet from the other side, meaning that four packets are used, at most, in this process. A three-way handshake is also commonly used to terminate the connection; the second host, upon receiving the first FIN pack, can send a FIN-ACK packet to acknowledge the reception of the first packet and issuing the termination of the connection at the same time. In this model, these types of packets are represented by their flags (e.g., 'SYN', 'ACK', 'FIN'). The exact structure of the TCP segment is not modelled in this example so that the presented model is simple and easier to read, but complex enough for the presentation and explanation of the general concepts of the CPN modelling language and CPN ML.

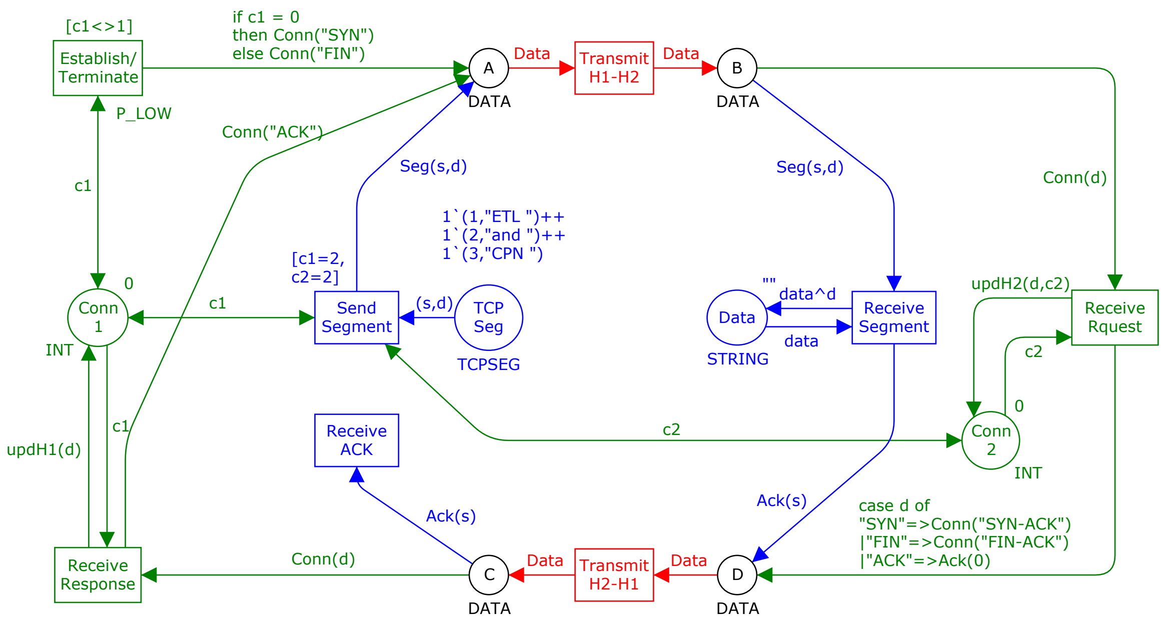


Figure 11 – Example CPN of the TCP protocol

The implemented model that describes the establishment and termination of a connection in the TCP protocol is presented in Figure 11. The left part of the model describes the *OS*, while the right part the *ETL Process*, and the middle part (in red) models the *Network*. The green part of the model represents the *connection management* (i.e. establishment and termination of the connection) for both hosts and the blue part models the transmission of packets over the network. The net is composed of eight *places*, eight *transitions*, directional arcs connecting these *nodes* and several inscriptions next to the arcs, places and transitions, written in CPN ML. The names of the places and transitions are mandatory, and are presented inside the circles or rectangles. They have no formal meaning, but are of great importance as they improve the model's readability. The state of the modelled system is represented by the places, together with the existing tokens that mark them, while the transitions represent the actions that cause the system's state to change.

2.3.1. Places, *colour sets* and declarations

The state of the OS is modelled by the places TCP Seg, representing the TCP segments that will be transmitted over the Network from the OS to the ETL process, and Conn1 that models the status of OS's connection. The state of the ETL process is modelled by the place Data, representing the DSA with the data received from the OS, and Conn2 that models the status of this host's connection. The Network's state is represented by the places A and B, representing any kind of data (e.g. data packet, SYN packet, or FIN packet) before and after it is transmitted from the OS to the ETL process, and C and D, representing data being transmitted from the ETL process to the OS.

The *colour sets* of the places (i.e., their type) can be defined by the means of an inscription usually written bellow the place and is used to define the set of *colours* (i.e., data values) that the tokens in that place can have. In this example, the existing colour sets are written in uppercase letters below the corresponding places. The places Conn1 and Conn2 have the colour set INT. An integer is used to represent the connection status of both hosts, where '0' stands for *no connection*, '1' for *pending connection or awaiting time out to disconnect* and '2' for *connection established*. The places A, B, C and D have the colour set DATA, the place TCP Seg has the colour set TCPSEG, and, finally, the place Data has the colour set STRING. In CPN ML, colour sets are defined using the keyword *colourset*; lets see how the referred colour sets can be defined in CPN ML:

```
colset INT = int;
colset STRING = string;
```

The colour set INT is defined to be equal to the integer type and the STRING colour set is defined to be equal to the string type. This means, that the tokens in Conn1 and Conn2 must have values of the integer type and the tokens in Data must be strings. These colour sets are called *simple* as they are not constructed from other colour sets. Other examples of simple colour sets are Units, Booleans, Enumerations and Indexes. The TCPSEG and DATA colour sets are implemented as:

```
colset TCPSEG = product INT * STRING;
colset DATA = union Seg:TCPSEG + Ack:INT + Conn:STRING;
```

These are *compound* colour sets as they are constructed from other, existing, colour sets. The colour set TCPSEG is defined to be the product of the INT and STRING colour sets and the tokens of this type are represented by a pair where the first element represents the sequence number of the TCP segment an the second element represents the data in that segment. The colour set DATA is the union of the TCPSEG, INT and STRING colour sets and models the different types of data packets that can be transmitted over the network: TCP segments (Seg), acknowledgment of the transmitted segment (Ack) and the segments used to manage the connection (Conn). This colour set is used so that the places A, B, C and D can handle these different colour sets at the same time, by using the correct constructor in the arc expression when passing a token to one of these places. The network module is greatly simplified by using this colour set as the two existing transitions and four places are sufficient to transmit three different kinds of packets.

Places also have an optional inscription used to define their initial marking, usually written above the circle representing the place. The places Conn1 and Conn2 have the value '0' as their initial marking, meaning that in the model's initial marking, both of these places have one token with the colour '0', indicating that they are disconnected in the beginning of the model's execution. The initial marking of Data is "", i.e., an empty string, indicating that no data has been received by the ETL process in the beginning of the execution. The initial marking of the place TCP Seg is '1`(1,"CPN ") ++ 1`(2,"and ") ++ 1`(3,"ETL ")', which represetns the packets that will be transmitted: three tokens of the colour set TCPSEG, i.e., a product of an integer and a string. The ++ and ` operators allow for the construction of a multi-set of token colours. The number before the ` operator indicates the number of that colour's occurrences and the ++ operator returns the union of two multi-sets. The places' initial marking can also be defined as a *constant* value and used as an inscription in the place, which is useful when a place's initial marking is formed by a large multi-set. The initial marking of TCP Seg is be defined as:

```
val TCPsegments = 1`(1,"ETL ")++ 1`(2,"and ")++ 1`(3,"CPN ")
```

2.3.2. Transitions, arcs and arc expressions

The events that cause the system's state to change are represented by the eight existing transitions. The Establish/Terminate transition is found in the OS part of the model and is responsible for establishing the connection with the ETL process, terminating it after all the TCP segments are transmitted. The connection establishment and termination requests are received and processed by the transition Receive Request on the right part of the model. The response is received and processed in the OS side by the transition Receive Response. The transitions in the Network part of the model are responsible for transmitting the data over the network from the OS to the ETL process and vice-versa. Finally, the transition Send Segment is responsible for sending the TCP segments (one at a time) to the ETL process, where they are received and processed by the transition Receive Segment; the acknowledgment of the segment sent to the ETL process is received in the OS side by the transition Receive Ack.

In CPN models transitions are also allowed to have inscriptions in CPN ML. Apart from the transition name, three types of inscriptions can be associated with transitions and they are all optional. Guard expressions are CPN ML Boolean expressions that evaluate to true or false and are surrounded by square brackets. A guard expression is used to test the input arc expression, preventing the transition from enabling if the expression evaluates to false - a guard may also be formed by multiple expressions. It is also possible to use time expressions to delay the firing of a transition, and code segments, but these two types of inscriptions weren't used in model's transitions. A priority value can also be associated with a transition, even though this isn't considered to be an expression. All transitions have the normal priority value by default and a transition with lower priority is enabled only if none of the normal priority transitions is enabled.

Like the PN, when a transition is fired a token is removed from the transition's input places and placed in its output places. The colour of the tokens passed to the output places are determined by the arc expression in the corresponding arc. *Bidirectional arcs* can also be used in CPN models. A place connected to a transition by this type of arc is both an input and output place for that transition, when the transition is fired a token is removed from the input place, used by the transition, and returned to the place. The arc expressions are written in CPN ML and can be formed by variables, operators, constants or functions. The variables used in CPN models have to be bound to a value of a given colour set and can be defined with the keyword *var* as:

```
var s, c1, c2 : INT;
var data, d: STRING;
var Data: DATA;
```

The variables `s`, `c1` and `c2` must be bound to an integer value. For example, the variable `c1` in the arc expressions connecting the place Conn1 with the transitions Establish/Terminate, Receive Response and Send Segment must be bound to an integer value that represent the status of the connection (i.e., '0', '1' or '2'). The same happens when using this variable in the guard expressions of the transitions Establish/Terminate and Send Segment. The variable `c2` is used to represent the connection status of the host holding the ETL process and the variable `s` is used to represent the sequence number of a TCP segment. The variables `d` and `data` are bound to string values and are used to represent the data in the TCP Segment transmitted over the network. The expression `(s,d)` (i.e., an integer and a string) in the arc connecting the place TCP Seg with Send Segment is composed by two of the just described variables and is used to represent a token with a value of the `TCPSEG` type, i.e., a product representing the TCP segment. The final defined variable – `Data` – is used to represent any kind of packet transmitted over the network, so that only four places with the same colour set (`DATA`) and two transitions are needed to represent the network. Notice how some of the arcs connected to these places, and connecting them to the transitions, use the defined constructors; for instance, the arc expression `Seg(s,d)`, in the arc connecting Send Segments to A, is used to tell that the data value that will be bound to the variable `Data` in the arc expression connecting A to Transmit H1-H2 represents a TCP segment with the values that are bound to `(s,d)`.

2.3.3. Control structures: if-then-else and case

Now lets analyse the arc expressions that use the available control structures: if-then-else and case. The arc connecting the transition Establish/Terminate to the place A has the following expression:

```
if c1 = 0
then Conn("SYN")
else Conn("FIN")
```

When the transition is fired a connection packet is generated in this arc expression according to the connection status, i.e., the value of `c1` removed from `Conn1` by the transition. If `c1` is bound to '0', then the OS is not connected and a SYN is sent to the ETL process. Otherwise, if a connection is established, a FIN is sent to the ETL process. In this arc expression, the possible values bound to `c1` are '0' or '2', since the transition is not allowed to activate if the value bound to `c1` is '1' as is implemented by the guard expression `[c1 <> 1]`. To illustrate the use of the case control structure let's look at the expression in the arc connecting the transition `Receive Request` to D:

```
case d of
"SYN"=>Conn("SYN-ACK")
|"FIN"=>Conn("FIN-ACK")
|"ACK"=> Ack(0)
```

This expression is used to determine which kind of response this host, depending on the received request, sends. If a SYN packet is received then a SYN-ACK is sent, if a FIN is received then a FIN-ACK is sent. It is also possible for an ACK packet to be received in the `Receive Request` transition; in this case nothing is sent over to the OS but a token must be placed in place D for the correct execution of the model, so an acknowledgment token is created, simulating the request of the first TCP segment and an indication that the connection was successfully established. Notice how the different constructors have to be used in order to identify the kind of packet sent to the place D with colour set `DATA`.

In order to save some of the model's space, and to make it more appealing, these control structures could be implemented as functions that would be used as the actual arc expressions. In fact every expression involving operations between variables, control structures and other functions can be defined as functions and used in arc expressions, guard expressions and initial markings. Functions are defined using the keyword *fun*, followed by the function name and a list of parameters, separated by commas, inside parenthesis; in the implemented model two functions – `updH1` and `updH2` – are used to update the connection status of the OS and the ETL process, respectively; they are defined as:

```
fun updH1(d) =
     case d of
     "SYN-ACK" => 2
     |"FIN-ACK" => 1

fun updH2(d,c) =
     case (d,c) of
     ("SYN",0) => 1
     |("ACK",1) => 2
     |("FIN",2) => 2
     |("ACK",2) => 0
```

The first function is used in the expression of the arc connecting Receive Response to Conn1 and is responsible for passing the value '2' (i.e., connection established) if a SYN-ACK is received and the value '1' (i.e., pending connection) if a FIN-ACK is received. The second function updates the status of the connection in the ETL process according to the received connection packet (in Receive Request) and the current status of the connection. This value is bound to the variable c2, which is also passed to Receive Request, when this transition is fired and used as the second parameter of the function.

2.3.4. Fusion Sets

As it has been referred, the process of transmitting TCP segments over the network can only begin when both hosts have an established connection, that is, the transition Send Segment is only enabled once the places Conn1 and Conn2 have a token with the colour '2'. This is expressed in the guard expression of this transition as [c1=2, c2=2], which means that the token representing the connection status of the second host needs to be exchanged between the modules representing both hosts. This is done by the bidirectional arc connecting Conn2 with Send Segment. *Fusion places* allow for places in different modules to represent a single place and can be looked at as way of reducing the number of arcs in a CPN model. So, when fusion places are used, the existing tokens are shared between all the places that are part of the same *Fusion Set*. The connection status of the ETL process can be implemented as a fusion set:

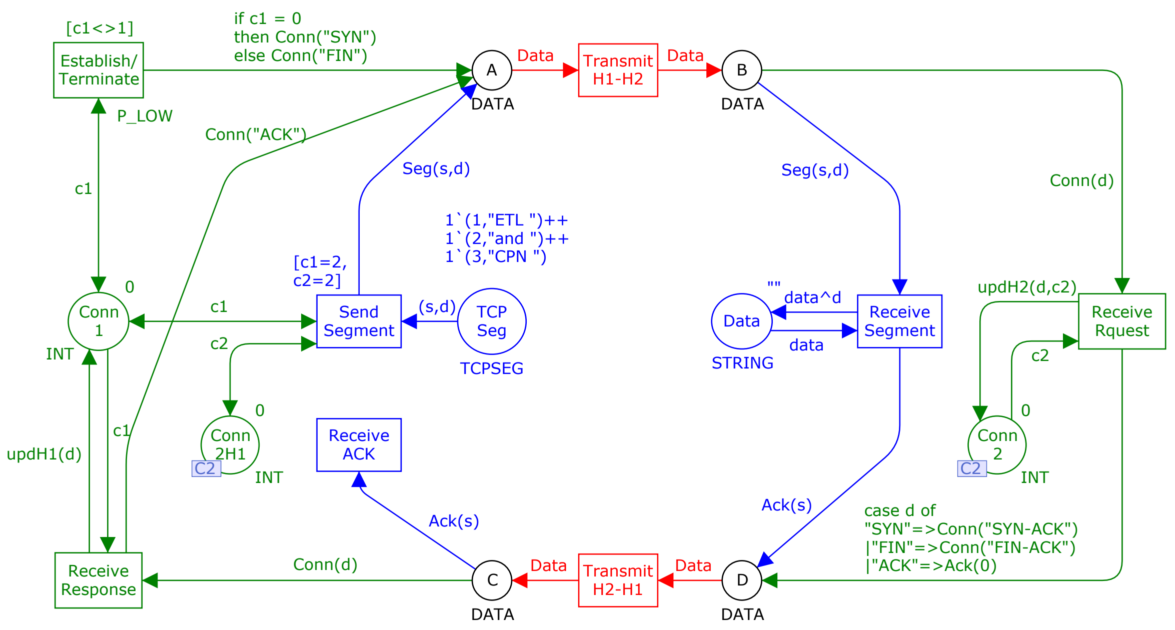


Figure 12 – The TCP Protocol CPN model implemented with a Fusion Set

Using the fusion set C2, the status of the connection in the ETL process can have an independent representation in both hosts, which is very useful when a hierarchical representation of this model is designed as the modules representing the OS and the ETL process can communicate, in order to exchange the connection status token, without an additional interface.

2.3.5. Hierarchy

The model described so far, although representing three separate modules (Operational Source, ETL Process, and Network), is not organized as a hierarchical network. The presented system is simple and resulted in a small and simple CPN model that can easily be analysed as a whole, but this is not always the case. When designing models of large and complex systems it is very convenient to implement them as a set of modules that are organized hierarchically. First of all, it is very inconvenient to draw such a system in a single net, and difficult to produce an appealing layout, which would significantly reduce the readability of the model. If different parts of the model are organized as a set of modules the modeller can draw and view the model in different abstraction levels, focusing on fewer details at a time. It is also possible to reuse the already designed modules in the same net, saving time in redesigning components that are frequently used in different parts of the system. To describe how modules can be used in the design of CPN based models, the previous example (Figure 12) is transformed into a *hierarchical CPN* by creating a module for the Operational Source, a module for the ETL Process, and a module for the Network. The module standing on top of this hierarchy (i.e., the *prime module* or *top-level module*) represents the entire system and is presented in Figure 13.

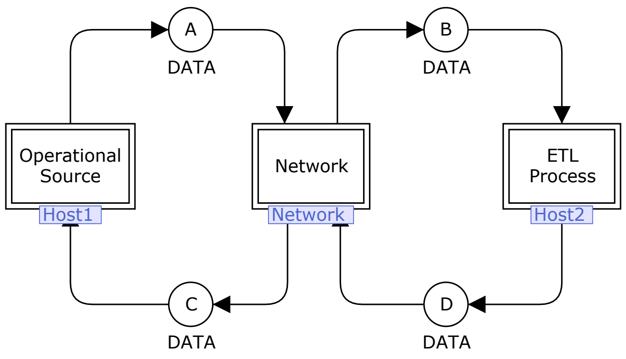


Figure 13 – The TCP protocol module (prime module)

The top-level module has three *substitution transitions* (i.e., the double lined rectangle boxes), each with a blue *substitution tag* that indicates the name of the sub-module represented by each transition. These substitution transitions are used to hide the more complex details about each of the represented sub-modules, allowing for an abstract view of the modelled system. The input place of a substitution transition is called an *input socket*, while the output place is called *output socket*. Lets now analyse how each of the sub-modules are organized. The module representing the OS is shown in Figure 14.

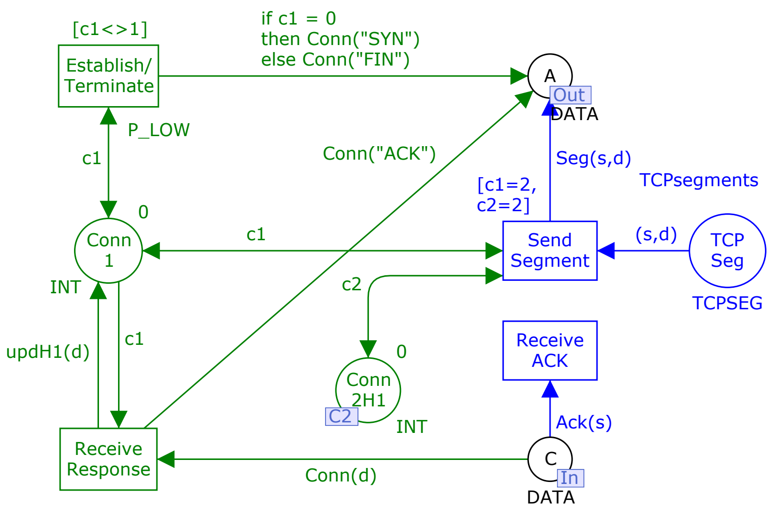


Figure 14 – Operational Source module

The module representing the OS integrates four transitions and five places, as already described. The place A is now called an *output port* and place C an *input port,* and together they form the *interface* that allows for this model's communication with the remaining modules. In order to communicate with the prime module, these *port places* must be related, through a *port-socket relation*, with the socket places A and C in the prime module described before. Although there is an independent representation of these places in two different modules, they are the same places and possess the very same tokens, independently of the module they are being visualised in. The places Conn1 and TCP Seg are internal to this module and the remaining place, Conn2H1, is part of the fusion set C2.

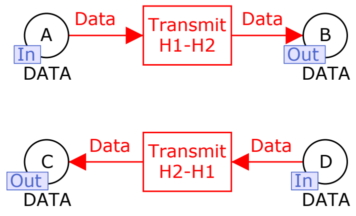


Figure 15 – The Network module

The Network module (Figure 15) is the simplest of the three sub-modules. The input ports are the places A and D and the output ports are the places B and C. All of these ports need to be related to the corresponding sockets in the prime module.

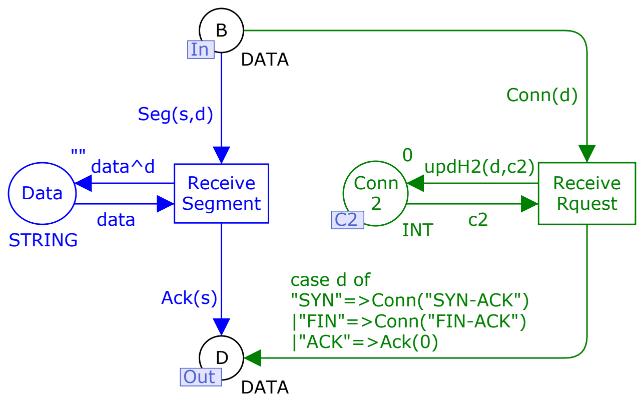
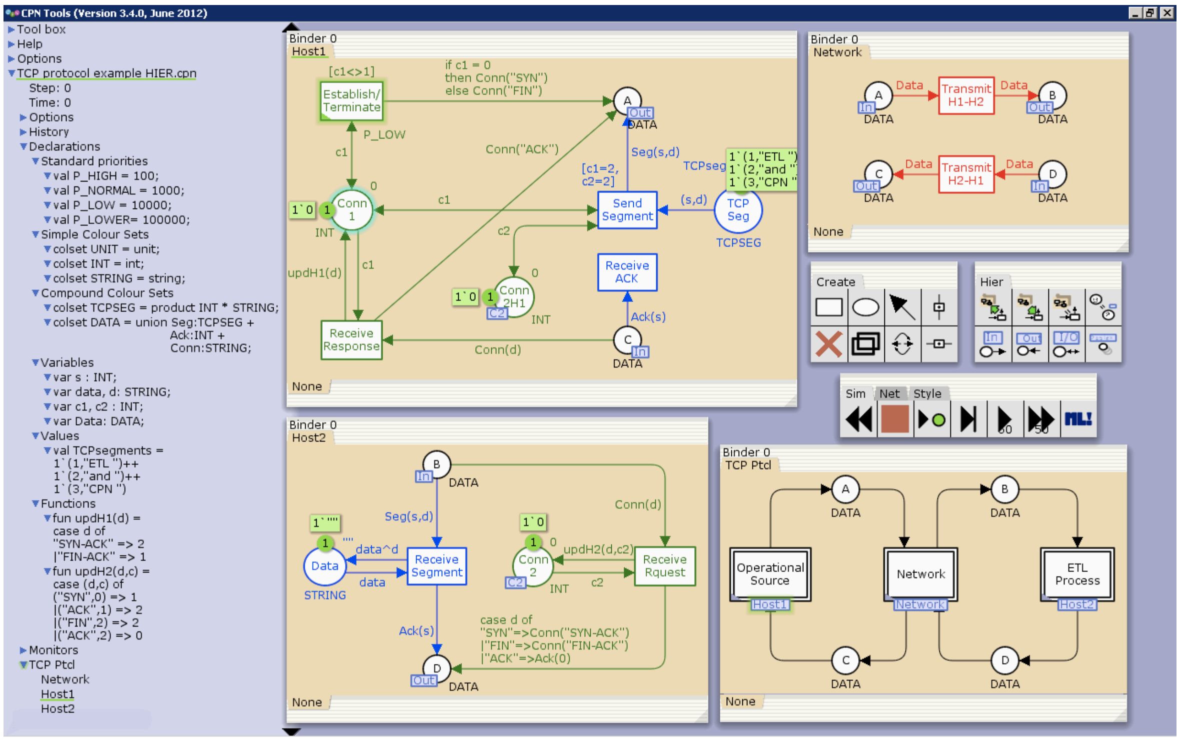


Figure 16 – The ETL Process module

Finally, the module representing the ETL process (Figure 16) is formed by two transitions and four places. B is the input port and D is the output port, while Data is an internal place. If the fusion set C2 was inexistent (Figure 11), the place Conn2 would assume the role of an output port place when this module was defined – the same would happen in the Operational Source module. There would also be an extra socket place in the prime module acting as the output socket of the substitution transition ETL Process and as the input socket of the substitution transition Operational Source.

2.4. Simulation of CPN models with CPN Tools

CPN Tools, the substitute of Design/CPN, is a tool that supports the construction and manipulation of CPN models, and therefore allows for their practical application, again developed by the CPN Group in Aarhus University (Jensen et al., 2007; Jensen and Kristensen, 2009). With this tool it is possible to edit, simulate and analyse CPN based models through a graphical user interface (GUI). This is an open-source tool, with over 10 000 licenses spread across 150 countries, currently available for Windows XP and Windows Vista. The system modeller works directly in the graphical representation of the CPN model through CPN Tool's GUI (Figure 17).



```
CPN Tools (Version 3.4.0, June 2012)
▶Tool box
▶Help
▶Options
▼TCP protocol example HIER.cpn
    Step: 0
    Time: 0
  ▶Options
  ▶History
  ▼Declarations
    ▼Standard priorities
      ▼val P_HIGH = 100;
      ▼val P_NORMAL = 1000;
      ▼val P_LOW = 10000;
      ▼val P_LOWER= 100000;
    ▼Simple Colour Sets
      ▼colset UNIT = unit;
      ▼colset INT = int;
      ▼colset STRING = string;
    ▼Compound Colour Sets
      ▼colset TCPSEG = product INT * STRING;
      ▼colset DATA = union Seg:TCPSEG +
                           Ack:INT +
                           Conn:STRING;
    ▼Variables
      ▼var s : INT;
      ▼var data, d: STRING;
      ▼var c1, c2 : INT;
      ▼var Data: DATA;
    ▼Values
      ▼val TCPsegments =
        1`(1,"ETL ")++
        1`(2,"and ")++
        1`(3,"CPN ")
    ▼Functions
      ▼fun updH1(d) =
        case d of
        "SYN-ACK" => 2
        |"FIN-ACK" => 1
      ▼fun updH2(d,c) =
        case (d,c) of
        ("SYN",0) => 1
        |("ACK",1) => 2
        |("FIN",2) => 2
        |("ACK",2) => 0
  ▶Monitors
  ▼TCP Ptcl
      Network
      Host1
      Host2
```

Figure 17 – The CPN Tools GUI

The CPN Tools GUI has two main areas: the *index*, which is the smaller rectangle area on the left side, and the *workspace*, located on the right side of the toolbox. The index contains a selectable set of tools, in the *Toolbox* section, that are used for the manipulation of the CPN models. In this Toolbox are included other tools for the *creation*, *simulation*, *creating hierarchical models* and *edition of their graphical layout*, among others. The index also includes an *overview* of the CPN models that are open. In this example the overview of the file 'TCP protocol example HIER.cpn' is presented, which is the name of the hierarchical CPN example described previously. Besides the model's *name*, in the overview are also presented the model's *declarations*, *modules* and *hierarchy structure*. The declarations area is used for defining (and viewing) the model's *priorities*, the *implemented colour sets*, in this case divided into simple and compound colour sets, and also the defined *variables*, *constant values* and *functions*.

The *workspace* of CPN Tools is where the graphical design and implementation of CPN models is actually done. This area uses a set of binders (i.e., the rectangle boxes) that can be dragged from the index into the workspace. For instance, each module of the CPN model can be dragged from the hierarchy structure section, in the bottom of the index, into the workspace, creating an independent binder for the dragged module. In figure 18, there are four binders containing each implemented module, and three extra binders that contain the create, simulate and hierarchy *tool palettes*. A tool palette is also dragged from the Toolbox section in the index into the workspace and each binder can have multiple tabs to save space in the working environment.

With CPN Tools' *simulation* tool it is possible to simulate the execution of the implemented model, while analysing the data flow and the results of each step of the model's execution, in the same working environment. An *interactive* simulation of the implemented net is now presented using *screenshots* taken in different markings of the CPN model. This simulation will consist on the execution of the *connection establishment* process, starting with both hosts *disconnected* and finishing when the connection is *established*, enabling the process responsible for transmitting TCP segments. Figure 18 shows the initial marking of the TCP protocol model (i.e., Marking *M0*); the marking of each place, if existent, is represented by a small green circle indicating the number of tokens in that place, while the actual token colours are shown in the adjacent green box that can optionally be hidden to present a cleaner view of the model.

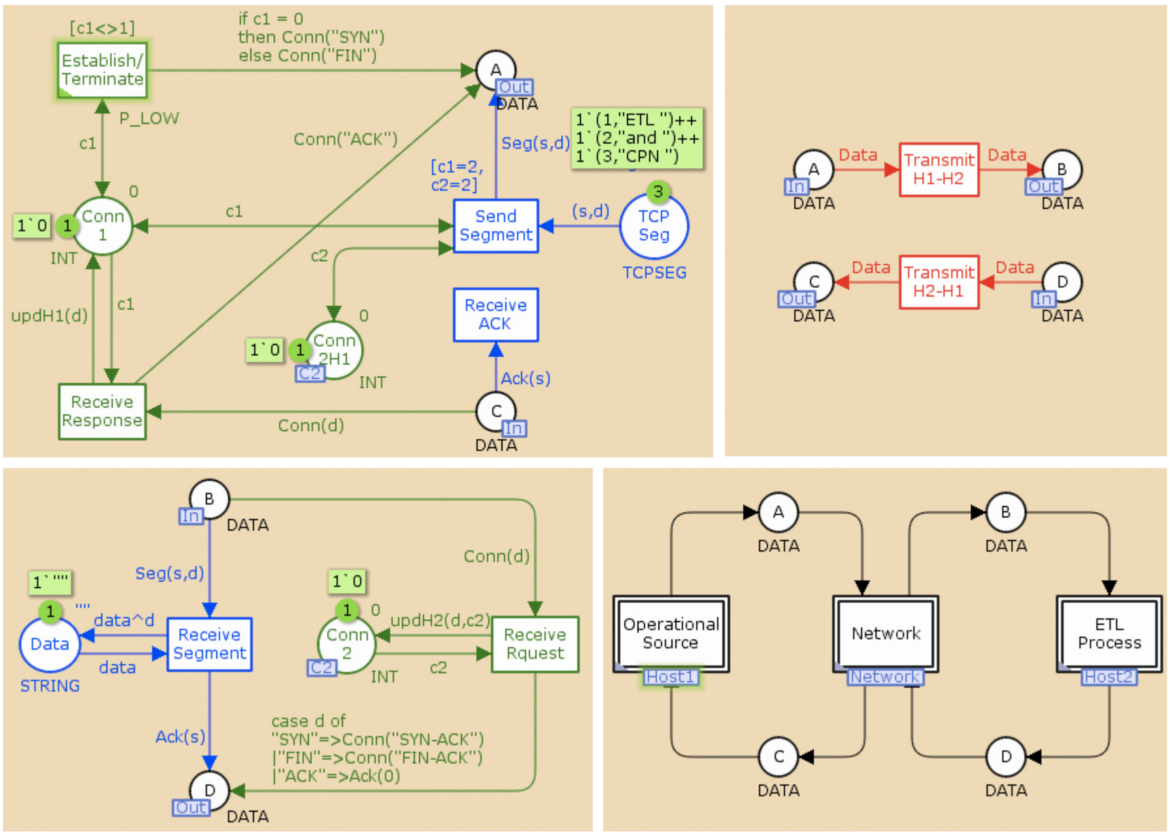


Figure 18 – The initial marking *M0* (Opr. Source, Network, ETL Process and Prime modules)

In the initial marking (Figure 18) there are three tokens in the place TCP Seg, representing the segments that will be sent to the ETL process once the connection is established, and one token with the colour "", an empty string used to indicate that no data has been received yet, in the place Data. There is also one token in Conn1 and in Conn2, both with the colour '0', representing the status of the connection, in this case disconnected. The place Conn2H1 shares its token with the place Conn2 since they are part of the same fusion set C2. The only enabled transition in the initial marking is Establish/Terminate in the Operational Source module, which is represented by a green shading surrounding the transition, as there is one token in its input place and the guard expression evaluates to true (i.e., the value bound to c1 is different than '1'). When the transition is fired, the token with colour '0' is removed from Conn1, and used to generate the appropriate connection token through the transition's output arc expression. The next marking (Figure 19) is reached in this step.

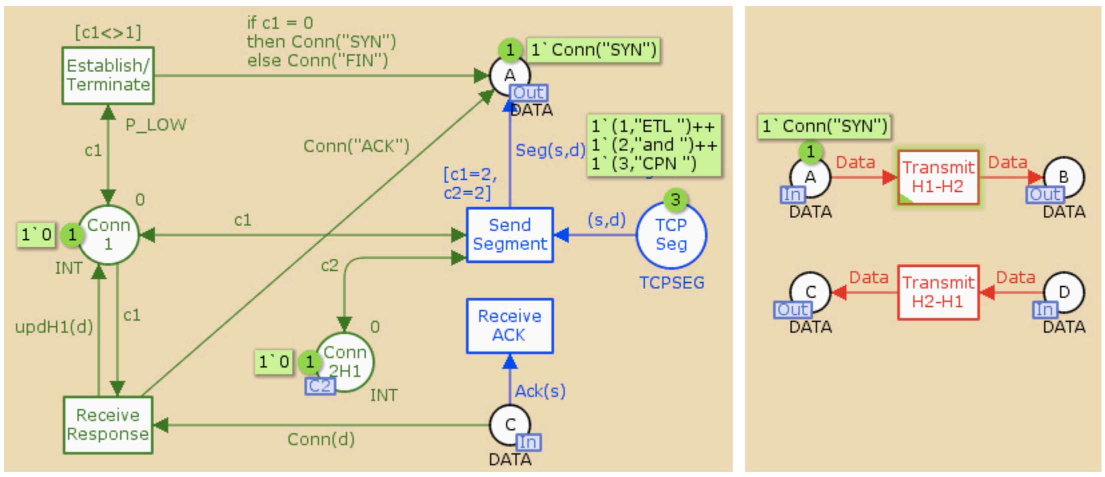


Figure 19 – The marking *M1* (Operational Source and Network modules)

In marking *M1* there is a new token in the place A with the colour 'Conn("SYN")', representing a packet requesting the establishment of the connection – notice how both representations of the place A, one in each module, contain the same token, as this place acts as the modules' communication interface. The markings of the remaining places remain unaltered. The token in Conn1 was returned to it after transition Establish/Terminate was fired as its input arc is a bidirectional arc. Transmit H1-H2 in the Network module is now enabled, and marking *M2* (Figure 20) is reached after it is fired.

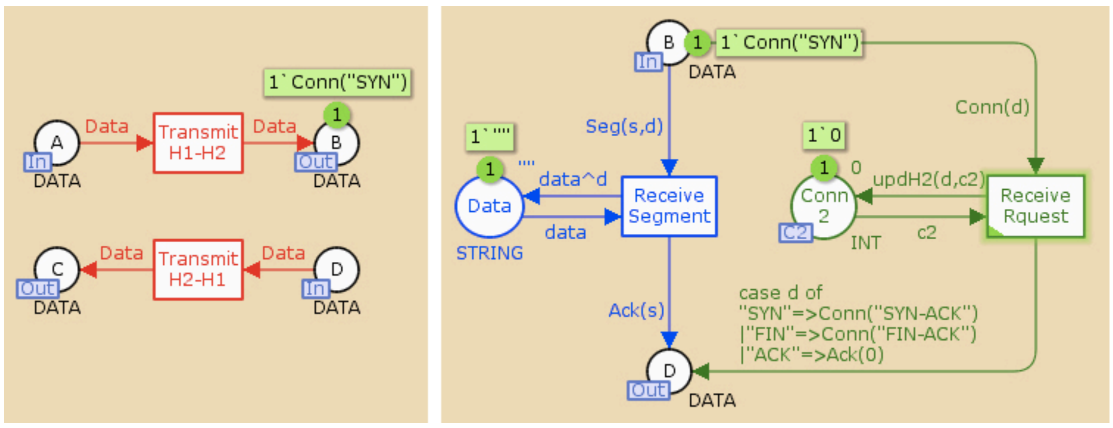


Figure 20 – The marking *M2* (Network and ETL Process modules)

In the following marking – *M2* – the token is removed from A an placed in B, enabling the transition Receive Request in the ETL Process module, since there is also an available token (with colour '0') in its other input place Conn2. This transition is responsible for processing the request sent by the OS and sending an adequate response. When it is fired, the tokens are removed from B and Conn2 and used to update the connection status of the ETL process through the output arc expression `updH2(d,c2)`. In marking *M3* (Figure 21) the token in Conn2 now has the colour '1' and there is an additional token in D with colour 'Conn("SYN-ACK")', generated by the `Receive Request` transition's output arc expression, allowing transition `Transmit H2-H1` in the Network module to activate.

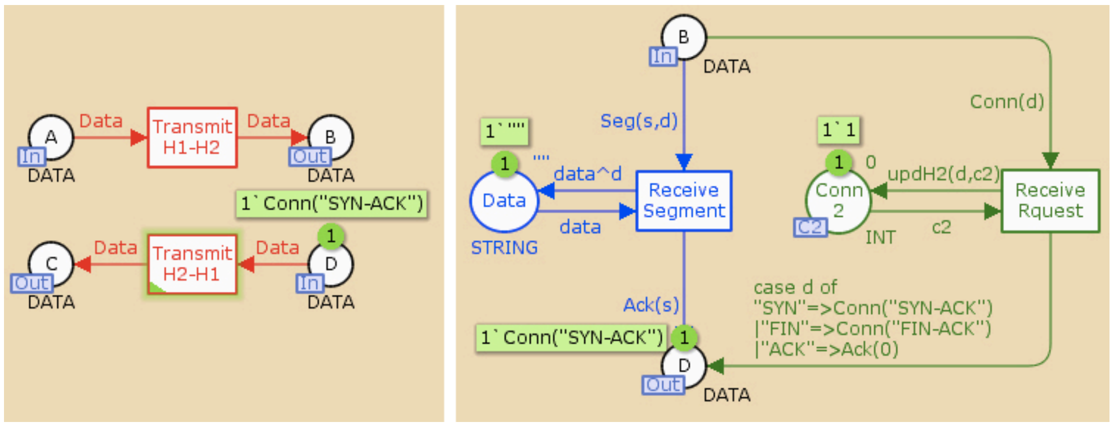


Figure 21 – The marking *M3* (Network and ETL Process modules)

Marking *M4* (Figure 22) is reached after `Transmit H2-H1` is fired. In this marking the token is passed to place C and received by the OS, enabling transition `Receive Response`.

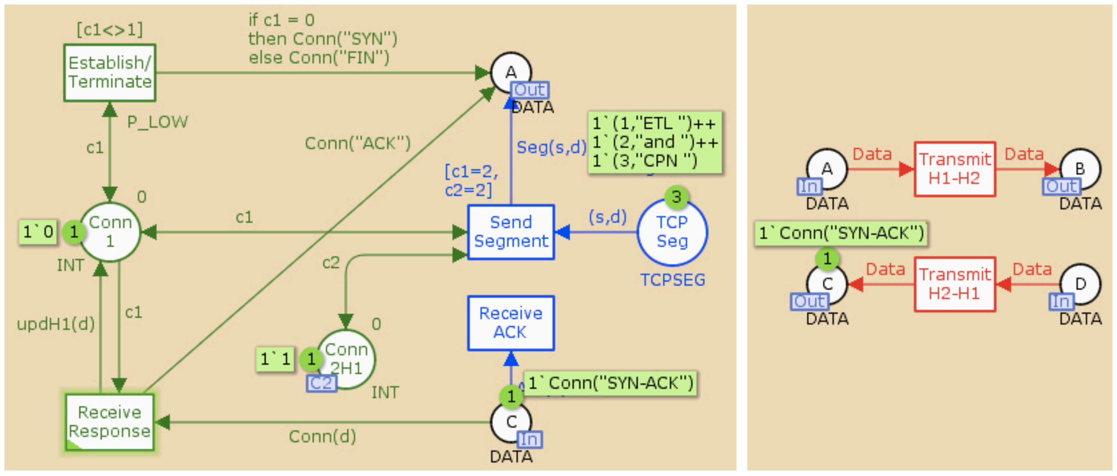


Figure 22 – The marking *M4* (Operational Source and Network modules)

When Receive Response is fired the token representing the status of the connection is removed from Conn1 and the token representing the ETL process's response is removed from C and used as a parameter in the function `updH1(d)`, one of the transition's output arc expressions, to update the status of the connection to the value '2'. The transition's second output arc is responsible for generating the acknowledgment connection packet, through the expression `Conn("ACK")`, that is placed in A and sent to the ETL process. This marking is shown in Figure 23.

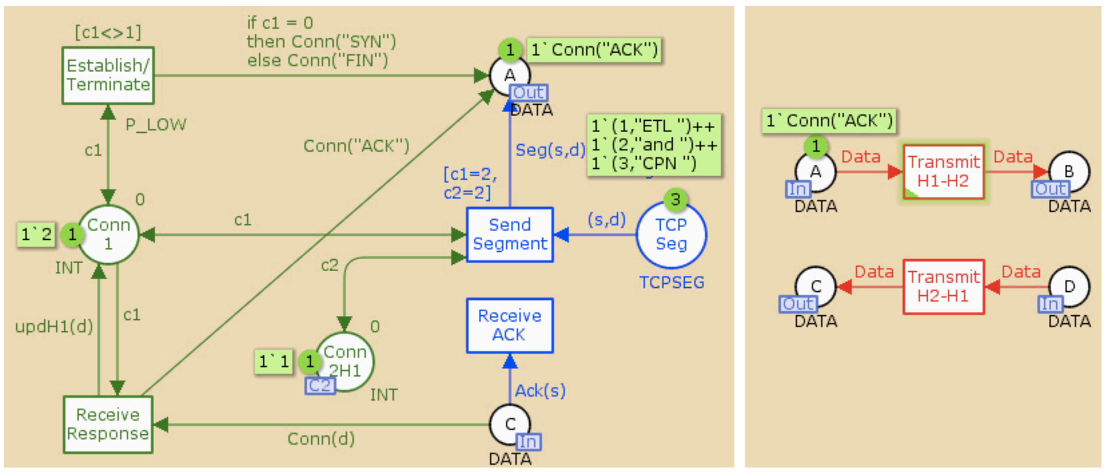


Figure 23 – The marking *M5* (Operational Source and Network modules)

The next marking – *M6* (Figure 24) – is reached after Transmit H1-H2 is fired (in marking *M5*); in this marking the ACK token is placed in B, allowing for transition Receive Request to be enabled again. The ACK indicates that the SYN-ACK packet was successfully received by the OS and the connection is established.

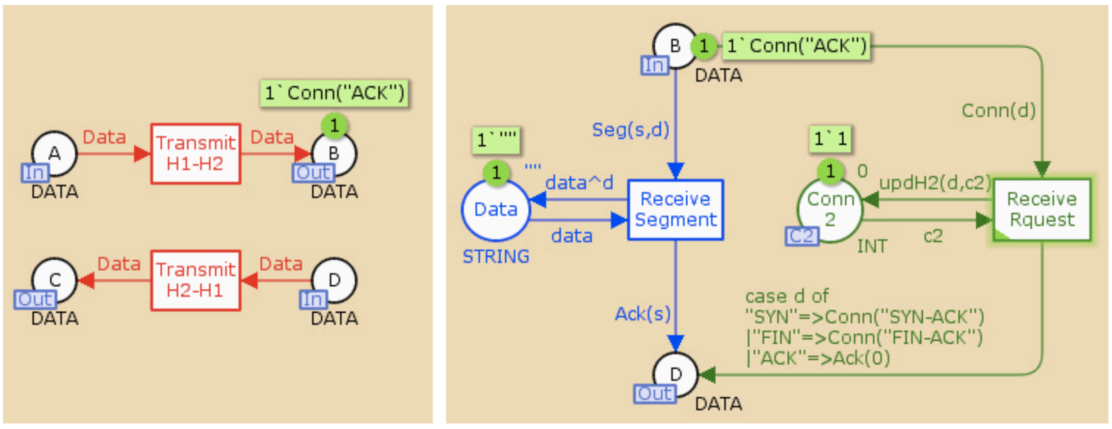


Figure 24 – The marking *M6* (Network and ETL Process modules)

In the connection establishment final marking – *M7* –, the connection status of the ETL Process is also updated to the value '2' and transition Send Segment is now enabled. The guard expression [c1=2, c2=2] was preventing this process to start until the connection was established for both hosts; the lower priority of the Establish/Terminate transition is now preventing it to activate until there exist no other enabled transitions, i.e., until all data segments are sent to the ETL process.

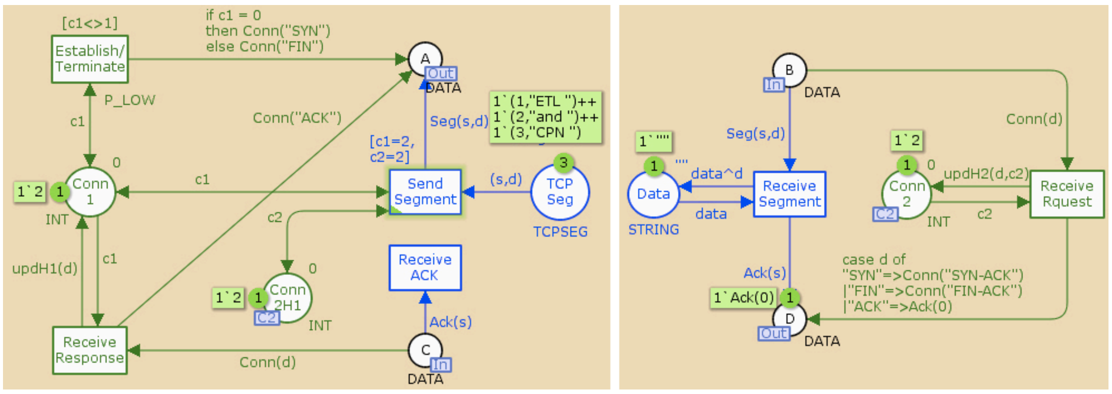


Figure 25 – The marking *M7* (Operational Source and ETL Process modules)

Once all the segments were received, i.e., all the tokens were removed from TCP Seg into Data, the Establish/Terminate transition would activate again and generate a token with colour 'Conn("FIN")' used to terminate the hosts' connection, as the status of the connection is now '2'.

Chapter 3

CPN in the Simulation of ETL Standard Tasks

3.1. ETL Processes, Some Standard Tasks

The size and the complexity of an ETL system vary depending on the implemented DWS. There are many factors that can influence the number of tasks in each ETL system, such as the number of operational sources, the quality of the extracted data – that will directly affect the complexity of the *Transform* stage, and the whole system –, and also the size and requirements of the DW where the data are to be loaded to. Nevertheless, there are still many operations that are systematically repeated in different ETL systems and even in the same system. For this reason, certain groups of operations are considered to be standard ETL tasks and have been defined in a generic way so that they can be used as a pattern or block; two of these tasks have been selected to show how the CPN modelling language can be used in the simulation of standard operations. The Surrogate Key Pipeline (SKP) is a process responsible for replacing natural keys with the corresponding surrogate keys in fact records before they are loaded into a fact table. There has to exist a SKP task for each fact table in the DW. The second standard operation, referred to as Slowly Changing Dimension (SCD), is responsible for loading dimensional records into this type of dimension, while maintaining the historic information of data. This process can be repeated many times in a single ETL system, as many as there are slowly changing dimensions in the DW.

3.2. The Surrogate Key Pipeline

The SKP process takes place during the loading stage of the records into a fact table. It's one of the last processes of this ETL stage, in which the natural keys of each record are converted into their corresponding surrogate keys (SK). There are different approaches to implement this process, for example, using mapping tables to generate and manage these attributes. However, for maximum performance, lookup tables are used – one table for each dimension. These tables are vital for the pipelining process, as their size is significantly smaller than the corresponding dimensions. This turns it possible to load and randomly access them in memory, avoiding unnecessary disk readings that probably deteriorate the performance of the system. The records in this kind of tables are called lookup records and are formed solely by each dimensional record's surrogate key, generated in a previous ETL process, and one or more corresponding natural keys. During this process, their corresponding surrogate keys replace the natural keys of each record. Thereby, there will be no natural keys in the final record but a sequence of surrogate keys, as many as the number of the existing dimensions – assuming, of course, that every single natural key must be substituted. Usually, each record should be passed though memory in a multithreaded process (Kimball and Caserta, 2004), that is, the substitution of the key of a record in memory happens simultaneously as other record substitutions that take place in different memory positions. This guarantees maximum process performance by avoiding the need to write the fact table to disk before the mapping for the next dimension begins, for each substitution process of each dimension.

As a first approach to the SKP modelling, two assumptions were made: there are four dimensions in this example, and the dimensional records come from one and the same operational source, having only one natural key. Being so, each fact table record is processed four times for the substitution of its natural keys and, therefore, it is necessary to have four lookup tables, one for each substitution stage.

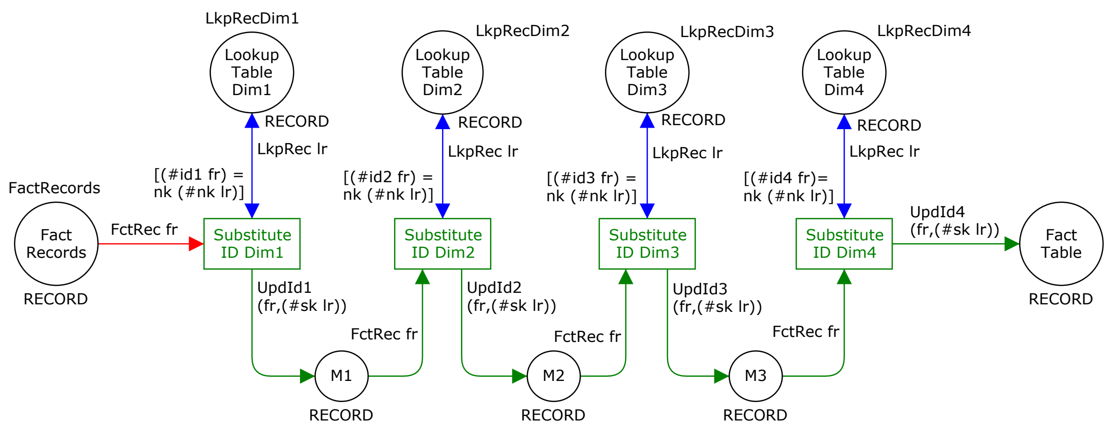


Figure 26 – The Surrogate Key Pipeline model

The CPN model in Figure 26 represents the SKP process previously described. It is composed of nine places and four transitions. Every existing place in the CPN model is of the colour set `RECORD`. This colour set is used to model a relational database record and it is defined as the union of the different kinds of records that are used in this process, namely:

```
colset RECORD = union LkpRec:LKPREC + FctRec:FCTREC;
```

Only two kinds of records are used in this process: the lookup records and the main records that are processed and loaded into the fact table (i.e., fact records). Still, this generalization makes the model more uniform and simple, as well as improves its readability. The places `Fact Records` and `Fact Table` are used to hold the tokens that represent the fact records before and after the key substitution process is applied to them, that is, before the natural keys of the initial records are replaced by the corresponding surrogate keys. Meanwhile, in each of these replacements, the places `M1` to `M3` are used to model memory positions staged by the fact records before they are actually loaded into the fact table. For this reason, all these five places receive the same type of token, with colour set `FCTREC`, representing a fact record that was defined as follows:

```
colset FCTREC = record id1:ID * id2:ID * id3:ID * id4:ID * fct:NO;
colset ID = union sk:NO + nk:ST;
colset NO = int;
colset ST = string;
```

This record has five fields, four of them corresponding to its keys, and the fifth field `fct` representing a business measure. The `id` fields with the colour set `ID`, defined as the union of an integer (representing the surrogate key) and a string (representing the natural key) are used so that the same field can assume the value of a natural key or the value of a surrogate key. Therefore, a single colour set can be used to model every state of the fact record during the substitution processes, without the need to define an independent colour set to represent a fact record in each of the memory positions, as well as the initial and final fact tables, where the colour set of the id fields varies.

The remaining four places, named respectively Lookup Table Dim(1-4), are used to model the lookup tables that correspond to the four existing dimensions in the data warehouse and they are managed during the surrogate key generation process. The colour set of these places is also `RECORD` and they receive tokens of the colour set `LKPREC`. The lookup record is a simple record with one field that represents the natural key of a record inserted in the corresponding dimension in a previous process, and another to represent the corresponding surrogate key:

```
    colset LKPREC = record sk:NO * nk:ST;
```

Finally, the four Substitute ID Dim(1-4) transitions represent the actual substitution event of the natural key of a fact record by a surrogate key. Each transition receives two tokens, one representing a fact record to be processed and the other representing a lookup record used to match the natural key to be substituted in that step with the corresponding surrogate key. The existing guards in each of the transitions are used to assure that the id field of the fact record is matched with a natural key belonging to one of the tokens representing the lookup records in the respective Lookup Table Dim place. The `UpdId(1-4)` functions in the output arcs of transitions are used to update the id fields of the fact records with the surrogate key value of the lookup record received as one of the transitions input. This function has two parameters: the actual fact record and the value of the surrogate key to be updated in the respective field; the four `UpdId` functions are identical, varying only in the field id that is being updated.

```
    fun UpdId1(fr:FCTREC, k) =
         1`FctRec(FCTREC.set_id1 fr (sk k));
```

A simulation of the model execution is now presented and described with the help of some images taken from three different markings of the simulation carryout.

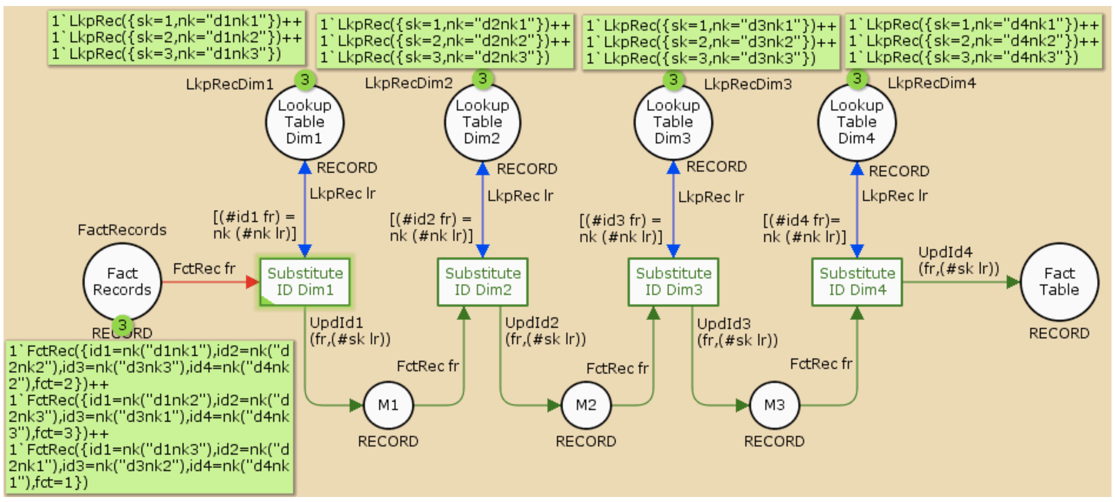

1`LkpRec({sk=1,nk="d1nk1"})++ 1`LkpRec({sk=2,nk="d1nk2"})++ 1`LkpRec({sk=3,nk="d1nk3"})
1`LkpRec({sk=1,nk="d2nk1"})++ 1`LkpRec({sk=2,nk="d2nk2"})++ 1`LkpRec({sk=3,nk="d2nk3"})
1`LkpRec({sk=1,nk="d3nk1"})++ 1`LkpRec({sk=2,nk="d3nk2"})++ 1`LkpRec({sk=3,nk="d3nk3"})
1`LkpRec({sk=1,nk="d4nk1"})++ 1`LkpRec({sk=2,nk="d4nk2"})++ 1`LkpRec({sk=3,nk="d4nk3"})
LkpRecDim1
LkpRecDim2
LkpRecDim3
LkpRecDim4
Lookup Table Dim1
Lookup Table Dim2
Lookup Table Dim3
Lookup Table Dim4
RECORD
LkpRec lr
[(#id1 fr) = nk (#nk lr)]
[(#id2 fr) = nk (#nk lr)]
[(#id3 fr) = nk (#nk lr)]
[(#id4 fr)= nk (#nk lr)]
FactRecords
Fact Records
FctRec fr
Substitute ID Dim1
Substitute ID Dim2
Substitute ID Dim3
Substitute ID Dim4
UpdId4 (fr,(#sk lr))
Fact Table
UpdId1 (fr,(#sk lr))
UpdId2 (fr,(#sk lr))
UpdId3 (fr,(#sk lr))
M1
M2
M3
1`FctRec({id1=nk("d1nk1"),id2=nk("d2nk2"),id3=nk("d3nk3"),id4=nk("d4nk2"),fct=2})++ 1`FctRec({id1=nk("d1nk2"),id2=nk("d2nk3"),id3=nk("d3nk1"),id4=nk("d4nk3"),fct=3})++ 1`FctRec({id1=nk("d1nk3"),id2=nk("d2nk1"),id3=nk("d3nk2"),id4=nk("d4nk1"),fct=1})

Figure 27 – The initial marking *M0*

The initial marking (Figure 27) has three tokens in `Fact Records`, representing the fact records with natural keys, and three tokens in the dimensions' lookup tables, representing the lookup records used in this process. For this simulation, the values of these tokens were defined through the `FactRecords` and `LkpRecDim(1-4)` constants as:

```
val FactRecords =
1`FctRec({id1=nk "d1nk3",id2=nk "d2nk1",id3=nk "d3nk2",id4=nk
"d4nk1",fct=1})++
1`FctRec({id1=nk "d1nk1",id2=nk "d2nk2",id3=nk "d3nk3",id4=nk
"d4nk2",fct=2})++
1`FctRec({id1=nk "d1nk2",id2=nk "d2nk3",id3=nk "d3nk1",id4=nk
"d4nk3",fct=3});

val LkpRecDim1 =
     1`LkpRec({sk=1,nk="d1nk1"})++
     1`LkpRec({sk=2,nk="d1nk2"})++
     1`LkpRec({sk=3,nk="d1nk3"});
val LkpRecDim2 =
     1`LkpRec({sk=1,nk="d2nk1"})++
     1`LkpRec({sk=2,nk="d2nk2"})++
     1`LkpRec({sk=3,nk="d2nk3"});
val LkpRecDim3 =
     1`LkpRec({sk=1,nk="d3nk1"})++
     1`LkpRec({sk=2,nk="d3nk2"})++
     1`LkpRec({sk=3,nk="d3nk3"});
```

```
val LkpRecDim4 =
      1`LkpRec({sk=1,nk="d4nk1"})++
      1`LkpRec({sk=2,nk="d4nk2"})++
      1`LkpRec({sk=3,nk="d4nk3"});
```

The pre-processed fact records are formed solely by strings representing the natural keys of the dimensions and the additional business measure field, while the pair with the surrogate/natural keys constitutes the lookup records. In this marking, Substitute ID Dim1 is the only transition that is enabled, as it is the only with a token in each of its input places and the guard expression evaluates to true. One token is removed from Fact Records through the arc expression `FctRec fr` and the corresponding lookup record token is removed from Lookup Table Dim1, through the arc expression `LkpRec lr`. The guard expression `[(#id1 fr) = nk (#nk lr)]` acts as a restriction, so that the value of the lookup record's `nk` field matches the value of the first `id` field of the fact record. Such a guard expression is used in each of the transitions so that, in each step, the correct fact record field is matched against the natural keys of the lookup records. When this transition is executed, the `id1` field of the token representing the first fact record is updated with the corresponding surrogate key, through the function `UpdId1(fr,(#sk lr))`, and passed to M1; this record is then ready for another key substitution, corresponding to the natural key of the second dimension (Figure 28).

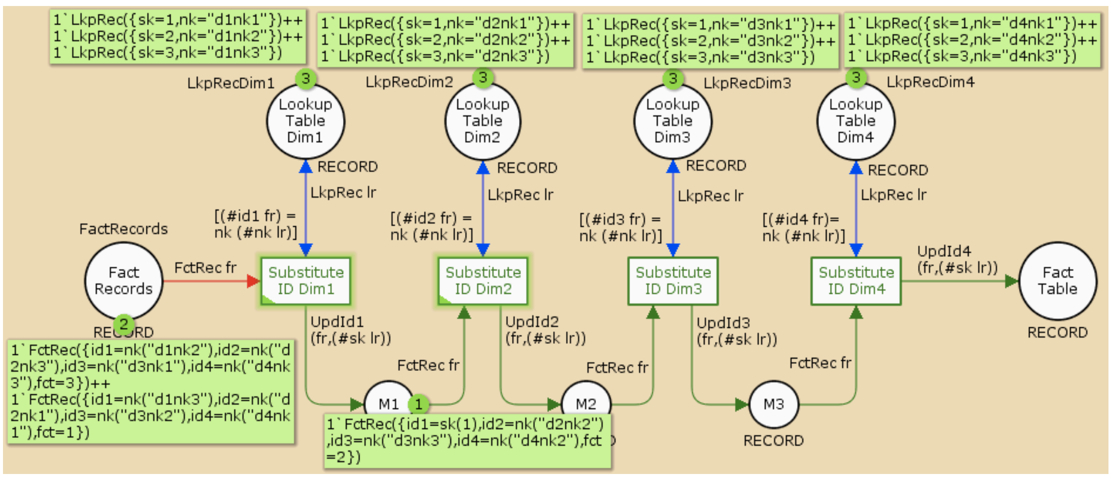

1`LkpRec({sk=1,nk="d1nk1"})++
1`LkpRec({sk=2,nk="d1nk2"})++
1`LkpRec({sk=3,nk="d1nk3"})
1`LkpRec({sk=1,nk="d2nk1"})++
1`LkpRec({sk=2,nk="d2nk2"})++
1`LkpRec({sk=3,nk="d2nk3"})
1`LkpRec({sk=1,nk="d3nk1"})++
1`LkpRec({sk=2,nk="d3nk2"})++
1`LkpRec({sk=3,nk="d3nk3"})
1`LkpRec({sk=1,nk="d4nk1"})++
1`LkpRec({sk=2,nk="d4nk2"})++
1`LkpRec({sk=3,nk="d4nk3"})
LkpRecDim1
3
Lookup Table Dim1
LkpRecDim2
3
Lookup Table Dim2
3
LkpRecDim3
Lookup Table Dim3
3
LkpRecDim4
Lookup Table Dim4
RECORD
LkpRec lr
[(#id1 fr) = nk (#nk lr)]
RECORD
LkpRec lr
[(#id2 fr) = nk (#nk lr)]
RECORD
LkpRec lr
[(#id3 fr) = nk (#nk lr)]
RECORD
LkpRec lr
[(#id4 fr)= nk (#nk lr)]
FactRecords
Fact Records
FctRec fr
Substitute ID Dim1
Substitute ID Dim2
Substitute ID Dim3
Substitute ID Dim4
UpdId4 (fr,(#sk lr))
Fact Table
RECORD
2
RECORD
1`FctRec({id1=nk("d1nk2"),id2=nk("d2nk3"),id3=nk("d3nk1"),id4=nk("d4nk3"),fct=3})++
1`FctRec({id1=nk("d1nk3"),id2=nk("d2nk1"),id3=nk("d3nk2"),id4=nk("d4nk1"),fct=1})
UpdId1 (fr,(#sk lr))
FctRec fr
UpdId2 (fr,(#sk lr))
FctRec fr
UpdId3 (fr,(#sk lr))
FctRec fr
M1
1
M2
M3
1`FctRec({id1=sk(1),id2=nk("d2nk2"),id3=nk("d3nk3"),id4=nk("d4nk2"),fct=2})
RECORD

Figure 28 – The marking *M1*

In the marking *M1* (Figure 3), reached after firing the Substitute ID Dim1 transition, there are now two enabled transitions – Substitute ID Dim1 and Substitute ID Dim2 – as it is possible to "remove" tokens from their input places and the guard expressions evaluate to true, allowing any of them to be fired. If Substitute ID Dim1 is fired, a new token representing a fresh pre-processed fact record is removed from Fact Records, its first natural key is replaced by the corresponding surrogate key, and then passed to M1. If the Substitute ID Dim2 is fired, the second natural key of the token, representing the first fact record, is replaced by the corresponding surrogate key, and the transition Substitute ID Dim 3 is enabled, as the token is passed to M2. After a few markings, as the tokens are processed, all transitions become enabled, allowing for several records to be processed in memory at the same time, before they are loaded into the Fact Table.

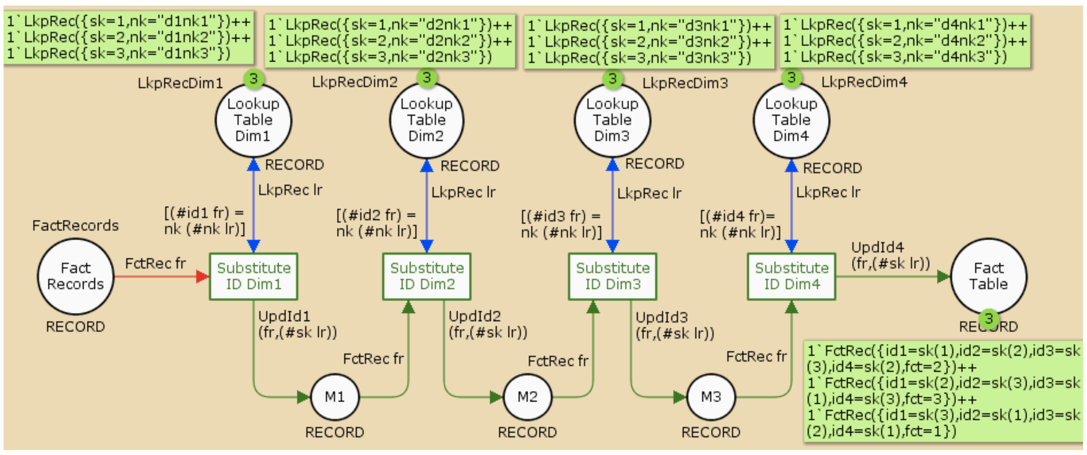

1`LkpRec({sk=1,nk="d1nk1"})++
1`LkpRec({sk=2,nk="d1nk2"})++
1`LkpRec({sk=3,nk="d1nk3"})
1`LkpRec({sk=1,nk="d2nk1"})++
1`LkpRec({sk=2,nk="d2nk2"})++
1`LkpRec({sk=3,nk="d2nk3"})
1`LkpRec({sk=1,nk="d3nk1"})++
1`LkpRec({sk=2,nk="d3nk2"})++
1`LkpRec({sk=3,nk="d3nk3"})
1`LkpRec({sk=1,nk="d4nk1"})++
1`LkpRec({sk=2,nk="d4nk2"})++
1`LkpRec({sk=3,nk="d4nk3"})
LkpRecDim1
LkpRecDim2
LkpRecDim3
LkpRecDim4
Lookup Table Dim1
Lookup Table Dim2
Lookup Table Dim3
Lookup Table Dim4
RECORD
LkpRec lr
[(#id1 fr) = nk (#nk lr)]
[(#id2 fr) = nk (#nk lr)]
[(#id3 fr) = nk (#nk lr)]
[(#id4 fr)= nk (#nk lr)]
FactRecords
Fact Records
FctRec fr
Substitute ID Dim1
Substitute ID Dim2
Substitute ID Dim3
Substitute ID Dim4
UpdId4 (fr,(#sk lr))
Fact Table
UpdId1 (fr,(#sk lr))
UpdId2 (fr,(#sk lr))
UpdId3 (fr,(#sk lr))
M1
M2
M3
1`FctRec({id1=sk(1),id2=sk(2),id3=sk(3),id4=sk(2),fct=2})++
1`FctRec({id1=sk(2),id2=sk(3),id3=sk(1),id4=sk(3),fct=3})++
1`FctRec({id1=sk(3),id2=sk(1),id3=sk(2),id4=sk(1),fct=1})

Figure 29 – The final marking *M12*

In the final marking (Figure 29) there are three tokens in Fact Table, representing the final fact records processed through each of the substitution events and the same three tokens in each of the lookup tables. All the natural key of the fact records have been replaced by the corresponding surrogate key values, originating fact records that are ready to be loaded into the corresponding table in the data warehouse.

3.3. Slowly Changing Dimensions

One of the key characteristics of a DWS is time-variance, i.e., data changes that occur over time are captured in the DW. This means that the data of the DW is not updatable in the same way as most transaction systems, where records are simply updated to their most recent values. Instead, all historical data are stored in the DW with the associated timestamp to preserve the modifications of the attributes of dimensional records over time. As an example, consider a small Data Mart with four dimensions (Date, Customer, Product and Store), and the grain of the fact table is the amount of a certain Product sold each Day to a Customer in a Store. Now let us consider that a regular customer at a certain store from his/her hometown changes his/her address to a different city. If the historical data is not stored, and the address is simply updated to its most recent value, the information on this Data Mart will not reflect what truly happened. In fact, it will seem as if the customer never lived in the previous city, but always went there for shopping. For this reason it is important to select the dimensions with attributes that may change over time and take special care when they are physically loaded into the DW.

The dimensions with data that can slowly change over time are called Slowly Changing Dimensions (SCD). Usually, they are managed through an ETL process that takes place during the loading stage of the dimensions, to preserve historical information whenever modifications occur in the operational sources. There are several conventional approaches to implement a SCD process. They are referred in the literature as Type 1 through 6. The process described here shall be referred to as Slowly Changing Dimensions with maintenance of historic records (SCD-H), as some of its characteristics cannot be found in the existing conventional SCD types, even though it is influenced by concepts of the Type 4 SCD methodology.

The SCD-H process uses a dimension historic table – with a N:1 cardinality to the corresponding Dimension table – to store all the historic changes of its records, while preserving both the structure of the main dimension and the surrogate keys assigned to the records in the moment of their insertion. A historic change happens when a record belonging to a SCD is updated in the source system(s) and also when a new record is inserted or deleted; these events must be captured through specific audit tables. Each SCD must have a corresponding audit table, where each tuple is composed, frequently, of the ID of the source system, the timestamp of the operation, the type of the captured operation (i.e., insert, update, delete) and all the attributes of the modified record. The first stage of the SCD-H consists on the analysis of this kind of table so that the corresponding dimension can be processed accordingly with the operation type of each tuple.

A new dimensional record has to be created and inserted in the target SCD table for each of the audit records that represent new insertions on the operational source(s). For this to happen, the SKP process needs to take place in order to assign the correct SK to the incoming record and to manage the mapping of this key with the natural key(s) of the record through lookup tables (Section 3.2.4). A dimensional record is composed of its newly assigned SK, its natural key(s), the timestamp of the insertion (in the operational source(s)), the record status (i.e., an attribute for indicating if the record is *Active* or *Inactive*) and the remaining dimensional attributes. The status needs to be set as *Active* for every newly inserted record.

No SCD records should be deleted from a DW even if the corresponding information is permanently removed from the operational source(s). Therefore, if *delete* operations are captured in the audit table, the status of the corresponding SCD records need to be updated to *Inactive* so that they can still be used for querying. The lookup table of the dimension is used to retrieve the SK that corresponds to the natural key(s) of the Audit record in order to update the correct SCD record.

The audit records with the *Update* operation type contain the information that needs to be updated in the DW while storing the old values of the attributes of the corresponding SCD record. In this process the lookup table is again used to fetch the correct SCD record. Before its attributes are updated, this record needs to be inserted in the historic table of the Dimension in order to preserve its former values, which include the timestamp of the last operation (e.g. the record's insertion), the SK of the record and the dimensional attributes. The SCD record can then be updated with the new values of the dimensional attributes and also the timestamp of the update in the source system(s), without having to modify the SK assigned to it in the moment of its insertion.

As it has been mentioned above, this process occurs during the Loading stage of the ETL, after the data has been cleansed and transformed accordingly with the target SCD. Even so, it is still possible that some records are unfit to be loaded in the SCD. Given the complexity of the Transformation stage of the ETL, it is not uncommon to find records with minor flaws or bugs that were left unnoticed. For this reason, the audit data used in the SCD process should be verified one last time before the corresponding dimension is updated with it. During this verification, the records that fail to succeed in a data quality test are moved into a quarantine table in order to be corrected by the DW administrator. It is important to register these data verification events so that the errors in the quarantined records can be easily addressed by the administrator. Being so, a SDC error log needs to be updated each time a record is quarantined with the timestamp and description of the error, followed by the quarantined record. In addition to the error log, a more general log journal is also commonly maintained during this ETL process. This type of log is updated when a record is inserted in, deleted in or updated from the SCD, as well as when a record is moved into quarantine. The general behaviour of the process can be consulted in the log journal table where each record has information on the SCD name (as this process can occur simultaneously for different dimensions), the operation registered in the log journal (inserted record, updated record, deleted record, quarantined record), the timestamp of the operation and the key of the record.

3.3.1. The Prime Module (SCD-H)

The CPN model for the SCD-H ETL process is hierarchical and thus, organized as a set of different modules. The module standing on top of this hierarchy (i.e., the *prime module*) represents the entire SCD-H process and allows for a more abstract view, where some details are hidden and represented solely in low-level modules. As referred previously, this ETL process can be divided in three main operations – Insert, Update and Delete – so the CPN model has been intuitively separated into these three main sub-modules. The *prime* module is presented in Figure 30.

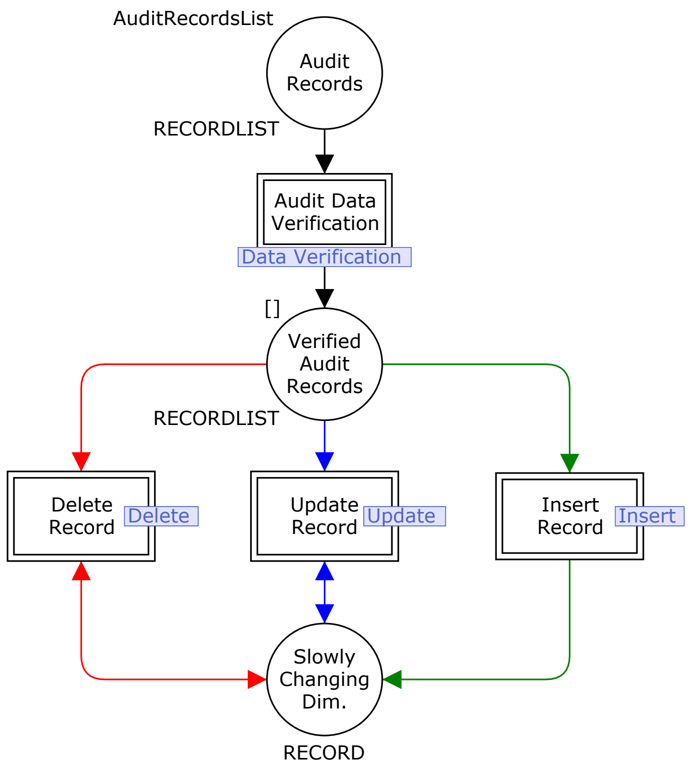


Figure 30 – The SCD-H prime module

The SCD-H module contains four substitution transitions and three places. The three sub-modules mentioned above are related respectively to the Insert Record, Update Record and Delete

Record substitution transitions. There is one additional sub-module – Data Verification – related to the Audit Data Verification substitution transition. The *input socket* of Audit Data Verification is the place Audit Records with colour set RECORDLIST, while the place Verified Audit Records, with the same colour set, is both the *output socket* of Audit Data Verification and the *input socket* of the Insert Record, Update Record and Delete Record substitution transitions. The *output socket* of these transitions is the Slowly Changing Dim place with colour set RECORD.

The colour set RECORDLIST, defined as `colset RECORDLIST = list RECORD`, is used to model the verified and pre-verified audit records, so that they can be processed as a FIFO list. This choice was made assuming that the records in the audit tables are ordered by the date of the captured operation in the operational sources and should be processed accordingly. Being so, the place Audit Records holds a list of audit records, ordered by the timestamp of the operation, ready for verification. The place Verified Audit Records holds the list of audit records that are fit to be processed according to its operation type in one of the succeeding substitution transitions. The colour set RECORD is used to model a record from a relational database and it is defined to be the union of the different types of records used in the SCDH process, as follows:

```
colset RECORD = union
     AudRec:AUDREC + DimRec:DIMREC + HstRec:HSTREC +
     CnfRec:CNFREC + LkpRec:LKPREC + LogRec:LOGREC +
     ErrLog:ERRLOG;
```

Aside from Audit Records and Verified Audit Records, the remaining places in the model that represent relational tables (e.g., Slowly Changing Dim) or individual records have the RECORD colour set. This makes the model more uniform, but special care must be taken when adding a record token to a place. For example, the place Slowly Changing Dim should only have tokens of the DIMREC colour set, as this place represents the final dimension and can only be loaded with the dimensional records. The type of record added to a place with the colour set RECORD must then be enforced through the arc expressions that lead to the place. Nevertheless, this generalisation improves the readability of the model and also the reusability of fragments (i.e. *groups*) of the modules. The colour sets that represent the existing record types are described in the following sections.

3.3.2. The Data Verification Module

The Data Verification module is used to model the data verification process applied to the records of the audit table. This module contains a single transition and six places (Figure 31).

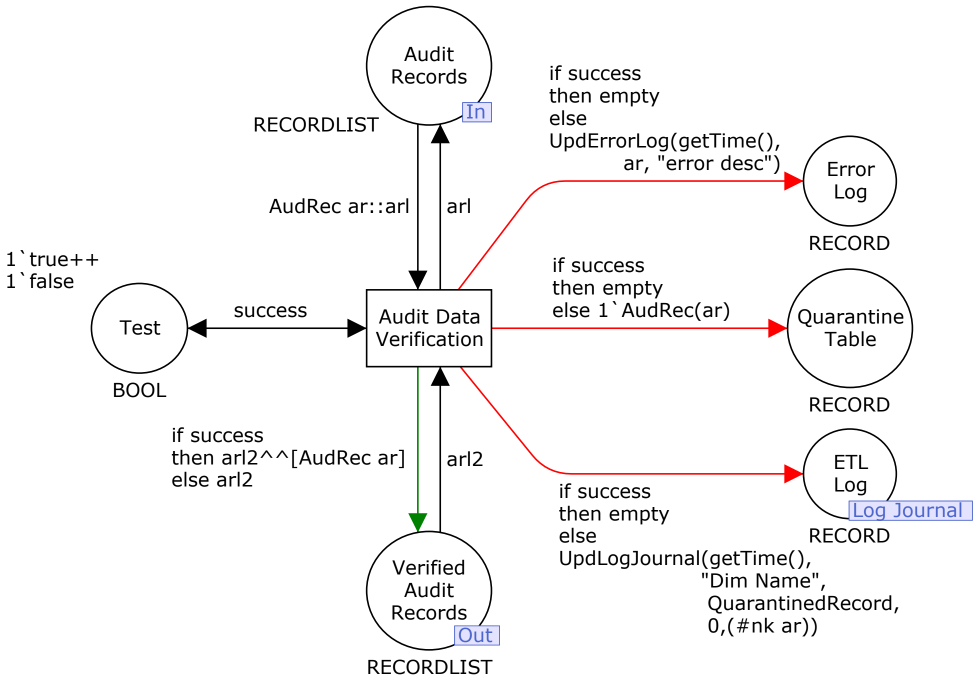

Audit Records
In
RECORDLIST
AudRec ar::arl
arl
if success then empty else UpdErrorLog(getTime(), ar, "error desc")
Error Log
RECORD
1`true++ 1`false
Test
BOOL
success
Audit Data Verification
if success then empty else 1`AudRec(ar)
Quarantine Table
RECORD
if success then arl2^^[AudRec ar] else arl2
arl2
ETL Log
Log Journal
RECORD
if success then empty else UpdLogJournal(getTime(), "Dim Name", QuarantinedRecord, 0,(#nk ar))
Verified Audit Records
Out
RECORDLIST

Figure 31 – The Data Verification module

The place Audit Records is the *input port* of this module and the *output port* is the place Verified Audit Records. These two places have already been characterised and together they constitute the interface through which this module communicates with the *prime module.* Both places hold a token representing a list of audit records, making this list the input (in a pre-verification state) and the output (after the verification) of this module. An audit record is a record with five fields, each one used to model a column from the audit table, and it is defined as follows:

```
colset AUDREC = record
      src:ST * dtm:ST * opr:OPERATION * nk:ST * atb:ST;
colset ST = string;
colset OPERATION = with Insert | Update | Delete;
```

The src, dtm, nk and atb fields are strings that correspond, respectively, to the name of the source system, the timestamp of the captured operation, the natural key of the record and the modified attribute. To make the model simpler, a single field (atb) is used to represent all the attributes that need to be preserved in the DW. The opr field (of the enumeration colour set OPERATION) is used to represent the three different operation types that can be captured in the audit table. The place Quarantine Table is used to model a table with the same name, where the records that fail the verification test rest until the DW administrator corrects the errors that lead to the failure. This place receives tokens of the colour set AUDREC but the use of a list is not necessary in this case.

As described earlier, it is necessary to maintain two kinds of log tables during the verification process and to model them two distinct places are used. The place Error Log represents the table where the data verification events are registered, which occurs when the data test is unsuccessful, through Error Log Records. This kind of record has a similar structure to the Audit Record but with two extra string fields; the errdtm field represents the timestamp of the verification process and the err field represents the error name or description. An Error Log Record is defined as:

```
colset ERRLOG = record src:ST * errdtm:ST * dtm:ST * opr:OPERATION *
     nk:ST * atb: ST * err:ST;
```

The ETL log journal information needs to be registered when a record is quarantined, but also when a record is inserted, updated or deleted during the SCD-H process; therefore, the *fusion place* ETL Log is used to model this kind of log table. The Log Journal *fusion set* allows its members to share their Log Record tokens as if they were a single place, even if distributed over different modules.

```
colset LOGREC = record
     dtm:ST * dnm:ST * opr:LOGOP * sk:NO * nk:ST;
colset LOGOP = with
     InsertedRecord | InactiveRecord |
     UpdatedRecord  | QuarantinedRecord;
colset NO = int;
```

The colour set LOGREC, with five fields, is used to represent a Log Record; the dtm is relative to the timestamp of the operation occurrence, the dnm is a string used to identify the SCD that is being processed, and the opr field of the colour set LOGOP is the enumeration set of operations that can be registered. The sk and nk fields represent the surrogate and natural keys of the processed record, respectively. In this stage of the process, no surrogate keys have been assigned to the audit records, so the natural keys are used to identify the records of the Log Journal table that have been quarantined. Finally, the place Test, with colour set BOOL, is used to simulate a test on each of the audit records. Even though no real testing is done to the attributes of the record, the success or failure of such test is simulated through the variable success that for each marking of the model randomly assumes the values of the tokens in this place (i.e., 'true' or 'false').

The single event that can take place in this module (i.e., the actual verification of data) is modelled by the transition Audit Data Verification that receives three tokens. The first, representing one audit record, is removed from the head of the list in Audit Records through the arc expression AudRec ar :: arl, where the variables ar and arl are declared as:

```
var ar : AUDREC;
var arl, arl2: RECORDLIST;
```

The second token is success, a Boolean variable that determines the success or failure of this test. This token is returned to Test so it can assume a new colour and simulate the same test on the next record. The last token represents the list of verified audit records and is removed from Verified Audit Records through the arc expression arl2. If the data test is successful (i.e., if success is 'true'), the verified audit record is inserted in this list through the expression in green arc; on the other hand, if the test is unsuccessful (i.e., if success is 'false') the list is not updated. In this case, the data flow is represented by the three red output arcs connected to Quarantine Table, Error Log and ETL Log. The audit record that fails the verification test is put in the Quarantine Table and it is also used to generate new Log records that are passed to the places Error Log and ETL Log respectively. The functions UpdErrorLog(getTime(), ar, "error desc") and UpdLogJournal(getTime(), "Dim Name", QuarantinedRecord, 0, (#nk ar)) are used as arc expressions to generate tokens of the ERRLOG and LOGREC colour set; the getTime function in the first parameter is used to generate a string representation of the current date, formatted in order to represent the timestamp of this operation. These functions are implemented as follows:

```
fun UpdErrLog(date, ar:AUDREC,error) =
     1`ErrLog{
          src = (#src ar),
          errdtm = date,
          dtm = (#dtm ar),
          opr = (#opr ar),
          nk = (#nk ar),
          atb = (#atb ar),
          err = error};

fun UpdLogJournal(date,dn,operation:LOGOP,sk:NO,nk:ST) =
     1`LogRec{
          dtm = date,
          dnm = dn
          opr = operation,
          sk = sk,
          nk = nk};

fun getTime()=
     Date.fmt "%d/%m/%%Y %H:%M:%S"
     (Date.fromTimeLocal (Time.now()));
```

After this data verification process, the socket place Verified Audit Records (Figures 30 and 31) hold the list of records that are actually used to update the SCD through the three consequent substitution transitions.

3.3.3. The Insert Module

This module represents the operations that are necessary to insert a new record into the SCD. From the selection of records, to the assignment of a fresh surrogate key, finishing with the loading process of the resulting dimensional record, all these processes are part of the Insert module (Figure 32).

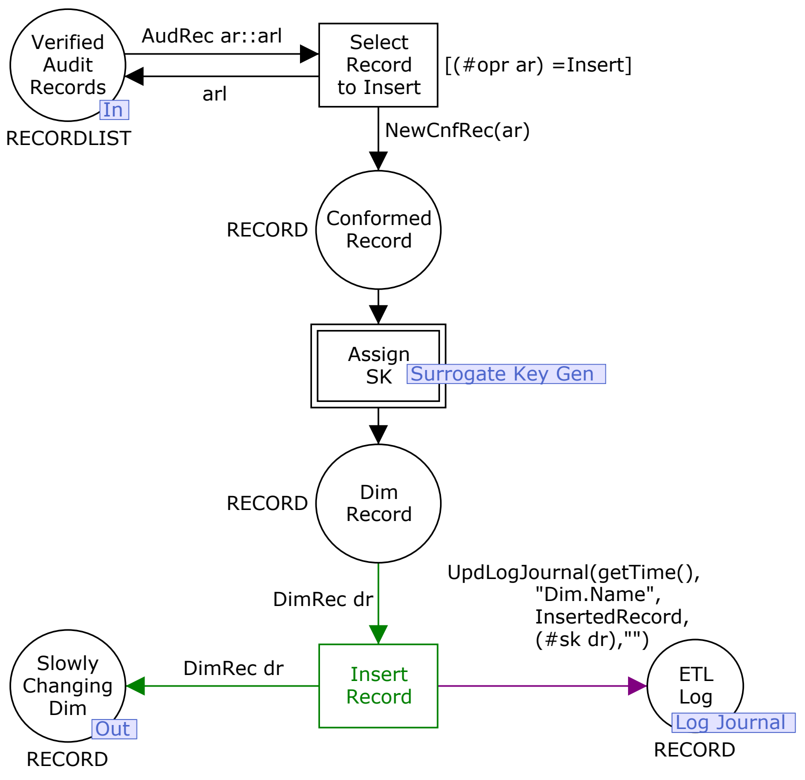


Figure 32 – The Insert module

The place Verified Audit Records is now the *input port* of this module while the *output port* is the place that represents the final SCD – Slowly Changing Dim. The places Conformed Record and Dim Record are respectively the input and output sockets of the substitution transition Assign SK that models the Surrogate Key Generation process. The fusion place ETL Log is used again to represent the ETL Log Journal. The selection of the records to be inserted is modelled by the Select Record to Insert transition that is enabled if the operation type of the first element of the list is 'Insert'; in this case the audit record is removed from the list and passed to the place Conformed Record. A conformed record does not have information on the type of operation captured in the operational source, nor should the id of the source system be stored in the SCD; for these reasons the audit record is restructured through the function `NewCnfRec` in the arc expression, which is also used to mark the status of the resulting record status as active. The conformed record colour set and the `NewCnfRec` function are implemented a follows:

```
colset CNFREC = record
     nk:ST * sta:STATUS * dtm: ST * atb:ST;
```

```
colset STATUS = with A | I;

fun NewCnfRec (ar:AUDREC) =
     1`CnfRec{
          nk=(#nk ar),
          sta=A,
          dtm=(#dtm ar),
          atb=(#atb ar)};
```

To generate a new conformed record the `sta` field, representing the status of the record, is updated to the value 'A' (i.e., active), while the `src` and `opr` fields belonging to the audit record are discarded. The values of the remaining fields are maintained. The record can now undertake its last transformation before it can be loaded into the DW. The assignment of a SK to the conformed records results in the final dimensional records modelled by tokens of the colour set `DIMREC` in the *output socket* Dim Record. A dimensional record is a conformed record with an extra field – `sk` – representing the newly assigned SK:

```
colset DIMREC = record sk:NO * nk:ST * sta:STATUS * dtm:ST * atb:ST;
```

The actual insertion of the record can now be performed and this is represented by the Insert Record transition, which is also the simplest in this module. A token representing the dimensional record is removed from the place Dim Record and passed to the output port place Slowly Changing Dim through the green arcs, while updating the Log Journal at the same time. The expression in the green arcs is `DimRec dr`, where the variable `dr` is declared as:

```
var dr : DIMREC;
```

At this stage, note that the Log Journal is updated with the SK assigned to the dimensional record instead of its natural key(s), in contrast with what happened in the Data Verification module.

3.3.4. The Surrogate Key Generation Module

Before a new conformed record can be loaded into the target SCD, a fresh SK must be created and assigned to it. The identification of the dimensional records through surrogate keys is highly recommended and brings numerous advantages in the development and management of the DW. The Surrogate Key Generation module (shown in Figure 33) represents this simple, yet very important ETL process.

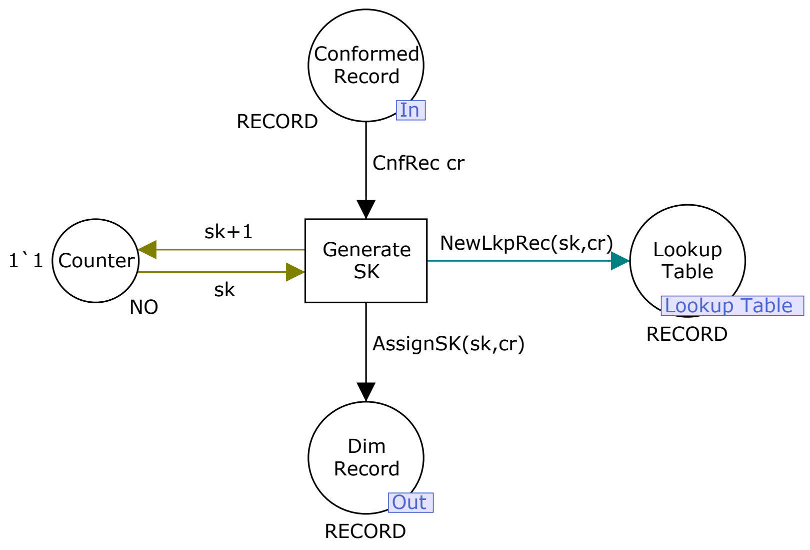


Figure 33 – The Surrogate Key Gen module - Assign SK

The implemented net is composed by four places and a single transition. The places Conformed Record and Dim Record, described in the Insert module, represent the record before and after a SK is assigned to it, meaning that a record is only considered to be a dimensional record (i.e., with the correct structure to be inserted in the dimension) after the SK generation process. The place Counter with colour set NO represents the SK counter and has a single token (i.e., an integer) that represents the SK to be assigned to the next record. The last place, Lookup Table, receives tokens that represent the lookup records used to map the newly assigned SK with the corresponding natural key. The lookup record is a simple record with one field to represent the natural key of the conformed record and another to represent the corresponding SK. It is implemented as follows:

```
colset LKPREC = record sk:NO * nk:ST;
```

The Generate SK transition removes one token representing the conformed record from Conformed Record and the existing token in the Counter, representing the SK value to be assigned, through the arc expressions `CnfRec cr` and `sk` where `cr` and `sk` are declared as:

```
var cr : CNFREC;
var sk : NO;
```

When the transition is fired, the SK is assigned to the record and passed to the place Dim Record through the arc expression `AssignSK(sk,cr)`. This function creates a new dimensional record with the information of the conformed record plus the value of the SK:

```
fun AssignSK (sk,cr:CNFREC) =
     1`DimRec{
          sk=sk,
          nk=(#nk cr),
          sta=(#sta cr),
          dtm=(#dtm cr),
          atb=(#atb cr)};
```

At the same time, a new token is created and placed in the Lookup Table place, through the expression `NewLkpRec(sk,cr)` in the remaining output arc. This function uses the value of the newly assigned SK and the value of the natural key of the conformed record to update the Lookup Table place with a new lookup record, and is implemented as:

```
fun NewLkpRec (sk,cr:CNFREC) =
     1`LkpRec{
          sk=sk,
          nk=(#nk cr)};
```

For the process to be completed, the SK needs to be incremented and placed in the SK counter so that it can be used to identify the next record, which is accomplished with the expression `sk + 1` in the input arc of the place Counter. If no tokens exist in the place Conformed Record then the SK value is preserved in this place until the transition is activated.

3.3.5. The Update Module

The Update module is the most complex module in the SCD-H process, where the dimensional records to be updated are preserved in a special historic dimension before its attributes are modified in the main SCD. The CPN model for this process is shown in Figure 34.

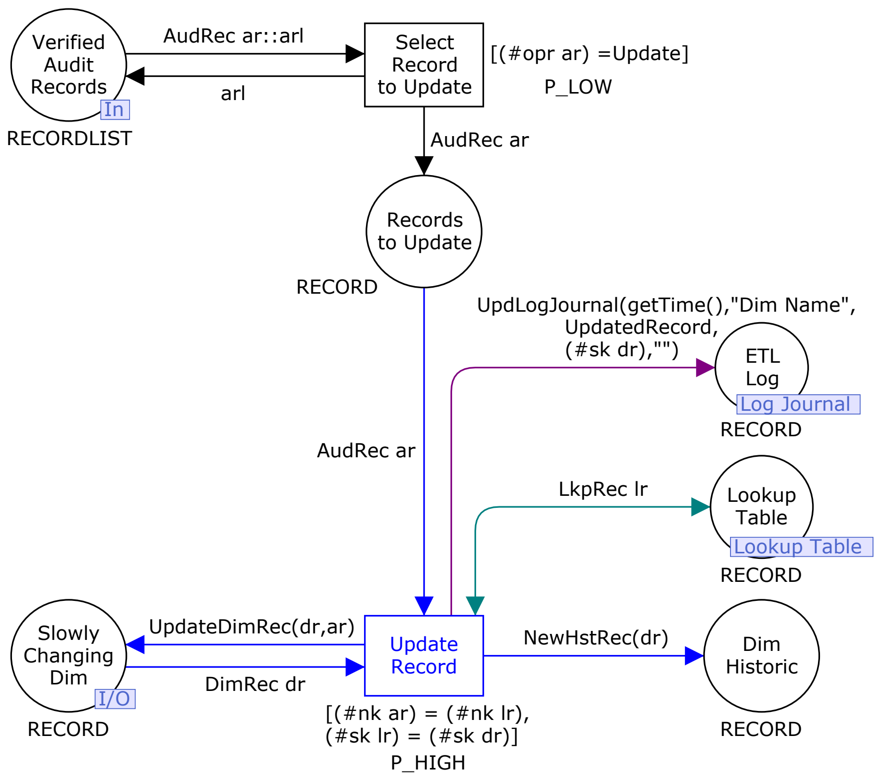


Figure 34 – The Update module

The selection of records from the list of audit records is handled in a similar way to the Insert module, through the Select Record to Update transition. The difference resides in the guard expression, indicating that the operation type of the audit record must equal the Update value. The extracted tokens then stay in the place Records to Update, representing the audit records that are used for updating the dimensional attributes of the corresponding dimensional records. It is important to mention that each module activates accordingly with the operation type of the head element of the list, keeping its processing sequential. The places Verified Audit Records, Slowly Changing Dim and ETL Log are analogous to the ones in the Insert module but, in this module, the place Slowly Changing Dim is both an *input* and *output port* since a token is also removed from it for the transition to activate.

The place Dim Historic represents a relational table that stores the historic of the dimension to be updated and receives tokens of the type `HSTREC`, representing the historical records. The historical record can be seen as a dimensional record stored in another table so their structure is very similar. The status field is no longer necessary because, from the moment a record becomes historic, it also becomes automatically inactive. The historical record is implemented as follows:

```
colset HSTREC = record sk:NO * nk:ST * dtm:ST * atb:ST;
```

The *fusion* place Lookup Table, member of the same *fusion set*, managed in the Surrogate Key Generation process represents the lookup table of the dimension. The use of this kind table is of great importance in this stage of the SCD-H process, allowing the correct dimensional record to be quickly found and retrieved from the SCD for further processing. The main operation of this module, representing the Update of the attributes of the dimensional record, while storing their old values in the historic table, is modelled by the Update Record transition. To enable this transition, it receives an audit record, a lookup record and a dimensional record. The audit record is removed from the Records to Update place, through the arc expression `AudRec ar`, only if there is a lookup record with the same natural key value, removed from the place Lookup Table through the arc expression `LkpRec lr`, where `lr` is defined as:

```
var lr : LKPREC;
```

If such record exists in the Lookup Table, it means that the corresponding dimensional record has already been inserted in the SCD and can be removed from the Slowly Changing Dim place into the transition through the arc expression `DimRec dr`. The transition may not be enabled if, for a given audit record, there isn't a matching lookup record, meaning that the corresponding dimensional record does not yet exist in the SCD (e.g., it may have been quarantined and therefore can't be updated); in this case the records remain in that place until the corresponding dimensional record is inserted in the SCD. This behaviour is modelled by the existing guard expression.

When there is a correspondence between these three types of records the transition becomes enabled and the update process begins. The lookup record is merely used to find the dimensional record that corresponds to the audit record and is returned to the Lookup Table place; the received information of the dimensional records is used to generate a new corresponding historical record and the audit record is used to update the attributes of the dimensional records. The dimension historic is updated through the input arc of the Dim Historic place with the expression `NewHstRec(dr)`. This function uses the dimensional record as the parameter and produces a fresh historical record from it, preserving the old values of the record:

```
fun NewHstRec (dr:DIMREC) =
     1`HstRec{
          sk=(#sk dr),
          nk=(#nk dr),
          dtm=(#dtm dr),
          atb=(#atb dr)};
```

The actual update of this dimensional record in the SCD is represented by the `UpdateDimRec(ar,dr)` expression in the input arc of the place Slowly Changing Dim:

```
fun UpdateDimRec(dr:DIMREC,ar:AUDREC) =
     1`DimRec{
          sk=(#sk dr),
          nk=(#nk dr),
          sta=(#sta dr),
          dtm=(#dtm ar),
          atb=(#atb ar)};
```

This function takes the audit record and the dimensional record as parameters and generates a dimensional record with the timestamp of the audit record (i.e., the timestamp of the update in the source system) and the new attribute values. The remaining attributes are left unchanged, i.e., the surrogate key, the natural key and the status of the dimensional record that remains active. To conclude the process, a new log record is generated and used to register the update operation of this dimensional record in the Log Journal.

3.3.6. The Delete Module

The final piece of the SCD-H model is the Delete module (Figure 35), that is responsible for marking a dimensional record as inactive instead of deleting it permanently from the SCD, and hence, from the DW. All the places in this module have already been described in the former modules. The *input/output* ports, as well as the audit records selection process, are the same as in the Update module, differing only in the guard expression of the Select Record to Delete transition which has now the 'Delete' value.

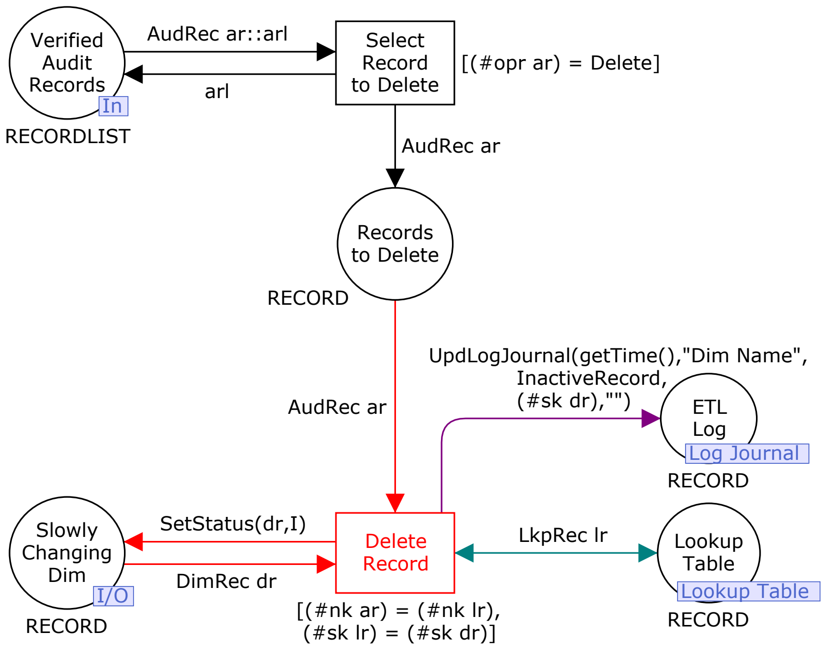


Figure 35 – The Delete module

The Lookup Table and the guard expression in the Delete Record transition are once again used to find the dimensional record that corresponds to the audit record removed from the place Records to Delete. The token representing the dimensional record to be deleted is removed from the place Slowly Changing Dim and passed to the transition so it can be marked as inactive and returned to the same place. The expression `SetStatus(sr,I)` in the input arc of Slowly Changing Dim represents the update of the dimensional record's status attribute; this function has two parameters, the first represents the dimensional record to be updated and the second represents the value of status to be updated:

```
    fun SetStatus(dr:DIMREC,s) = 1`DimRec(DIMREC.set_sta dr s);
```

In this case the value of the status (i.e., the `sta` field) is updated from 'A' to 'I', indicating that this dimensional record, from now on, is marked as inactive. Once again, to finish the deletion process of a dimensional record, the Log Journal must be updated with this information; for that a new log record is inserted with the timestamp of the operation, the name of the SCD, the name of the operation ('DeleteRecord' in this case) and the primary key of this record (i.e., the SK), leaving the field of the natural key empty.

3.3.7. Simulating the SCD-H

In this section the execution of the SCD-H process is described with the help of images taken during its simulation in CPN Tools. Figure 36 shows the initial marking of the SCD-H model (i.e., Marking *M0*). In the initial marking there is a single token in Audit Records, representing the list of unverified audit records. The list of records is defined through the `AuditRecordsList` constant as:

```
val AuditRecordsList = 1`[
AudRec({src="SRC",dtm="25/02/2012 13:45:33",opr=Insert,nk="dnk1",atb="ins-dnk1"}),
AudRec({src="SRC",dtm="25/02/2012 13:46:01",opr=Insert,nk="dnk2",atb="ins-dnk2"}),
AudRec({src="SRC",dtm="26/02/2012 11:35:13",opr=Update,nk="dnk2",atb="upd-dnk2"}),
AudRec({src="SRC",dtm="26/02/2012 12:10:44",opr=Delete,nk="dnk2",atb="del-dnk2"}),
AudRec({src="SRC",dtm="26/02/2012 12:45:33",opr=Insert,nk="dnk3",atb="ins-dnk3"}),
AudRec({src="SRC",dtm="27/02/2012 13:46:01",opr=Insert,nk="dnk4",atb="ins-dnk4"}),
AudRec({src="SRC",dtm="27/02/2012 15:18:02",opr=Update,nk="dnk4",atb="upd-dnk4"}),
AudRec({src="SRC",dtm="27/02/2012 15:28:55",opr=Update,nk="dnk4",atb="upd2-dnk4"}),
AudRec({src="SRC",dtm="28/02/2012 21:11:33",opr=Update,nk="dnk3",atb="upd-dnk3"})];
```

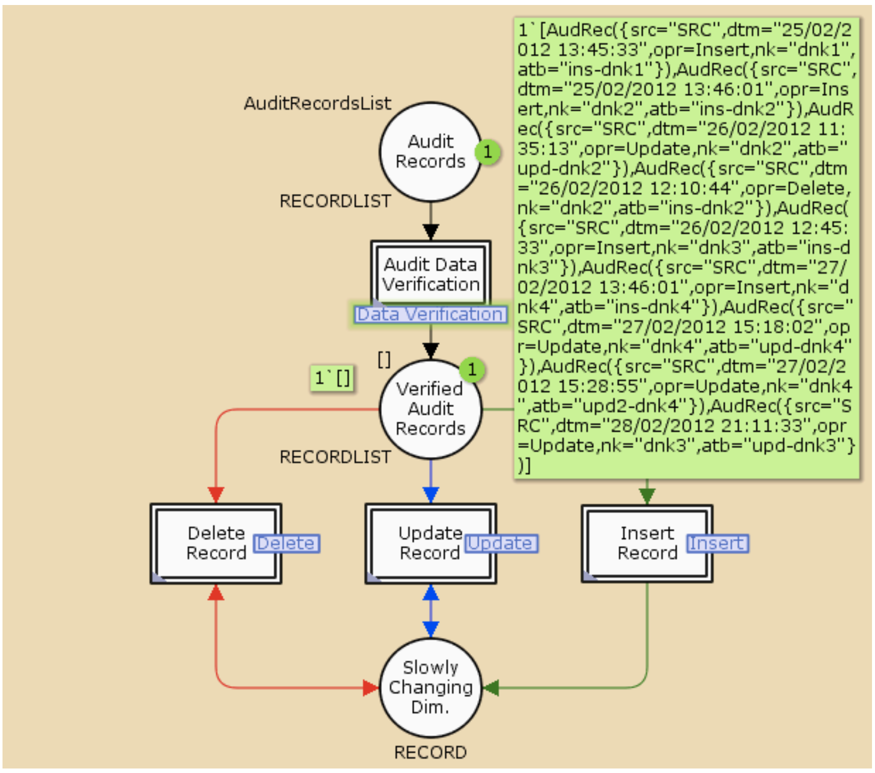

1`[AudRec({src="SRC",dtm="25/02/2012 13:45:33",opr=Insert,nk="dnk1",atb="ins-dnk1"}),AudRec({src="SRC",dtm="25/02/2012 13:46:01",opr=Insert,nk="dnk2",atb="ins-dnk2"}),AudRec({src="SRC",dtm="26/02/2012 11:35:13",opr=Update,nk="dnk2",atb="upd-dnk2"}),AudRec({src="SRC",dtm="26/02/2012 12:10:44",opr=Delete,nk="dnk2",atb="ins-dnk2"}),AudRec({src="SRC",dtm="26/02/2012 12:45:33",opr=Insert,nk="dnk3",atb="ins-dnk3"}),AudRec({src="SRC",dtm="27/02/2012 13:46:01",opr=Insert,nk="dnk4",atb="ins-dnk4"}),AudRec({src="SRC",dtm="27/02/2012 15:18:02",opr=Update,nk="dnk4",atb="upd-dnk4"}),AudRec({src="SRC",dtm="27/02/2012 15:28:55",opr=Update,nk="dnk4",atb="upd2-dnk4"}),AudRec({src="SRC",dtm="28/02/2012 21:11:33",opr=Update,nk="dnk3",atb="upd-dnk3"})]
AuditRecordsList
Audit Records
1
RECORDLIST
Audit Data Verification
Data Verification
[]
1`[]
1
Verified Audit Records
RECORDLIST
Delete Record
Delete
Update Record
Update
Insert Record
Insert
Slowly Changing Dim.
RECORD

Figure 36 – The SCDH module - Initial marking $M0$

There are a total of nine audit records in this list, four of them with the Insert operation type, one of them with the Delete operation type and the remaining records are updates of the previously inserted ones. The value in the `src` field is the same value for all the audit records, assuming that they come from the same source. It is also assumed that, once they are captured into the audit table in the DSA, they are ordered by the timestamp of the captured operation so that they can be processed in that same order. For modelling and simulation purposes, a single dimensional attribute is needed in each of the audit records (i.e., `atb` field) and the value of this field is the operation type followed by its natural key (e.g., 'ins-dnk1', meaning that the record with the 'dnk1' natural key was inserted). These values are only used to make it easier to view and analyse the behaviour and result of the three main operations and could be replaced by any other values.

The initial marking of the model has an additional token in Verified Audit Records with colour '[]' (i.e., an empty list). This means that no audit records have been verified yet and, for this reason, the Audit Data Verification substitution transition is the only one enabled in this sate; the remaining substitution transitions in the SCD-H module operate on verified audit records. Figure 37 shows the new marking $M1$ reached after the first audit record has been verified (i.e., the head of the list in the Audit Records place). In this case, the verification test is successful so the audit record 'AudRec({src="SRC", dtm="25/02/2012 13:45:33", opr=Insert, nk="dnk1", atb="ins-dnk1"})' is appended to the list in the Verified Audit Records place. Note that the existing transition remains enabled, as there are now eight audit records yet to be verified.

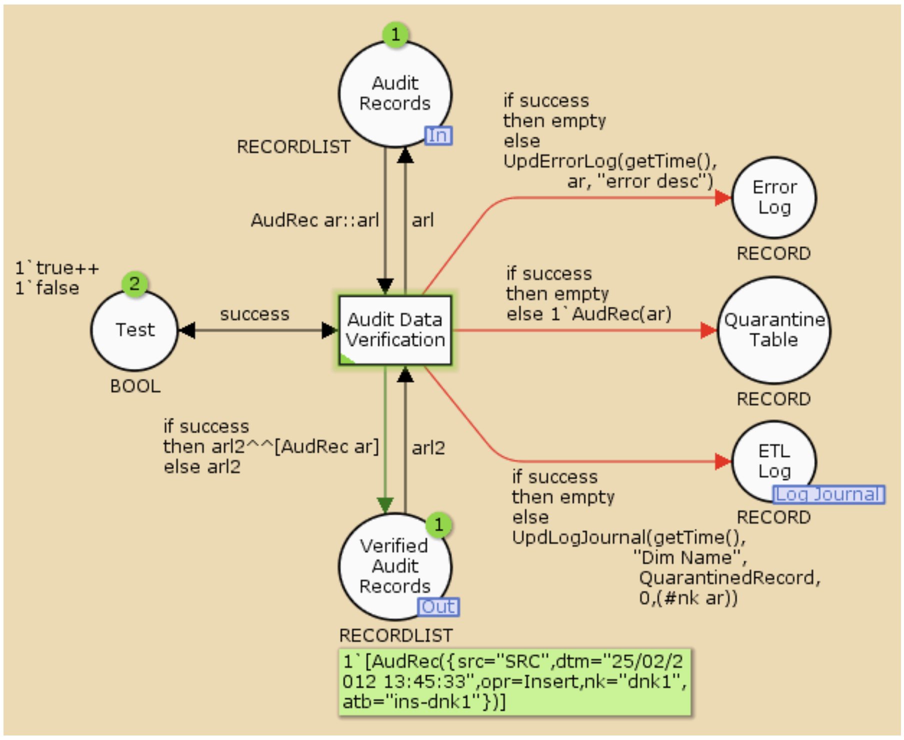

1
Audit Records
In
RECORDLIST
if success then empty else UpdErrorLog(getTime(), ar, "error desc")
Error Log
RECORD
AudRec ar::arl
arl
1`true++ 1`false
2
Test
BOOL
success
Audit Data Verification
if success then empty else 1`AudRec(ar)
Quarantine Table
RECORD
if success then arl2^^[AudRec ar] else arl2
arl2
ETL Log
Log Journal
RECORD
if success then empty else UpdLogJournal(getTime(), "Dim Name", QuarantinedRecord, 0,(#nk ar))
1
Verified Audit Records
Out
RECORDLIST
1`[AudRec({src="SRC",dtm="25/02/2012 13:45:33",opr=Insert,nk="dnk1", atb="ins-dnk1"})]

Figure 37 – The Data Verification module - Marking M1

As it has been explained during this module presentation, when this test fails the record in question must be quarantined and new entries need to be added to the respective logs. The marking *M5* in Figure 38 presents such a scenario, where the failure of the verification test is simulated for the the fifth audit record in the initial unverified list 'AudRec({src="SRC", dtm="26/02/2012 12:45:33", opr=Insert, nk="dnk3", atb="ins-dnk3"})'. The marking of place Verified Audit Records remains the same and the token representing the audit record is sent to the Quarantine Table instead; at the same time both places Error Log and ETL Log are updated with one token representing the error log record and the log journal record, respectively:

```
1`ErrLog(
     {src="SRC", errdtm="28/05/2012 17:15:41",
     dtm="26/02/2012 12:45:33", opr=Insert,
     nk="dnk3", atb="ins-dnk3", err="error desc"})

1`LogRec(
     {dtm="28/05/2012 17:15:41",dnm="Dim Name",
     opr=QuarantinedRecord, sk=0, nk="dnk3"})
```

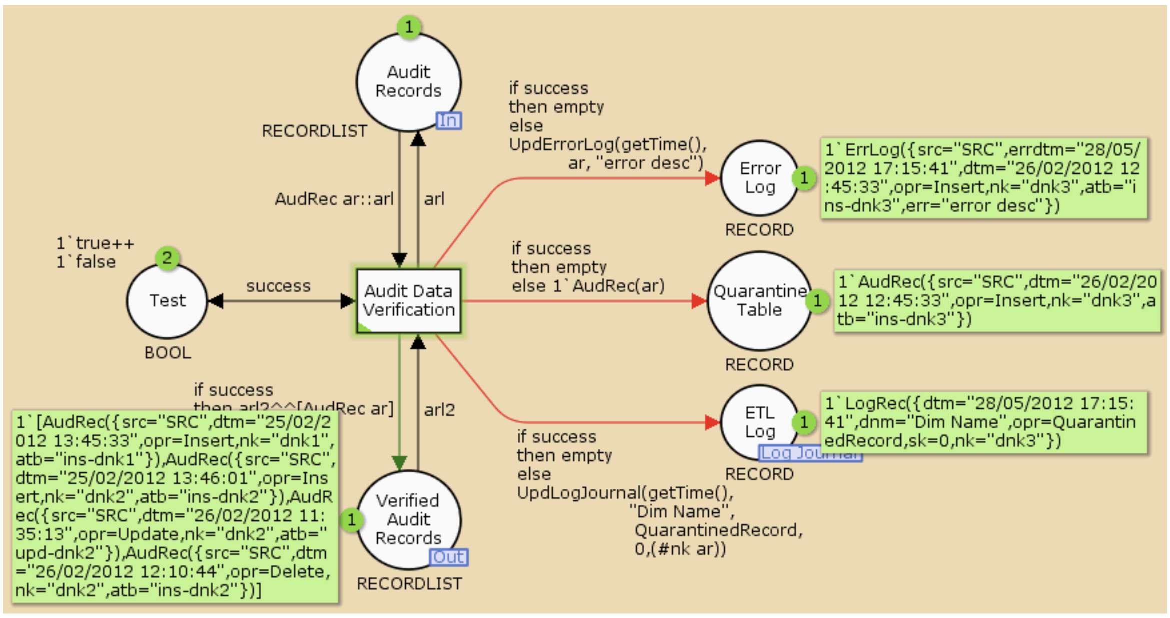

1
Audit Records
In
RECORDLIST
AudRec ar::arl
arl
if success
then empty
else
UpdErrorLog(getTime(),
ar, "error desc")
Error Log
1
RECORD
1`ErrLog({src="SRC",errdtm="28/05/2012 17:15:41",dtm="26/02/2012 12:45:33",opr=Insert,nk="dnk3",atb="ins-dnk3",err="error desc"})
1`true++
1`false
2
Test
BOOL
success
Audit Data Verification
if success
then empty
else 1`AudRec(ar)
Quarantine Table
1
RECORD
1`AudRec({src="SRC",dtm="26/02/2012 12:45:33",opr=Insert,nk="dnk3",atb="ins-dnk3"})
if success
then arl2^^[AudRec ar]
arl2
ETL Log
1
Log Journal
RECORD
1`LogRec({dtm="28/05/2012 17:15:41",dnm="Dim Name",opr=QuarantinedRecord,sk=0,nk="dnk3"})
if success
then empty
else
UpdLogJournal(getTime(),
"Dim Name",
QuarantinedRecord,
0,(#nk ar))
1`[AudRec({src="SRC",dtm="25/02/2012 13:45:33",opr=Insert,nk="dnk1",atb="ins-dnk1"}),AudRec({src="SRC",dtm="25/02/2012 13:46:01",opr=Insert,nk="dnk2",atb="ins-dnk2"}),AudRec({src="SRC",dtm="26/02/2012 11:35:13",opr=Update,nk="dnk2",atb="upd-dnk2"}),AudRec({src="SRC",dtm="26/02/2012 12:10:44",opr=Delete,nk="dnk2",atb="ins-dnk2"})]
1
Verified Audit Records
Out
RECORDLIST

Figure 38 – The Data Verification module - Marking *M5*

After all the audit records are verified the marking *M9* is reached (Figure 39). In this marking there are two tokens, just like the initial marking, but with different colours. The token in place Audit Records is now '[]' while the token in Verified Audit Records is the list of audit records that succeeded the verification test and, therefore, can be loaded into the SCD. This list now includes all the audit records except the one that was quarantined in the marking *M5*. The Insert Record is now the single enabled substitution transition as the operation type of the first verified audit record to be processed is 'Insert'.

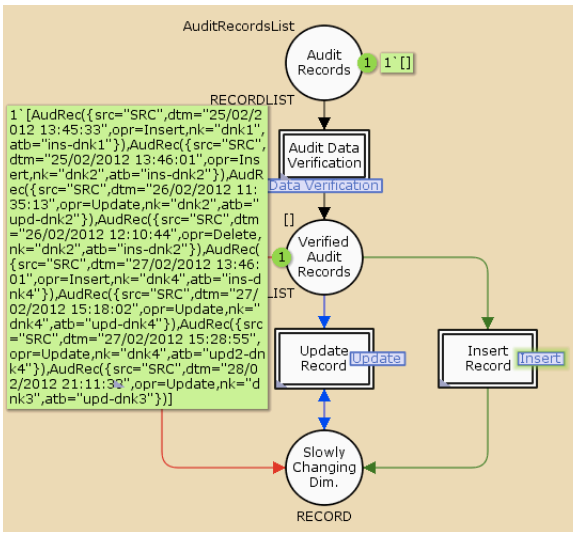

AuditRecordsList
Audit Records
1
1`[]
RECORDLIST
1`[AudRec({src="SRC",dtm="25/02/2012 13:45:33",opr=Insert,nk="dnk1",atb="ins-dnk1"}),AudRec({src="SRC",dtm="25/02/2012 13:46:01",opr=Insert,nk="dnk2",atb="ins-dnk2"}),AudRec({src="SRC",dtm="26/02/2012 11:35:13",opr=Update,nk="dnk2",atb="upd-dnk2"}),AudRec({src="SRC",dtm="26/02/2012 12:10:44",opr=Delete,nk="dnk2",atb="ins-dnk2"}),AudRec({src="SRC",dtm="27/02/2012 13:46:01",opr=Insert,nk="dnk4",atb="ins-dnk4"}),AudRec({src="SRC",dtm="27/02/2012 15:18:02",opr=Update,nk="dnk4",atb="upd-dnk4"}),AudRec({src="SRC",dtm="27/02/2012 15:28:55",opr=Update,nk="dnk4",atb="upd2-dnk4"}),AudRec({src="SRC",dtm="28/02/2012 21:11:32",opr=Update,nk="dnk3",atb="upd-dnk3"})]
Audit Data Verification
Data Verification
[]
Verified Audit Records
1
LIST
Update Record
Update
Insert Record
Insert
Slowly Changing Dim.
RECORD

Figure 39 – The SCDH module - Marking $M9$

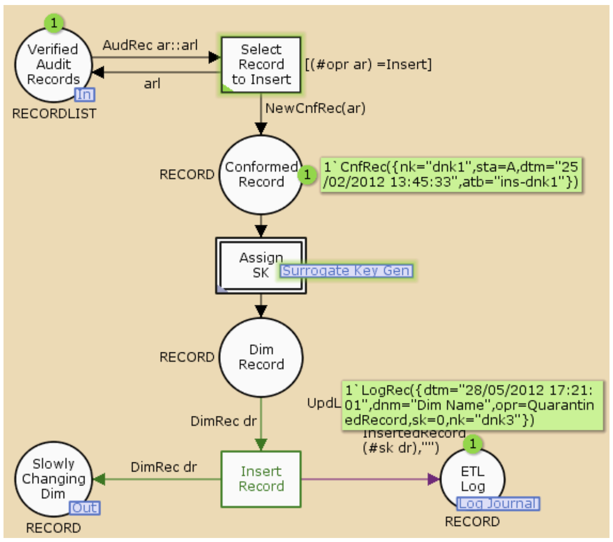


Figure 40 – The Insert module - Marking $M10$

The next marking ($M10$) is reached after firing the Select Record to Insert transition in the Insert module shown in Figure 40. The head of the verified audit record list is removed from it and passed to the transition and the function `NewCnfRec(ar)` is applied to the audit record. The result is the token '1`CnfRec({nk="dnk1", sta=A, dtm="25/02/2012 13:45:33", atb="ins-dnk1"})', representing a conformed record that is put in the Conformed Record place. This function is used to modify the structure of the audit record and prepare it for the consequent operations. The guard in the transition acts as a restriction so that it activates only if the head of the list is a record to be inserted, meaning that the new head of the list is also a record to be inserted because the transition is enabled. In the current marking the substitution transition Assign SK is also enabled, receiving as input the token representing the conformed record. Figure 41 shows the current marking ($M10$) in the Surrogate Key Gen module, where the Generate SK transition is enabled and the record in Conformed Record is the same in both this module and the Insert module shown in Figure 40. There is one additional token in place Counter with colour '1', representing the value of the first SK to be assigned.

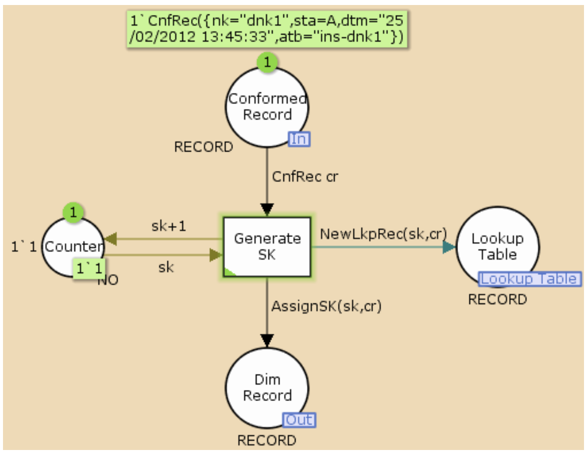


Figure 41 – The Surrogate Key Gen module - Marking $M10$

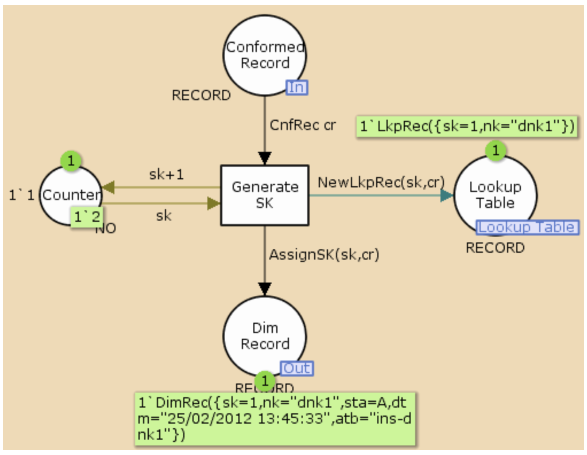


Figure 42 – The Surrogate Key Gen module - Marking $M11$

The marking $M11$ (Figure 42) is reached after the SK is assigned to the conformed record and the resulting dimensional record 'DimRec({sk=1, nk="dnk1", sta=A, dtm="25/02/2012 13:45:33", atb="ins-dnk1"})' is put in the place Dim Record through the `AssignSK(sk,cr)` arc expression. At the same time, the value of SK stored in place Counter is incremented and the token 'LkpRec({sk=1, nk="dnk1"})', representing the lookup record used to map the natural key of the conformed record with the newly assigned SK, is put in place Lookup Table. The same marking ($M11$) is also shown in Figure 43 for the Insert module where the substitution transition is no longer enabled, i.e., the SK has already been assigned, and the resulting token rests in Dim Record. The Insert Record transition, representing the last step of the Insert process, is now enabled. When this transition is triggered the token representing the dimensional record is simply passed to the *output port* Slowly Changing Dim, while the log journal record 'LogRec({dtm="28/05/2012 17:30:53", dnm="Dim.Name", opr=InsertedRecord, sk=1, nk=""})' is created and passed to the *fusion place* ETL Log. This marking ($M12$) and the results of the Insert Record transition are shown in Figure 44.

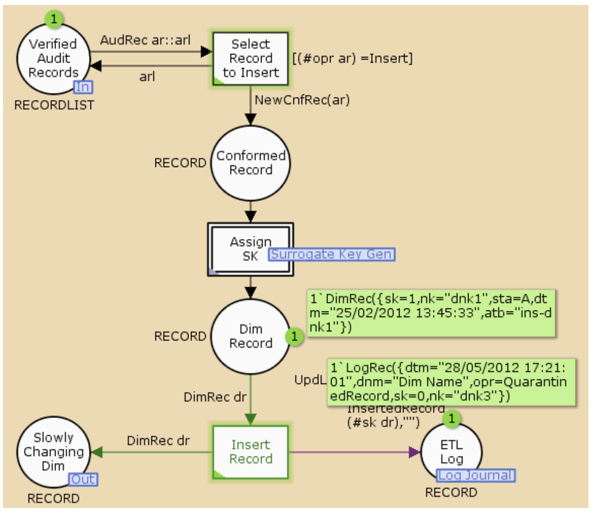


Figure 43 – The Insert module - Marking $M11$

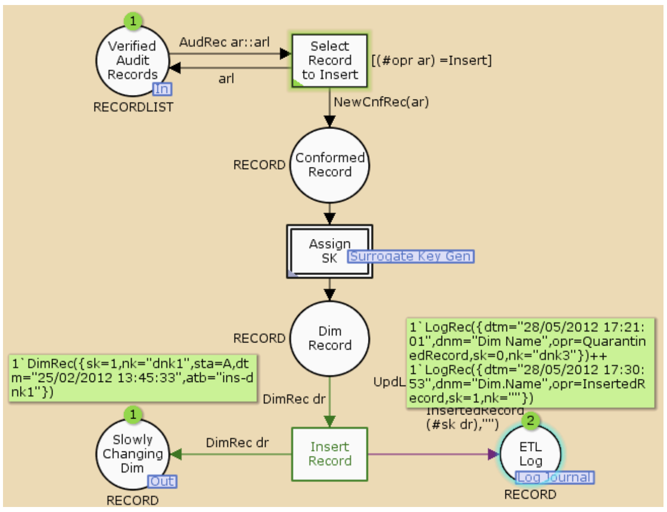

Verified Audit Records
In
RECORDLIST
AudRec ar::arl
arl
Select Record to Insert
[(#opr ar) =Insert]
NewCnfRec(ar)
RECORD
Conformed Record
Assign SK
Surrogate Key Gen
RECORD
Dim Record
1`DimRec({sk=1,nk="dnk1",sta=A,dtm="25/02/2012 13:45:33",atb="ins-dnk1"})
1`LogRec({dtm="28/05/2012 17:21:01",dnm="Dim Name",opr=QuarantinedRecord,sk=0,nk="dnk3"})++
1`LogRec({dtm="28/05/2012 17:30:53",dnm="Dim.Name",opr=InsertedRecord,sk=1,nk=""})
UpdL
DimRec dr
InsertedRecord (#sk dr),"")
Slowly Changing Dim
Out
RECORD
DimRec dr
Insert Record
ETL Log
Log Journal
RECORD

Figure 44 – The Insert module - Marking $M12$

After the next token with colour 'AudRec({src="SRC", dtm="25/02/2012 13:46:01", opr=Insert, nk="dnk2", atb="ins-dnk2"})' is inserted, the Select Record to Update transition in the Update module activates and consequently, the Update Record substitution transition in the SCD-H module is also enabled, similarly to what happened with the Insert module. Figure 45 shows the Update module in the marking $M16$, reached after the occurrence of the Select Record to Update transition. In this marking the audit record 'AudRec({src="SRC", dtm="26/02/2012 11:35:13", opr=Update, nk="dnk2", atb="upd-dnk2"})' is removed from the verified audit record list and passed to Records to Update without any transformation because the record itself is not going to be inserted in the dimension; instead, its dimensional attributes will be used to update the token representing the corresponding dimensional record in the place Slowly Changing Dim.

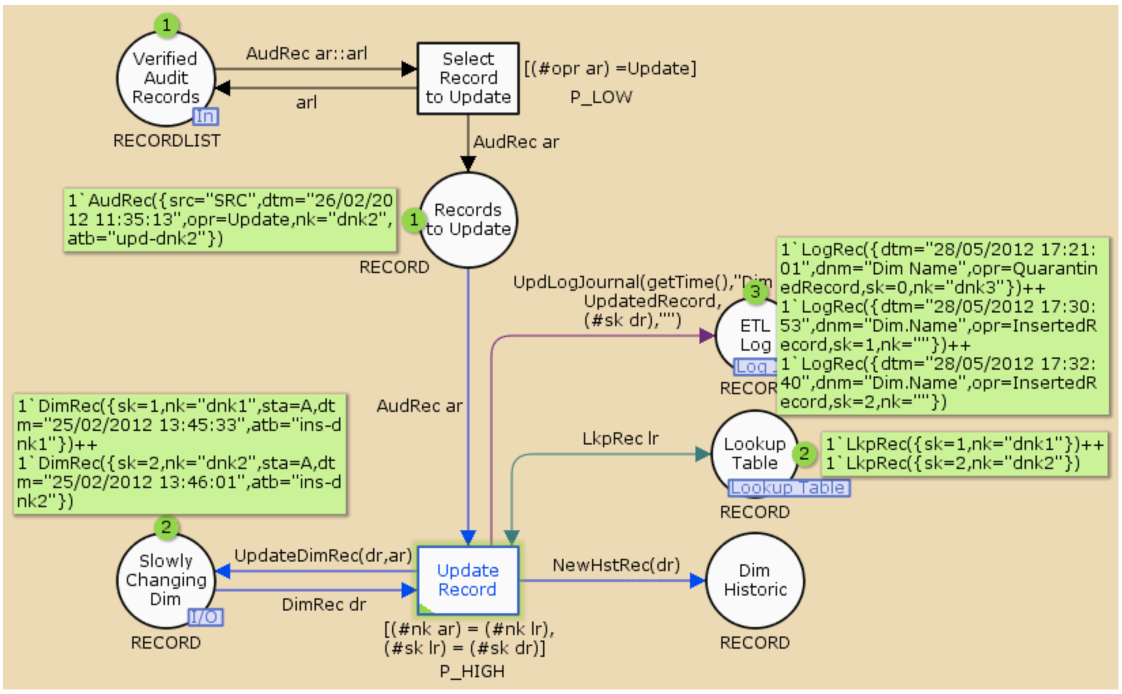

1
Verified Audit Records
In
RECORDLIST
AudRec ar::arl
arl
Select Record to Update
[(#opr ar) =Update]
P_LOW
AudRec ar
1`AudRec({src="SRC",dtm="26/02/2012 11:35:13",opr=Update,nk="dnk2",atb="upd-dnk2"})
1
Records to Update
RECORD
UpdLogJournal(getTime(),"Dim UpdatedRecord,(#sk dr),"")
3
ETL Log
Log
RECORD
1`LogRec({dtm="28/05/2012 17:21:01",dnm="Dim Name",opr=QuarantinedRecord,sk=0,nk="dnk3"})++
1`LogRec({dtm="28/05/2012 17:30:53",dnm="Dim.Name",opr=InsertedRecord,sk=1,nk=""})++
1`LogRec({dtm="28/05/2012 17:32:40",dnm="Dim.Name",opr=InsertedRecord,sk=2,nk=""})
AudRec ar
1`DimRec({sk=1,nk="dnk1",sta=A,dtm="25/02/2012 13:45:33",atb="ins-dnk1"})++
1`DimRec({sk=2,nk="dnk2",sta=A,dtm="25/02/2012 13:46:01",atb="ins-dnk2"})
LkpRec lr
Lookup Table
2
1`LkpRec({sk=1,nk="dnk1"})++
1`LkpRec({sk=2,nk="dnk2"})
Lookup Table
RECORD
2
Slowly Changing Dim
I/O
RECORD
UpdateDimRec(dr,ar)
DimRec dr
Update Record
[(#nk ar) = (#nk lr),
(#sk lr) = (#sk dr)]
P_HIGH
NewHstRec(dr)
Dim Historic
RECORD

Figure 45 – The Update module - Marking $M16$

The Update Record is the only enabled transition in this marking, which means that there is a correspondence between the tokens representing the audit record, used in the update, and the dimensional record that is retrieved from place Slowly Changing Dim by the means of a lookup record, created when this dimensional record was inserted. The first part of the guard expression of the enabled transition evaluates to *true* as there is a token in Lookup Table with the same ('dnk2') natural key value as the audit record in the place Records to Update. The surrogate key value of the token removed from place Lookup Table is used, in the second part of the guard expression, to find the dimensional record with the same value ('2') in the sk field. When both parts of the expression are true, then there is a token, representing a dimensional record, that corresponds to the incoming token, representing the audit record, and the update process can continue.

In the following marking (Figure 46) both the attribute (atb) and the timestamp (dtm) fields of the dimensional record with the SK value '2' have renewed values. For the transition to be fired, this token is removed from place Slowly Changing Dim and used as a parameter in the UpdateDimRec function in the respective output arc together with the token representing the audit record. The dimensional attribute and timestamp fields of the dimensional record are updated to the same values of these fields in the audit record resulting in a new token representing the updated dimensional record '1`DimRec({sk=2, nk="dnk2", sta=I, dtm="26/02/2012 11:35:13", atb="upd-dnk2"})'. Notice how the `atb` field now indicates that it was updated and the value of the `dtm` field is the same as the timestamp of the update operation captured in the source system. This process is not complete until the former values of the dimensional record are preserved in the place Dim Historic, which is accomplished by using the token representing the dimensional record removed from place Slowly Changing Dim as a parameter of the `NewHstRec` function in the remaining output arc. This function creates and passes the new token '1`HstRec({sk=2, nk="dnk2", dtm="25/02/2012 13:46:01", atb="ins-dnk2"})' to place Dim Historic with the historic values of the dimensional record. This operation is once again registered in place Log Jounal and the new token '1`LogRec({dtm="28/05/2012 17:35:21", dnm="Dim Name", opr=UpdatedRecord, sk=2, nk=""})' is created and put in place ETL Log for this purpose.

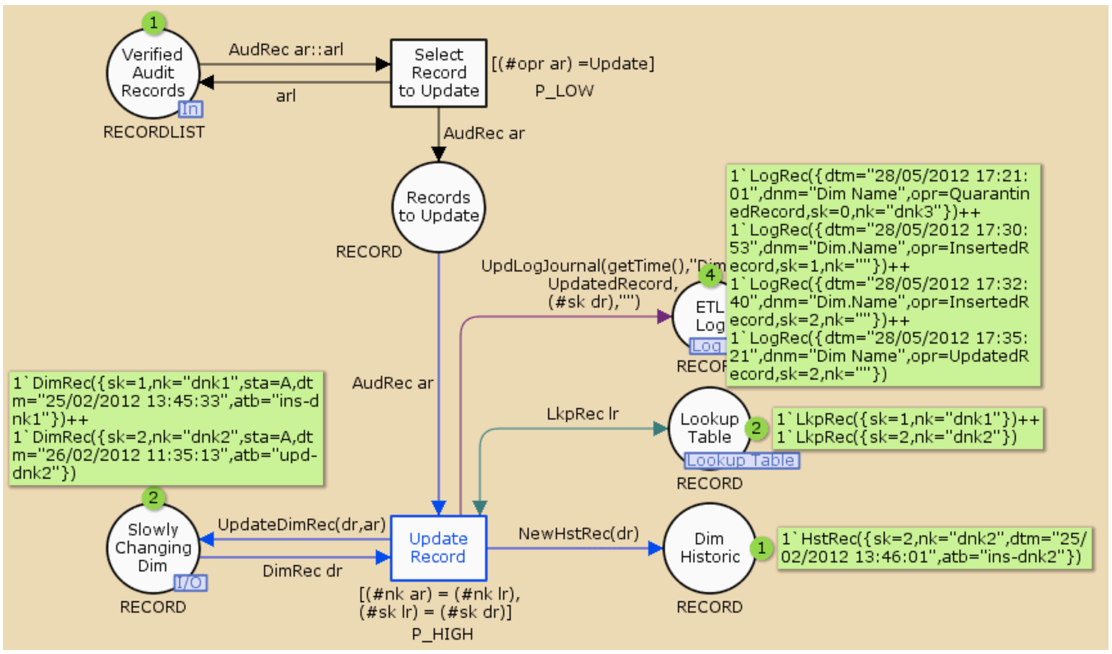

1
Verified Audit Records
In
RECORDLIST
AudRec ar::arl
arl
Select Record to Update
[(#opr ar) =Update]
P_LOW
AudRec ar
Records to Update
RECORD
UpdLogJournal(getTime(),"Dim Name", UpdatedRecord, (#sk dr),"")
4
ETL Log
Log
RECORD
1`LogRec({dtm="28/05/2012 17:21:01",dnm="Dim Name",opr=QuarantinedRecord,sk=0,nk="dnk3"})++
1`LogRec({dtm="28/05/2012 17:30:53",dnm="Dim.Name",opr=InsertedRecord,sk=1,nk=""})++
1`LogRec({dtm="28/05/2012 17:32:40",dnm="Dim.Name",opr=InsertedRecord,sk=2,nk=""})++
1`LogRec({dtm="28/05/2012 17:35:21",dnm="Dim Name",opr=UpdatedRecord,sk=2,nk=""})
AudRec ar
LkpRec lr
Lookup Table
Lookup Table
RECORD
2
1`LkpRec({sk=1,nk="dnk1"})++
1`LkpRec({sk=2,nk="dnk2"})
1`DimRec({sk=1,nk="dnk1",sta=A,dtm="25/02/2012 13:45:33",atb="ins-dnk1"})++
1`DimRec({sk=2,nk="dnk2",sta=A,dtm="26/02/2012 11:35:13",atb="upd-dnk2"})
2
Slowly Changing Dim
I/O
RECORD
UpdateDimRec(dr,ar)
DimRec dr
Update Record
[(#nk ar) = (#nk lr), (#sk lr) = (#sk dr)]
P_HIGH
NewHstRec(dr)
Dim Historic
RECORD
1
1`HstRec({sk=2,nk="dnk2",dtm="25/02/2012 13:46:01",atb="ins-dnk2"})

Figure 46 – The Update module - Marking $M17$

The selection of a record to be deleted from the verified audit record list is processed in the very same way as in the Update module described previously. The selection of the first record to be deleted happens in marking $M17$, when transition Select Record to Delete fires (Figure 47). In marking $M18$, there is one token in place Records to Delete representing the information in the SCD that needs to be deleted or, in this case, simply marked as inactive. Again, place Lookup Table and the guard expression are used to make a correspondence between the correct token in place Slowly Changing Dim and the token in place Records to Delete, analogously to what happens in the Update module.

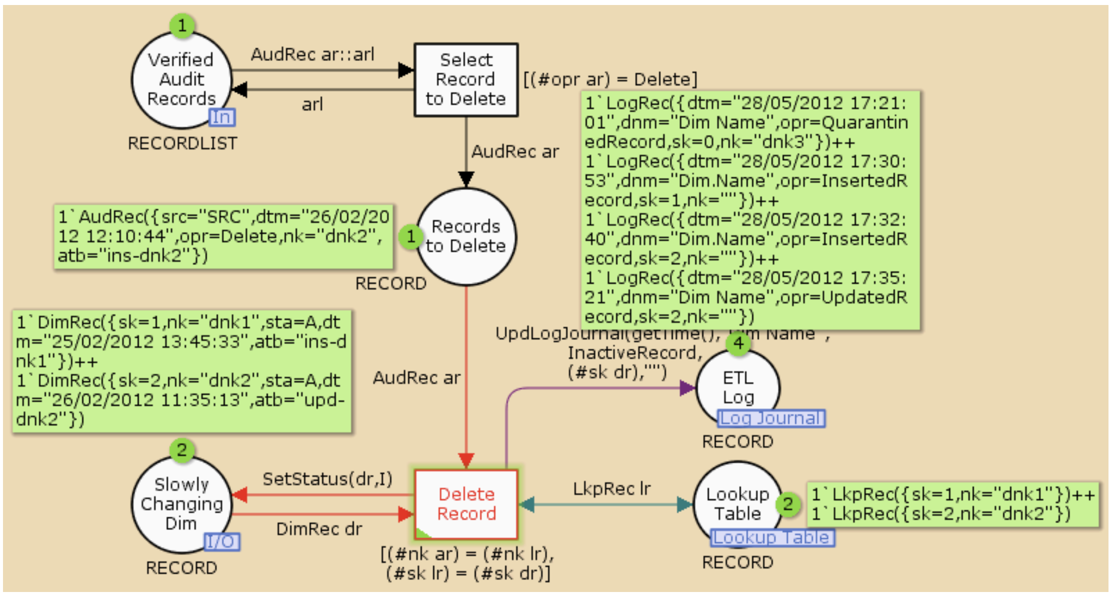

Verified Audit Records
In
RECORDLIST
1
AudRec ar::arl
arl
Select Record to Delete
[(#opr ar) = Delete]
AudRec ar
1`LogRec({dtm="28/05/2012 17:21:01",dnm="Dim Name",opr=QuarantinedRecord,sk=0,nk="dnk3"})++
1`LogRec({dtm="28/05/2012 17:30:53",dnm="Dim.Name",opr=InsertedRecord,sk=1,nk=""})++
1`LogRec({dtm="28/05/2012 17:32:40",dnm="Dim.Name",opr=InsertedRecord,sk=2,nk=""})++
1`LogRec({dtm="28/05/2012 17:35:21",dnm="Dim Name",opr=UpdatedRecord,sk=2,nk=""})
1`AudRec({src="SRC",dtm="26/02/2012 12:10:44",opr=Delete,nk="dnk2",atb="ins-dnk2"})
Records to Delete
1
RECORD
1`DimRec({sk=1,nk="dnk1",sta=A,dtm="25/02/2012 13:45:33",atb="ins-dnk1"})++
1`DimRec({sk=2,nk="dnk2",sta=A,dtm="26/02/2012 11:35:13",atb="upd-dnk2"})
AudRec ar
UpdLogJournal(getTime(), "Dim Name", InactiveRecord, (#sk dr),"")
4
ETL Log
Log Journal
RECORD
2
Slowly Changing Dim
I/O
RECORD
SetStatus(dr,I)
DimRec dr
Delete Record
[(#nk ar) = (#nk lr), (#sk lr) = (#sk dr)]
LkpRec lr
Lookup Table
2
Lookup Table
RECORD
1`LkpRec({sk=1,nk="dnk1"})++
1`LkpRec({sk=2,nk="dnk2"})

Figure 47 – The Delete module - Marking $M18$

When the transition fires, the tokens representing the audit record and the corresponding dimensional record are removed from their source places. The audit record is discarded and not used further in this process, because it is only relevant to find the respective dimensional record. After that, all that is left to do is to mark it as inactive and return it to place Slowly Changing Dim. This is accomplished through the `SetStatus` function in the red output arc expression, which creates the token '1`DimRec({sk=2, nk="dnk2", sta=I, dtm="26/02/2012 11:35:13", atb="upd-dnk2"})', where the value of the `sta` field is now `I´ (i.e., Inactive). Once again, the process isn't complete before the new token '1`LogRec({dtm="28/05/2012 17:38:13", dnm="Dim Name", opr=InactiveRecord, sk=2, nk=""})', which represents a log journal record, is created and put in place ETL Log (Figure 48).

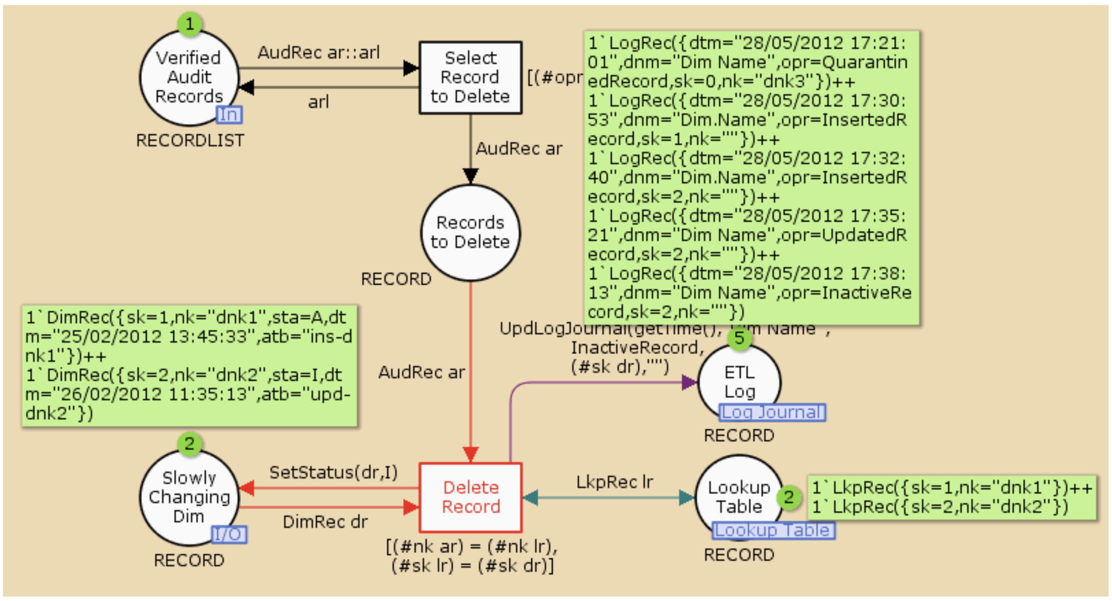

1
Verified Audit Records
In
RECORDLIST
AudRec ar::arl
arl
Select Record to Delete
[(#opr
AudRec ar
Records to Delete
RECORD
1`LogRec({dtm="28/05/2012 17:21:01",dnm="Dim Name",opr=QuarantinedRecord,sk=0,nk="dnk3"})++
1`LogRec({dtm="28/05/2012 17:30:53",dnm="Dim.Name",opr=InsertedRecord,sk=1,nk=""})++
1`LogRec({dtm="28/05/2012 17:32:40",dnm="Dim.Name",opr=InsertedRecord,sk=2,nk=""})++
1`LogRec({dtm="28/05/2012 17:35:21",dnm="Dim Name",opr=UpdatedRecord,sk=2,nk=""})++
1`LogRec({dtm="28/05/2012 17:38:13",dnm="Dim Name",opr=InactiveRecord,sk=2,nk=""})
1`DimRec({sk=1,nk="dnk1",sta=A,dtm="25/02/2012 13:45:33",atb="ins-dnk1"})++
1`DimRec({sk=2,nk="dnk2",sta=I,dtm="26/02/2012 11:35:13",atb="upd-dnk2"})
AudRec ar
UpdLogJournal(getTime(),"Dim Name",
InactiveRecord,
(#sk dr),"")
5
ETL Log
Log Journal
RECORD
2
Slowly Changing Dim
I/O
RECORD
SetStatus(dr,I)
DimRec dr
Delete Record
LkpRec lr
Lookup Table
2
Lookup Table
RECORD
1`LkpRec({sk=1,nk="dnk1"})++
1`LkpRec({sk=2,nk="dnk2"})
[(#nk ar) = (#nk lr),
(#sk lr) = (#sk dr)]

Figure 48 – The Delete module - Marking $M19$

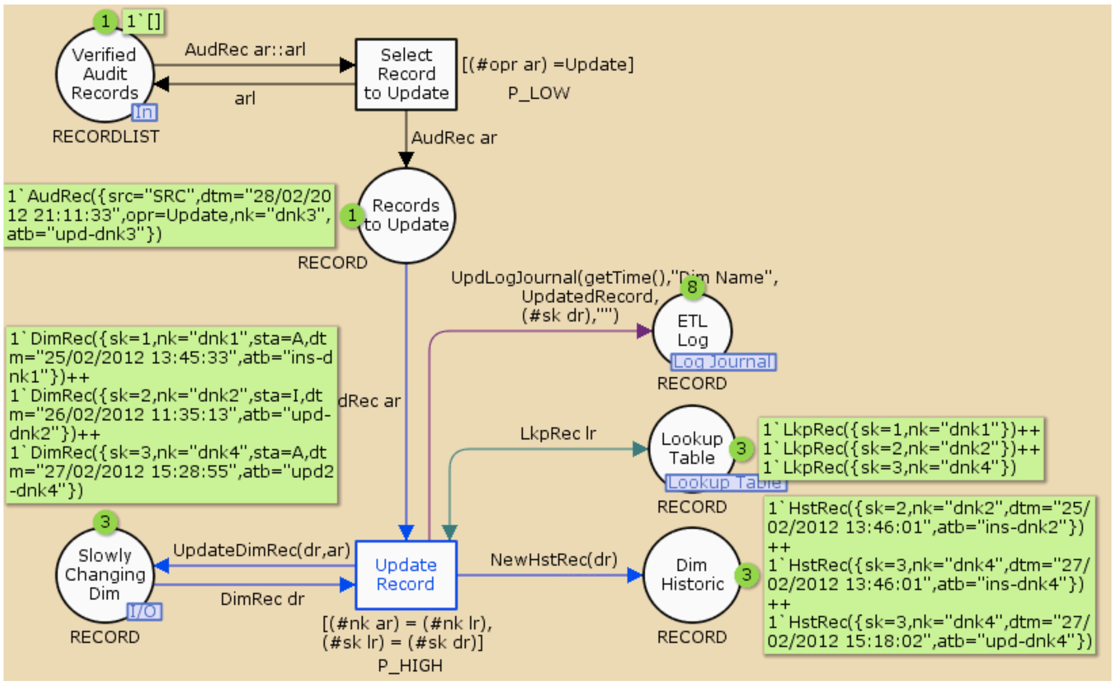

1
1`[]
Verified Audit Records
In
RECORDLIST
AudRec ar::arl
arl
Select Record to Update
[(#opr ar) =Update]
P_LOW
AudRec ar
1`AudRec({src="SRC",dtm="28/02/2012 21:11:33",opr=Update,nk="dnk3",atb="upd-dnk3"})
1
Records to Update
RECORD
UpdLogJournal(getTime(),"Dim Name",
UpdatedRecord,
(#sk dr),"")
8
ETL Log
Log Journal
RECORD
1`DimRec({sk=1,nk="dnk1",sta=A,dtm="25/02/2012 13:45:33",atb="ins-dnk1"})++
1`DimRec({sk=2,nk="dnk2",sta=I,dtm="26/02/2012 11:35:13",atb="upd-dnk2"})++
1`DimRec({sk=3,nk="dnk4",sta=A,dtm="27/02/2012 15:28:55",atb="upd2-dnk4"})
dRec ar
LkpRec lr
Lookup Table
3
Lookup Table
RECORD
1`LkpRec({sk=1,nk="dnk1"})++
1`LkpRec({sk=2,nk="dnk2"})++
1`LkpRec({sk=3,nk="dnk4"})
3
Slowly Changing Dim
I/O
RECORD
UpdateDimRec(dr,ar)
DimRec dr
Update Record
[(#nk ar) = (#nk lr),
(#sk lr) = (#sk dr)]
P_HIGH
NewHstRec(dr)
Dim Historic
3
RECORD
1`HstRec({sk=2,nk="dnk2",dtm="25/02/2012 13:46:01",atb="ins-dnk2"})++
1`HstRec({sk=3,nk="dnk4",dtm="27/02/2012 13:46:01",atb="ins-dnk4"})++
1`HstRec({sk=3,nk="dnk4",dtm="27/02/2012 15:18:02",atb="upd-dnk4"})

Figure 49 – The Update module - Marking $M27$

In the final marking ($M27$), shown in Figure 49, the colour of the token in place Verified Audit Records is '[]', meaning that all the records have been processed in the respective modules. There are three tokens in place Lookup Table, representing the lookup records that were created when the corresponding records were inserted. Because the record with the natural key 'dnk3' didn't pass the verification test and was quarantined, it wasn't processed in this simulation and, therefore, no lookup record was created to match that natural key with a surrogate key. The token in place Records to Update represents a verified audit record that should be used to update the record that was quarantined. As there is no correspondence between this record and the lookup dimensional records (i.e., the guard expression evaluates to false), the token representing the audit record rests in that place until the quarantined record is fit to be inserted in the SCD. Place Slowly Changing Dim has one additional token representing a dimensional record that was inserted and updated two times between the markings *M19* and *M27* as it can be observed by the two additional tokens in place Dim Historic.

Chapter 4

ETL CPN Modules

4.1. The Base for ETL CPN Modules

The CPN models presented in the previous chapter represent ETL operations that are used as standard tasks in many ETL systems, more specifically when loading data into the DW. In this chapter, one of the most interesting and important ETL operations is presented and modelled: the Change Data Capture (CDC). The CDC is responsible for identifying and registering all the data modifications that happens in the operational source(s) of a DW and store them in audit tables. These data, kept in the DSA, may then be submitted to a number of transformations before they are loaded into the DW by the SKP or SCD-H processes, depending on their type and quality. The objective of creating these modules is to use them in the design and validation of ETL systems, simply by adding and relating the already defined packages. To illustrate this, an ETL CPN model formed by the SKP, SCD-H and CDC processes is also presented in this chapter.

4.2. Change Data Capture

One of the main functions of a DW is the preservation of the state of the data across time (i.e., time-variance of data). This means that every modification to the data of the source systems –

being it the insertion of new rows, the deletion of rows, or the modification of the values the rows– must be preserved in the DW by the means of SCD tables. These special dimensions contain the most recent state of the data of the source systems and its former states, stored in a special historic table (Section 3.2). For these SCD to be loaded, there is the need to identify and capture the changes that occur in the tables of the source systems, so that they can be delivered to the target system (i.e., the DW) - such task can be accomplished through *Change Data Capture* (CDC) design patterns.

There are a number of different ways to set up a CDC mechanism. The simpler ones consist on having one or more dedicated columns in the source tables, in order to track the row modifications, such as timestamp, version number and status indicator columns whose values need to be updated when a modification takes place. This CDC mechanism, although simple, must be considered in the design stage of the database of the source system, so that the tables that need to be audited possess the correct structure right from scratch. Another problem with this method, depending on the size of the database and the number of transactions, is the accentuated growth of the tables that need to be audited, since every record that is updated/deleted needs to be kept in the same table for auditing purposes.

A more reliable method to implement the CDC mechanism is the use of triggers on the tables that need to be audited (also known as log trigger). The log trigger acts as an automatic mechanism to record information about changes that occur in a specific transactional table into a corresponding audit table; the latter acts as an ordered queue that can later be directly used in the population of the target SCD. This method overcomes the downsides of the previously presented method, by using audit tables the impact on the source tables that need to be audited is minimum and their structure remains unaltered; however, it is still considered to be an intrusive CDC method since the log trigger may not be implemented in the transactional source tables that need to be audited and, in many occasions, the DW Administrator may not have the permission to do so during the implementation of this ETL process.

A different manner to capture the changes made in the databases of the operational sources is to read the system transaction log file to find the modifications that occurred in the source tables that need to be audited. This CDC mechanism is the less intrusive one, with minimal impact on the source systems, since the schema of the database does not need to be modified and no triggers need to be implemented in the tables. Instead, all the information about data modification is scanned and interpreted from the transaction log into audit tables that can be similar to the ones used in the log trigger CDC mechanism. Because of the non-intrusive manner in which the source modifications are captured, this is the CDC method adopted for this ETL process. The reading and interpretation of the transaction log presents itself as the most challenging part of the process since each DBMS adopts a specific structure and content for its transaction log file, without providing the much needed documentation. The SQL Server 2008 was the chosen DBMS for the analysis of the structure of the transaction log file and the interpretation of its content so that, from this initial study, the corresponding CPN model can be designed.

4.2.1. The Transaction Log

In SQL Server 2008 the `fn_dblog` function allows us to view the transaction log as a relational table. This function has two optional parameters; the *starting* and *ending* log sequence number (LSN), which can be viewed as the unique identifier of each log record. Replacing the 'NULL' values with an actual LSN limits the number of presented log records. The first step of this modelling process is to analyse the structure and content of the transaction log, as well as the implication of each of the records that belong to transactions created by *Data Manipulation Language* (DML) queries, as these are the ones responsible for the updates, insertions, and deletions. The transaction log has 119 columns, the majority of them possess solely 'NULL' values and their importance is left unknown, as there is no available documentation. For this reason a view of the transaction log (Table 1) was created with the necessary and most relevant attributes, for an easier analysis, and a subset of the view is displayed in the following table.

LSN	T. ID	Operation	T. Name	End Time	AllocUnitName	RowLog Contents
1	1	LOP_BEGIN_XACT	INSERT	NULL	NULL	NULL
2	1	LOP_INSERT_ROWS	NULL	NULL	dbo.TestTable	0x10006C0...
3	1	LOP_COMMIT_XACT	NULL	2012/08/03 22:57:05	NULL	NULL
4	2	LOP_BEGIN_XACT	UPDATE	NULL	NULL	NULL
5	2	LOP_MODIFY_ROW	NULL	NULL	dbo.TestTable	0x63
6	2	LOP_MODIFY_ROW	NULL	NULL	dbo.TestTable	0x63
7	2	LOP_COMMIT_XACT	NULL	2012/08/03 22:58:29	NULL	NULL
8	3	LOP_BEGIN_XACT	DELETE	NULL	NULL	NULL
9	3	LOP_DELETE_ROWS	NULL	NULL	dbo.TestTable	0x10006C0...
10	3	LOP_DELETE_ROWS	NULL	NULL	dbo.TestTable	0x10006C0...
11	3	LOP_COMMIT_XACT	NULL	2012/08/03 22:58:49	NULL	NULL

Table 1 – A Transaction Log View

The created view has only 7 attributes, of the initial 119, which is, in our point of view, sufficient for the CDC process that we intend to model. The LSN is the unique identifier of each transaction log record, while the T. ID (i.e., the Transaction ID) is the reference to the database transaction that generated the log record. Note that the same transaction usually creates several entries in the transaction log. Both these attributes contain hexadecimal values that can be long and more difficult to differentiate. In this example they were replaced by sequential integer values (with initial value '1') in order to simplify the analysis and interpretation of each record. The Operation represents the type of operation that was performed and recorded in the log; each transaction begins with the LOP_BEGIN_XACT operation, and ends with the LOP_COMMIT_XACT operation. The T. Name indicates the name of the transaction, which can be identified in its first log record, i.e., the LOP_BEGIN_XACT record. In the remaining records of the transaction, this field is left as 'NULL'. Each transaction has an associated timestamp and the End Time is recorded in each LOP_COMMIT_XACT statement and left as 'NULL' in the remaining log records. The name of the schema and table, where the modifications occurred, are stored in the AllocUnitName for the log records that represent the actual modifications and are left as 'NULL' in the remaining records.

The types of operations, their timestamps and the name of the tables where the modifications took place have been identified, so the final step of this process is the identification of the attributes of the transaction that store information about the actual modifications, that is, the names of the attributes, their types and the modified values. Unfortunately, this information is not directly displayed in the transaction log. Instead, it is stored as a hexadecimal value in a series of attributes named Row Log Contents. There are a series of 5 Row Log Contents attributes in the transaction log, numbered from 0 to 4, which store the modified information according to the type of operation. In this example, once again to make it simpler and more readable, only one attribute is used to represent this series of attributes; the actual extraction of data from these attributes is out of the scope of this study but is simulated in the implemented model of the CDC process. Table 1 represents 3 transactions and 11 log records. The type of the first transaction can be identified in the T. Name attribute (i.e., an Insert) of the first record, the following record is the one that holds the information about the inserted row, while the timestamp of the operation is presented in the End Time attribute of the final log record of the first transaction. The second transaction, an Update, begins in the fourth record of the table; in this transaction there are two records with the LOP_MODIFY_ROW operation, meaning that two rows were updated with a single DML query. The third transaction, representing a Delete operation on two rows, can be interpreted in an analogous way.

4.2.2. The Change Data Capture CPN Model

The CDC process is implemented using a hierarchical CPN composed of three main modules that hold the most intricate details and behaviour of the CPN model. The prime module, composed of the three sub-modules, allows for a more abstract and cleaner view of the entire process (Figure 50).

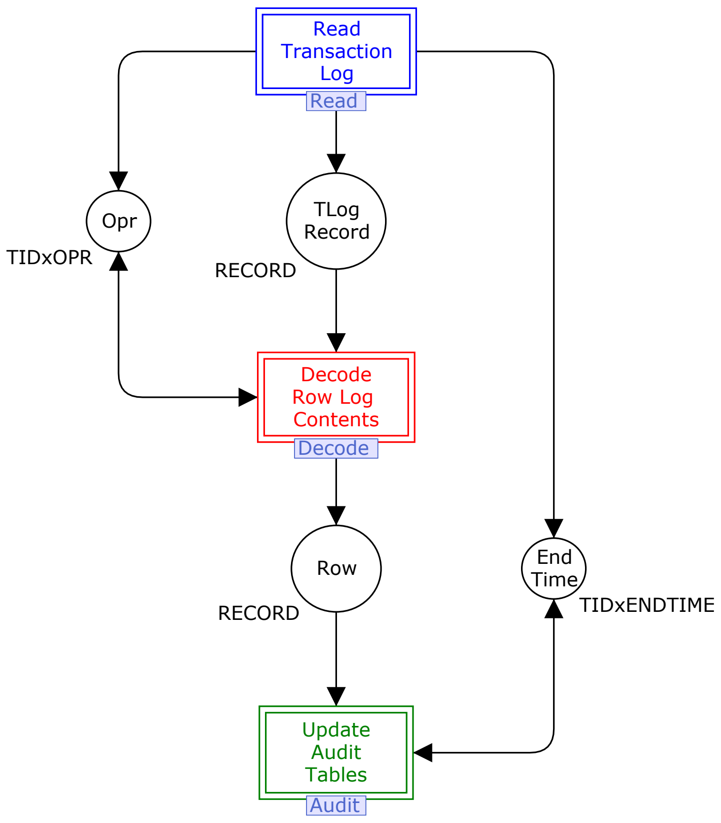


Figure 50 – The CDC process prime module

The first step of the CDC process is to read the transaction log and process the log records according to its operation type, which is accomplished in the Read module, represented by the Read Transaction Log substitution transition. The Decode module is responsible for the extraction of the modified data from the Row Log Content attribute of the record, which is represented by the Decode Row Log Contents substitution transition. Finally, in the Audit module represented by the Update Audit Tables substitution transition, the audited data is inserted in the corresponding audit table.

In addition to these substitution transitions, the prime module is also formed by four places. The place Opr is the *output socket* of the Read Transaction Log substitution transition and the *input/output socket* of the Decode Row Log Contents substitution transition. The second *output socket* of Read Transaction Log is the place End Time, which is also the *input/output socket* of Update Audit Tables. The place TLog Record, used to model a transaction log record, is the third *output socket* of Read Transaction Log and the *input socket* of Decode Row Log Contents and finally, the place Row, that models a log record after the decoding of the Row Log Contents, acts as the *output socket* of Decode Row Log Contents and *input socket* of Update Audit Tables. The `TIDxOPR` and `TIDxENDTIME` colour sets both represent the product of an integer and a string. Although they are equal, these colour sets have different representations because they represent tuples with attributes of different data types (e.g., `VARCHAR` to store the operation and `DATETIME` to store the end time):

```
colset NO = int;
colset ST = string;
colset TIDxENDTIME = product NO * ST;
colset TIDxOPR = product NO * ST;
```

The `RECORD` colour set is used to represent a record from a relational database and it is defined as the union of the different types of records used in this CDC process:

```
colset RECORD = union
     TLogRec:TLOGREC + AudRec:AUDREC + DecRec:DECREC;
```

In order to make the model more uniform, all the places that represent a relational table (e.g., an Audit Table) or individual records have this colour set. The colour sets, which are part of this union and represent the existing types of records, are described in the following sections.

4.2.3. The Read Module

The Read module (Figure 51) is responsible for the extraction and initial processing of the records of the transaction log according to their type of operation. Each transaction is composed of three or more records with different operations. The log record with the LOP_BEGIN_XACT operation marks the beginning of the transaction and holds the name of the transaction (Insert, Update or Delete), the record with the LOP_COMMIT_XACT operation is the final record of every transaction and also holds information on its end time. The LOP_INSERT_ROWS, LOP_UPDATE_ROWS and LOP_DELETE_ROWS operations belong to the log records holding the information about the actual row modification in the operational sources. A single transaction is responsible solely for one kind of operation (Insertion, Deletion or Update), but the same operation can occur in several different records.

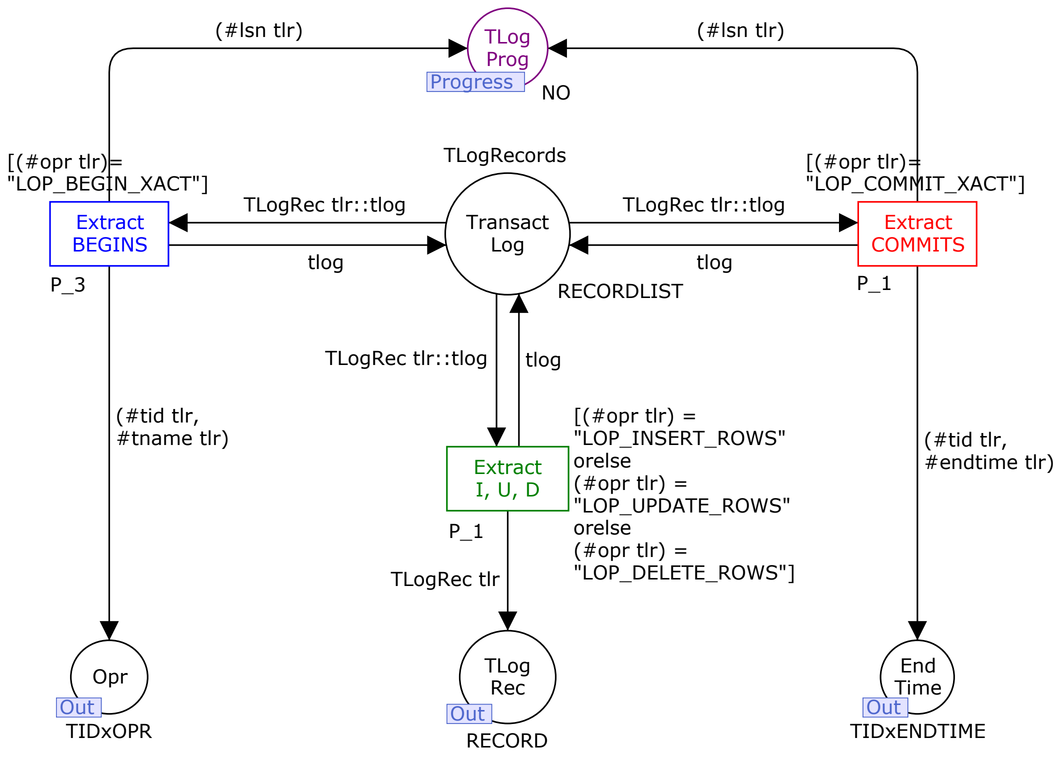


Figure 51 – The Read module

This module is composed of five places and three transitions. Three of the places – Opr, TLog Rec and End Time – are the *output ports* of this module and have been described previously. The fusion place TLog Prog of the colour set `NO` is used to manage the progress of this CDC process by saving the LSN of the last log record to be fully processed, which is necessary to resume the process should it terminate unexpectedly. The last place, Transact Log of the colour set `RECORDLIST`, is used to model the transaction log view presented previously. In spite of representing a relational table, the colour set `RECORDLIST` is used exceptionally, instead of

RECORD, so that the log records can be processed sequentially as a FIFO list. This colour set is implemented as:

```
colset RECORDLIST = list RECORD;
```

The three existing transitions are used to extract individual log records from the place Transact Log according to the type of operation of the records; the arc expression TLogRec tlr::tlog is used to extract the head of the list (i.e., a log record) into the corresponding transition, while the remaining list is returned to Transact Log through the arc expression tlog1:

```
var tlog: RECORDLIST;
var tlr : TLOGREC;
```

The colour set TLOGREC is used to model the transaction log record, a seven field record defined as:

```
colset TLOGREC = record
     lsn:NO * tid:NO * opr:ST * tname:ST *
     endtime:ST * aun:ST * rlc:ST;
```

The lsn and tid fields, with colour set NO, are integers used to represent the LSN and the Transaction ID respectively. The remaining fields – tname, opr, endtime, aun and rlc – with colour set ST, are strings that represent the Transaction Name (i.e., Insert, Update or Delete), the operation of the log record, the End Time of the transaction, the Alloc Unit Name and the Row Log Contents of the log record, respectively.

The Extract BEGINS transition is responsible for extracting the first record of each transaction, i.e., the records with the LOP_BEGIN_XACT operation, through the guard expression [(#opr tlr) = "LOP_BEGIN_XACT"]. From this point, the ID and the name of the transaction are passed to the *output port* Opr, through the arc expression (#tid tlr, #tname tlr), and the place TLog Prog is updated with the LSN of the processed record through the arc expression (#lsn tlr). The importance of this first record is to mark the beginning of a new transaction, as well as to preserve the operation of the transaction in the place Opr, so that the remaining log records (i.e., the ones that contain the Row Log Contents values) can be processed in accordance with the Decode module. The transition Extract I, U, D is responsible for the extraction of all the log records that are neither BEGINS nor COMMITS; the guard expression allows the transition to be enabled if the operation of the log record is LOP_INSERT_ROWS, LOP_MODIFY_ROWS or LOP_DELETE_ROWS. The records extracted through this transition are passed to the *output port* TLog Rec so that the information contained in the `rlc` field can be decoded in the next module.

The final piece of information needed to construct an audit record is the end time of the transaction; this information is presented in final record of every transaction, which is extracted from Transact Log through the transition Extract COMMITS. From this transition, the place End Time is updated with the timestamp of the commit operation of the transaction, as well as the transaction ID so that this timestamp can be correctly associated with the records that possess modified information (i.e., the ones extracted in the transition Extract I, U, D). This is accomplished through the arc expression `(#tid tlr, #tname tlr)`; at the same time, the progress of the CDC process is once again recorded. Note that the TLog Prog is not updated for the log records extracted in the transition Extract I, U, D as they aren't fully processed until they are used to update the corresponding audit table. In this module, the Extract BEGINS transition has the lowest priority (P_3) so that a new transaction is only processed after the current transaction's records have been fully processed in the remaining modules.

4.2.4. The Decode Module

After the transaction's LOP_BEGIN_XACT and LOP_COMMIT_XACT log records have been fully processed, the next step is to process the remaining records extracted in the previous module, i.e., the ones with the LOP_INSERT_ROWS, LOP_MODIFY_ROWS or LOP_DELETE_ROWS operation. These are the records that posses hexadecimal values in the Row Log Contents attributes, rather that 'NULL' values, and this where the information about the inserted, deleted or updated data resides. SQL Server keeps these hexadecimal values in a specific format, described in detail by Delaney et al. (2009), to facilitate the recovery of modified data so that the hex values can be transformed according to the defined format. The actual transformation of the hexadecimal values into the modified rows is out of the scope of this study, which is focused on the interpretation of the transaction log for CDC and the update of the audit tables once the modified data has been decoded. Nevertheless, in the physical design of this ETL process, these transformations would be represented by a decoding function that is simulated in this module.

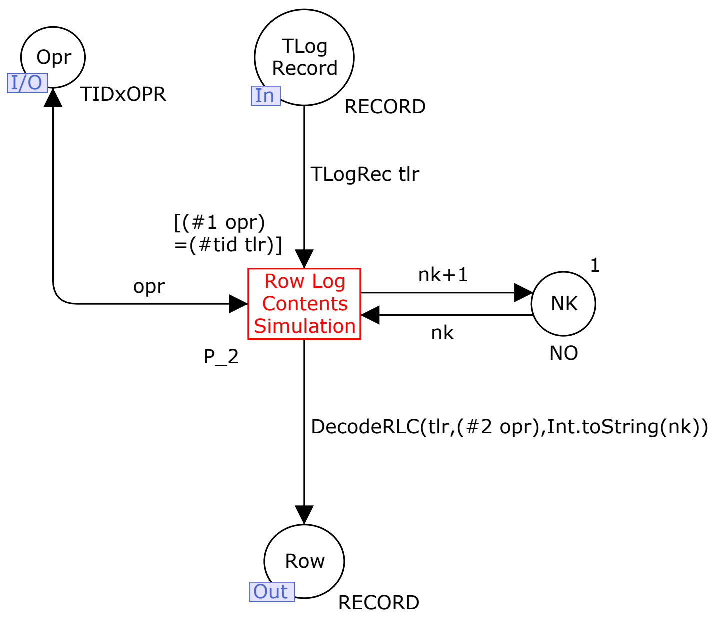


Figure 52 – The Decode Module

The Decode module (Figure 52) is used to simulate the transformation of hexadecimal values into the modified rows and is formed by a single transition and four places. The place TLog Record acts as an *input port* and is used to model the log records with modified data extracted in the previous module, while the place Row is used to model a log record after the Row Log Contents attribute has been decoded. This new record is referred to as decoded record and contains additional attributes that represent the modified row in the operational source:

```
colset DECREC = record lsn:NO * tid:NO * src:ST * opr:ST * tn:ST *
      nk:ST * atb:ST;
```

This new type of record maintains the `lsn`, `tid` and `opr` fields from the original log record. The `nk` and `atb` fields are strings used to represent the natural key and a second attribute resulting from the Row Log Contents decoding. The fields `src` and `tn`, both strings, represent the name of the operational source and the name of the audited table respectively, these values are derived from the `aun` field of the original log record. The place Opr, an *input/output port*, holds tokens with information on the transactions' names, plus the transactions' IDs, extracted from the LOP_BEGIN_XACT log records in the first module. The guard expression `[(#1 opr) = (#tid tlr)]` is used so that the correct transaction name ('Insert', 'Delete' or 'Update') is passed to the transition and used later in this module. To simulate the decoding of the Row Log Contents the place `NK`, with colour set `NO` and initial marking '1', is used together with the function `DecodeRLC` in the arc expression that leads to Row. In each marking the variable `nk` is incremented and passed to the transition trough the arc expression `nk + 1`; the incremented integer is then converted to a string and used as parameter in the `DecodeRLC` function. This function takes three parameters: the log record represented by the `tlr` variable, the name of the transaction and the incremented value of `nk`.

```
fun DecodeRLC(tlr:TLOGREC,tranName,nk) =
     1`DecRec{
          lsn= (#lsn tlr),
          tid=(#tid tlr),
          src="srcDB",
          opr=tranName,
          tn=substring((#aun tlr),4,2),
          nk="nk"^nk,
          atb="atb"^nk};
```

The new decoded record maintains the values in the log record's `lsn` and `tid` fields. The value of the `opr` field is substituted by the transaction name passed in the `tranName` variable so that the operation is displayed as 'Insert', 'Delete' or 'Update', instead of 'LOP_INSERTED_ROW', 'LOP_MODIFIED_ROW' or 'LOP_DELETED_ROW'. The name of the audited table in the operational source is extracted from the alloc unit name field (`aun`) of the original log record, which contains the schema name and the name of the table where the operation occurred. This is done through the `substring` function and the resulting string is saved into the `tn` field of the decoded record. The name of the source database can be can be obtained either by querying the operational source for the current database name, or by executing the stored procedure *sp_Msforeachdb* and compare the audited table's name with the column 'name' in either *sys.tables* or *sysobjects* tables. This will check all the tables' names in the existing databases and return the name of the database that has the requested table name. This procedure can be implemented as:

```
exec master.dbo.sp_msforeachdb
     "USE [?] SELECT db_name() FROM sysobjects WHERE name='T1'"
```

In this model, however, the execution of such a query is not possible and the `DecodeRLC` function is also used to simulate this operation by updating the `src` field. To simulate the decoding of the modified source record's natural key value, the `nk` variable passed as a function parameter is concatenated with the string "nk" so that a different natural key value (e.g. 'nk1', 'nk2', 'nk3') is created for each record. The same happens with the `atb` field.

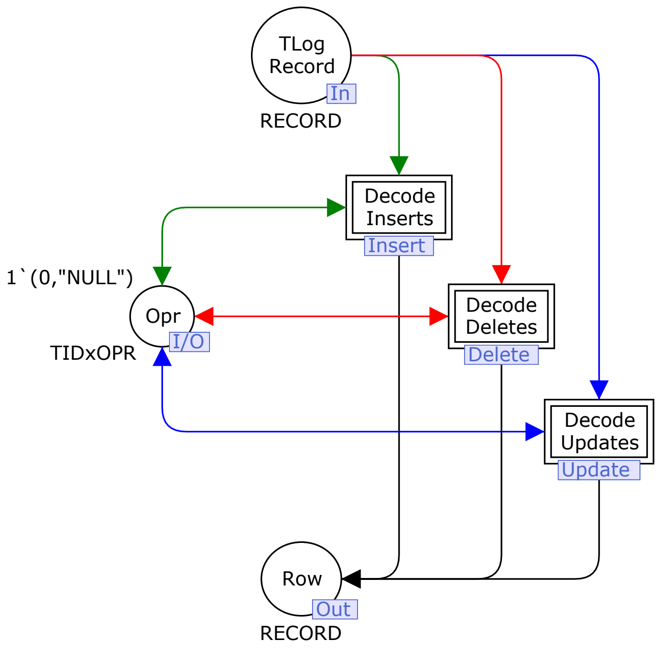


Figure 53 – The Decode implemented with Hierarchy

The simple Decode module just described is adequate to model and simulate the decoding of the hexadecimal values in the Row Log Contents attribute. However, as it has been referred, the SQL Server's transaction log is composed by four Row Log Contents attributes that may or may not contain hexadecimal information, depending on the type of operation. For this reason, the decoding functions used to extract the modified data may also vary depending on the log record's type of operation. If this was to be taken into account, an alternative version of the Decode module (Figure 53) could be used to separate the behaviour of the decoding operations into three different sub-modules (Insert, Update and Delete). In this case, a substitution transition would only activate with a combination of the corresponding operation and a log record with a transaction ID that matches the operation's transaction ID. As an example, the guard expression used for Insert module would have to be `[(#2 opr)="Insert", [(#1 opr) = (#tid tlr)]`.

4.2.5. The Audit Module

The final module, Audit, is responsible for the assignment of the transactions' timestamps to the correct decoded records, as well as their insertion in the corresponding audit tables.

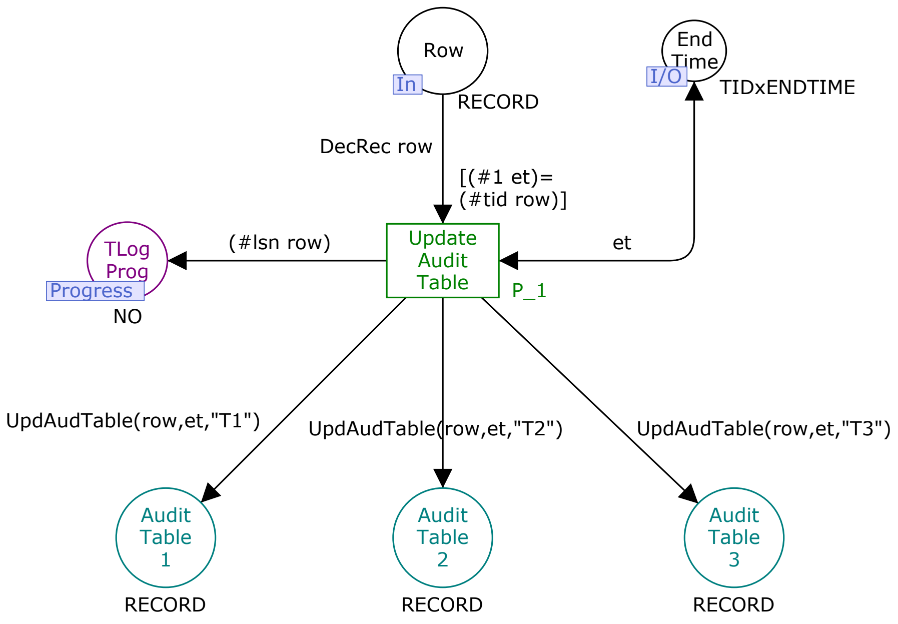


Figure 54 – The Audit module

The place Row is an *input port* and has tokens representing the decoded records, the place End Time is an *input/output port* and has tokens with information on each transactions' ID and the corresponding timestamp and the *fusion* place TLog Prog is once again used to record the log progress. The places Audit Table 1,2 and 3 are used to model the audit tables of three different relations in the operational sources and receive tokens representing the final audit records. The colour set `AUDREC` is used to model this type of records and it is defined as:

```
colset AUDREC = record src:ST * dtm:ST * opr:ST * nk:ST * atb:ST;
```

This record maintains the `src`, `opr`, `nk` and `atb` fields and has an extra field, `dtm`, which represents the timestamp of the operation in the operational source. The single transition Update Audit Table receives a decoded record through the arc expression `DecRec row` and the corresponding timestamp through the variable et in the arc that connects the place End Time with the transition. For the transition to activate, the guard expression `[(#1 et) = (#tid row)]` must be true, meaning that there is a transaction ID, associated with a timestamp in the place End Time, which matches the transaction ID of one of the decoded records in Row. The `UpdAudTable` function is responsible for creating a fresh audit record with the corresponding timestamp and inserting it in the correct audit table:

```
fun UpdAudTable(r:DECREC,et:TIDxENDTIME,tname) =
     if (#tn r)=tname
     then CreateAudRec(r,et)
     else empty
```

This function receives the decoded record `r`, the pair transaction ID – end time, represented by the variable `et` and the name of the table, represented by the variable `tname`. Then, it compares the value in the `tn` field of the decoded record with the value in the `tname` variable passed as a parameter; if they match then the correct audit table is being updated and a new audit record can be created through the function `CreateAudRec`:

```
fun CreateAudRec(r:SRCREC,et:TIDxENDTIME) =
     1`AudRec{
          src= (# src),
          dtm= (#2(et)),
          opr= (#opr r),
          nk= (#nk r),
          atb= (#nk r)};
```

The `CreateAudRec` function creates a fresh audit record; the only piece of information missing is the timestamp of the operation, passed as a parameter through the variable `et`. By using these two functions and a single transition, it is possible to insert the final audit record in the correct audit table. The final step of this module's process is to update the *fusion* place TLog Prog with the LSN of the decoded records used to generate the audit records.

4.2.6. Simulating the CDC Process

In this section the execution of the CDC model is simulated in the CPN Tools environment and described with the help of several images representing different markings. The initial marking is examined in the Read module, since this is the only one with an enabled transition in this stage (Figure 55). A small green circle, which indicates the number of tokens in the corresponding place, represents the marking of a place and the actual tokens are displayed in an adjacent green box than can be minimized for a cleaner presentation of the model.

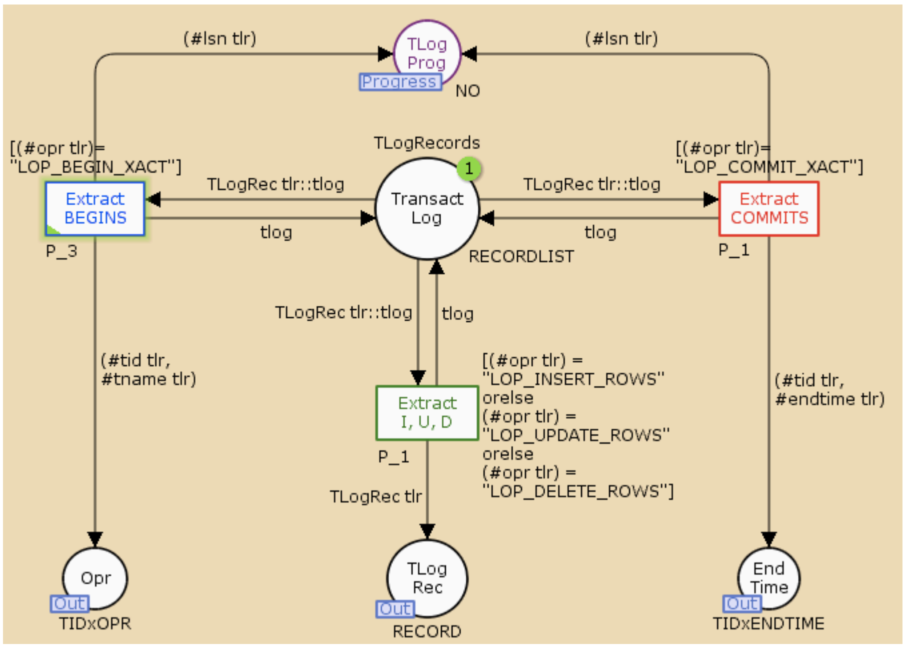


Figure 55 – The Read module - Initial marking $M0$

In this initial marking there is a single token in `Transact Log`, representing the log records that will be used for the CDC process but haven't been read from the transaction log yet. The colours of these tokens are defined in the `TLogRecords` constant, implemented as:

```
val TLogRecords=
1`[
TLogRec({lsn=1, tid=1, opr="LOP_BEGIN_XACT", tname="Insert", endtime="NULL", aun="NULL",
      rlc="NULL"}),
TLogRec({lsn=2, tid=1, opr="LOP_INSERT_ROWS", tname="NULL", endtime="NULL", aun="dbo.T1",
      rlc="0x10006C0"}),
TLogRec({lsn=3, tid=1, opr="LOP_COMMIT_XACT", tname="NULL", endtime="20/06/2012 10:20:11",
      aun="NULL", rlc="NULL"}),
TLogRec({lsn=11, tid=4, opr="LOP_BEGIN_XACT", tname="Delete", endtime="NULL", aun="NULL",
      rlc="NULL"}),
TLogRec({lsn=12, tid=4, opr="LOP_DELETE_ROWS", tname="NULL", endtime="NULL", aun="dbo.T1",
      rlc="0x10006C0"}),
TLogRec({lsn=13, tid=4, opr="LOP_DELETE_ROWS", tname="NULL", endtime="NULL", aun="dbo.T1",
      rlc="0x10006C0"}),
TLogRec({lsn=14, tid=4, opr="LOP_COMMIT_XACT", tname="NULL", endtime="20/06/2012 14:12:01",
      aun="NULL", rlc="NULL"})]
```

Fourteen log records, relative to four transactions, compose the list that represents the records in the transaction log that will be used for the CDC process. Note that only transactions 1 and 4 are displayed above. In the following simulation, the CDC process will be described for the log records used to record transaction 1 (i.e., `tid` is '1'). This transaction is recorded in the transaction log with three records. The first one is a LOP_BEGIN_XACT record that indicates the beginning of the transaction, as well as its name. The second record is a LOP_INSERT_ROW record that contains the modified information in the operational source's table and the final record is a LOP_COMMIT_XACT record indicating that the transaction successfully committed. The second transaction is also an insert, but this time in table T2 of the operational source. The third recorded transaction is responsible for updating two records on table T1, and the final recorded transaction is a delete operation of two other records, also on table T1.

In the initial marking *M0* the single enabled transition is Extract BEGINS, which is responsible for the extraction and processing of all the log records that mark the beginning of a new transaction (i.e., the records with the LOP_BEGIN_XACT operation). The first record to be processed, 'TLogRec({lsn=1, tid=1, opr="LOP_BEGIN_XACT", tname="Insert", endtime="NULL", aun="NULL", rlc="NULL"})', marks the beginning of transaction number 1 and is the current head of the log record list. When the transition is fired, the record is removed from the list and its `tid` and `tname` fields are used to update the place Opr through the arc expression `(#tid tlr, #tname tlr)` with the current transaction's ID and name/operation; at the same time progress of the CDC process is recorded in TLog Prog with the LSN of the extracted record.

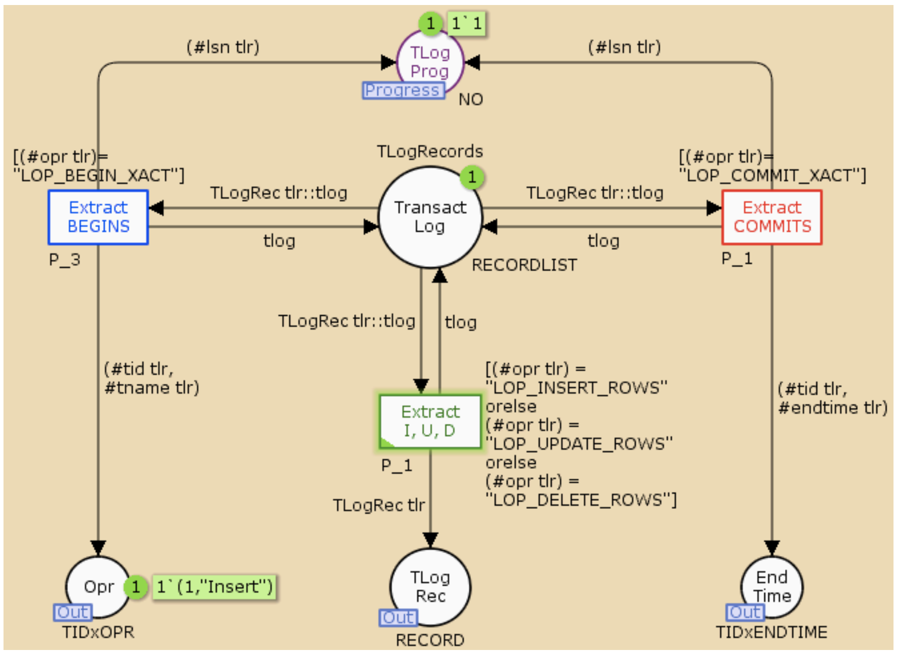


Figure 56 – The Read module - Marking *M1*

The new marking M1 (Figure 56) is reached after the Extract BEGINS transition is fired, resulting in the extraction and processing of the first log record. In this marking there is one token in Transact Log, representing the list of the remaining log records, and two new tokens, one in Opr and the other in TLog Prog. The new enabled transition is Extract I, U, D, which is responsible for the extraction of the log records that contain the information of the modified records in the operational source's tables. When this transition is fired the log record 'TLogRec({lsn=2, tid=1, opr="LOP_INSERT_ROWS", tname="NULL", endtime="NULL", aun="dbo.T1", rlc="0x10006C0"})' is removed from Transact Log into TLog Rec and the new marking *M2* is reached (Figure 57).

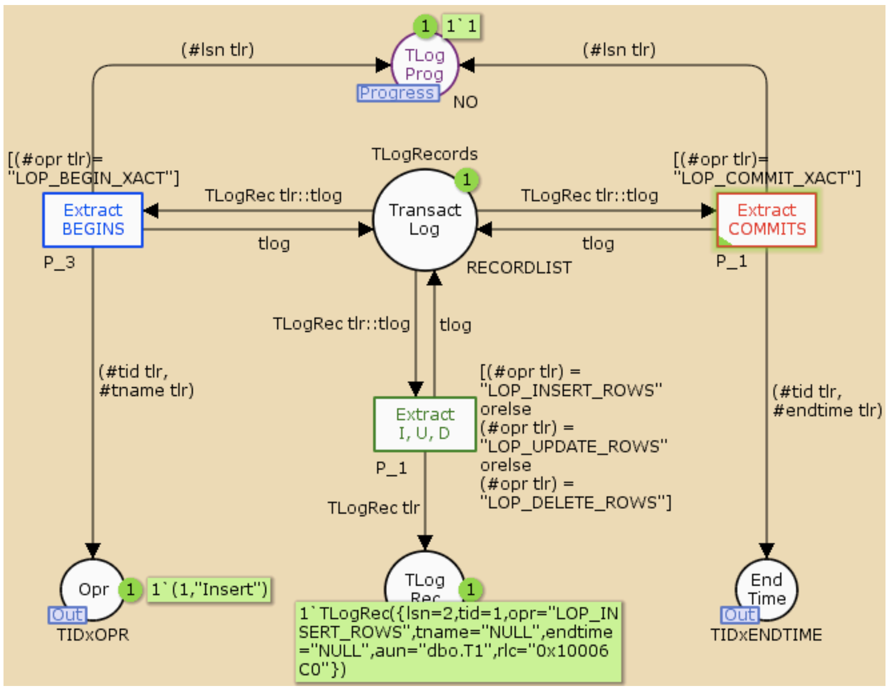


Figure 57 – The Read module - Marking *M2*

In *M2* there is only one additional token in TLog Rec, representing the previously extracted log record, while the marking of the remaining places remains unaltered. Note that the process' progress is not updated in this marking, as this type of record is not fully processed yet, this will only happen once the corresponding audit record is created and inserted in the destined audit table. Extract COMMITS is now enabled, since the third log record, the new head of the list, represents the commit operation of the first transaction. When this transaction is fired the marking *M3* (Figure 58) is reached. The record 'TLogRec({lsn=3, tid=1, opr="LOP_COMMIT_XACT", tname="NULL", endtime="20/06/2012 10:20:11", aun="NULL", rlc="NULL"})', is extracted from the list in Transact Log and its `tid` and `endtime` fields are used to update the output port End Time with the current transaction's ID and timestamp. In this marking there is an additional token in TLog Prog since the extracted log record is fully processed in this stage.

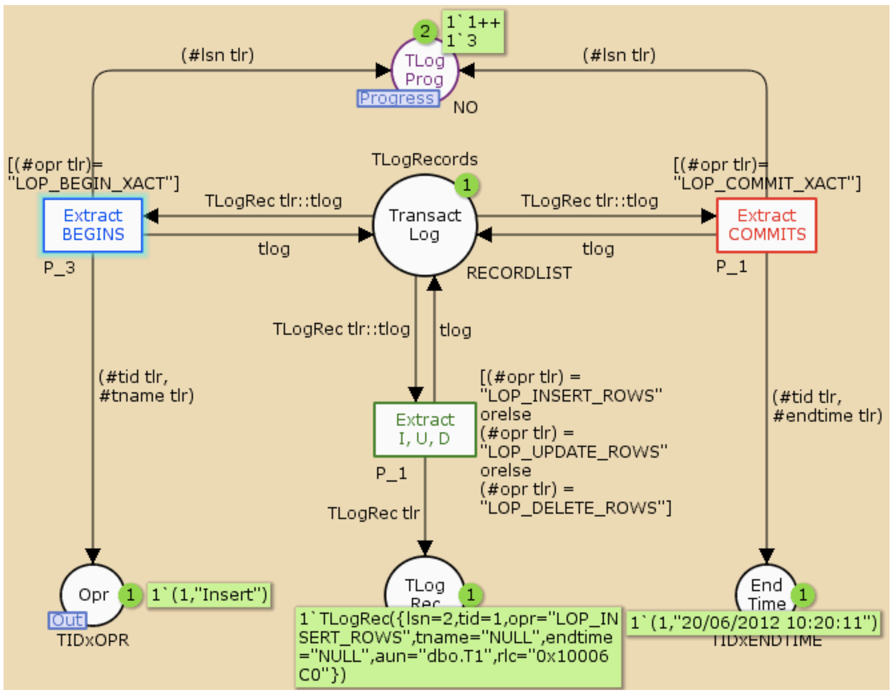


Figure 58 – The Read module Marking *M3*

In *M3* there are no enabled transitions in the Read module, because the defined priorities prevent the Extract BEGINS transition to activate and extract the following log record (i.e., `lsn`='4' and `tid`='2') until the current transaction's record (i.e., `lsn`='2' and `tid`='1') is fully processed in the remaining modules and inserted in the respective audit table. This same marking is shown in the Decode module of the CDC process (Figure 59), which is the one with next enabled transition. In *M3*, the single existing transition in the Decode module is enabled and there is one token in Opr, TLog Record and NK. The tokens in Opr and TLog Record result from the extraction and processing of the first two records in the Read module while the token in NK is used to generate different natural keys and attributes, hence simulating the decoding of the log record's `rlc` field.

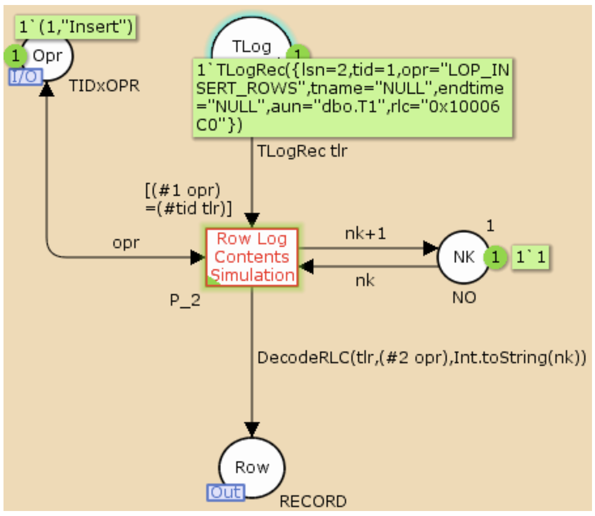

1`(1,"Insert")
1
Opr
I/O
TIDxOPR
TLog
1
1`TLogRec({lsn=2,tid=1,opr="LOP_INSERT_ROWS",tname="NULL",endtime="NULL",aun="dbo.T1",rlc="0x10006C0"})
TLogRec tlr
[(#1 opr)=(#tid tlr)]
opr
Row Log Contents Simulation
P_2
nk+1
nk
1
NK
1
1`1
NO
DecodeRLC(tlr,(#2 opr),Int.toString(nk))
Row
Out
RECORD

Figure 59 – The Decode module - Marking *M3*

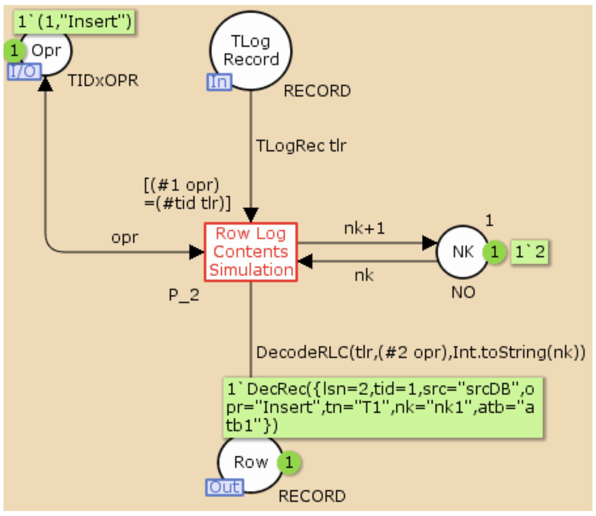

1`(1,"Insert")
1
Opr
I/O
TIDxOPR
TLog Record
In
RECORD
TLogRec tlr
[(#1 opr)=(#tid tlr)]
opr
Row Log Contents Simulation
P_2
nk+1
nk
1
NK
1
1`2
NO
DecodeRLC(tlr,(#2 opr),Int.toString(nk))
1`DecRec({lsn=2,tid=1,src="srcDB",opr="Insert",tn="T1",nk="nk1",atb="atb1"})
Row
1
Out
RECORD

Figure 60 – The Decode module - Marking *M4*

The Row Log Contents Simulation transition removes one token from TLog Record, the corresponding token from Opr and the existing token from NK. When the transition is fired (*M4*) a new decoded record is created and passed to Row through the function `DecodeRLC`. At the same time the place NK is updated with the incremented value of `nk`. The new decoded record now resided in this module's output port (Figure 60) so that it can be passed to the final Audit module.

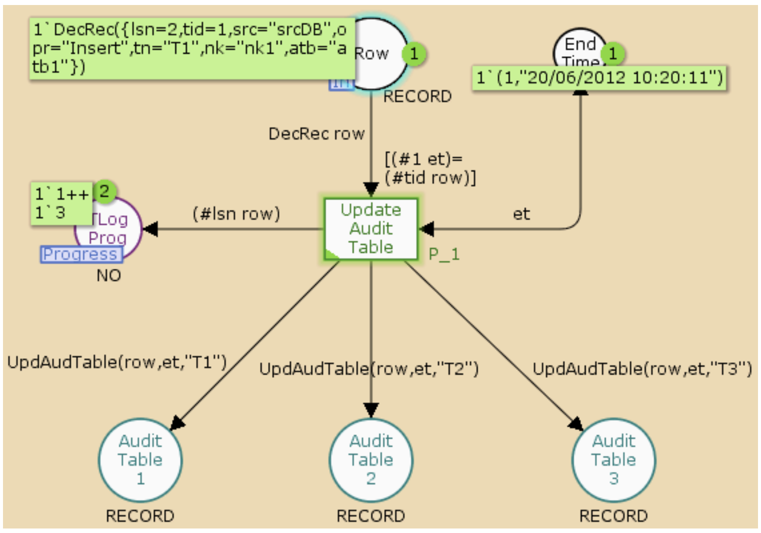


Figure 61 – The Audit module - Marking *M4*

Figure 61 shows the same marking – *M4* – in the Audit module. There are two tokens in TLog Progress representing the LSNs of the previously processed log records, one token in Row representing the decoded record processed in the previous module and one token in End Time representing the timestamp extracted from the log record with the commit operation of transaction 1. In this module there is once again only one transition, Update Audit Table. When there is a correspondence between a token in End Time and one token in Row the transition is fired and the marking *M5* is reached (Figure 62).

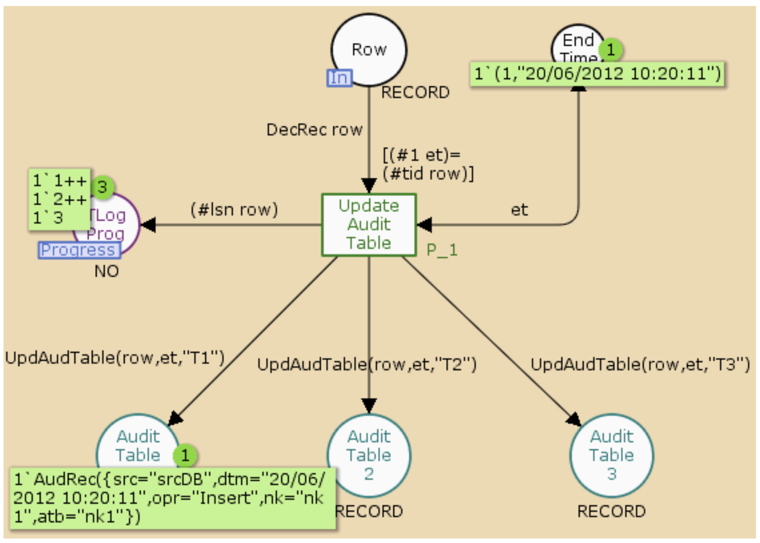


Figure 62 – The Audit module - Marking *M5*

In the new marking one token is removed from Row and used as a parameter in the `UpdAudTable` function, together with this transaction's timestamp passed to the transition in the variable et. Through this function a new audit record is created and passed to the corresponding audit table, depending on the operational source's table name stored in the decoded record's `tn` variable. In this example the new token representing an audit record is passed to the place Audit Table 1 as the Insert operation occurred in table T1. The CDC progress for the second log record (i.e., `lsn`='2' and `tid`='1') is updated in this stage with its LSN since it has been fully processed and inserted.

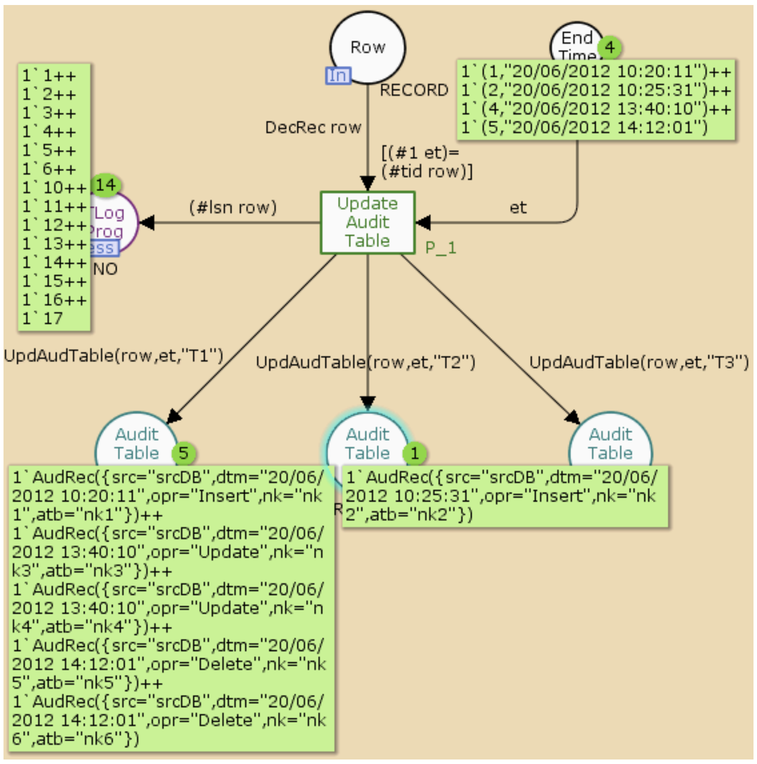

Row
In
RECORD
DecRec row
[(#1 et)=
(#tid row)]
End
Time
4
1`(1,"20/06/2012 10:20:11")++
1`(2,"20/06/2012 10:25:31")++
1`(4,"20/06/2012 13:40:10")++
1`(5,"20/06/2012 14:12:01")
1`1++
1`2++
1`3++
1`4++
1`5++
1`6++
1`10++
1`11++
1`12++
1`13++
1`14++
1`15++
1`16++
1`17
14
Log
Progress
NO
(#lsn row)
Update
Audit
Table
P_1
et
UpdAudTable(row,et,"T1")
UpdAudTable(row,et,"T2")
UpdAudTable(row,et,"T3")
Audit
Table
5
Audit
Table
1
Audit
Table
1`AudRec({src="srcDB",dtm="20/06/2012 10:20:11",opr="Insert",nk="nk1",atb="nk1"})++
1`AudRec({src="srcDB",dtm="20/06/2012 13:40:10",opr="Update",nk="nk3",atb="nk3"})++
1`AudRec({src="srcDB",dtm="20/06/2012 13:40:10",opr="Update",nk="nk4",atb="nk4"})++
1`AudRec({src="srcDB",dtm="20/06/2012 14:12:01",opr="Delete",nk="nk5",atb="nk5"})++
1`AudRec({src="srcDB",dtm="20/06/2012 14:12:01",opr="Delete",nk="nk6",atb="nk6"})
1`AudRec({src="srcDB",dtm="20/06/2012 10:25:31",opr="Insert",nk="nk2",atb="nk2"})

Figure 63 – The Audit module - Final marking

In the final marking every log record has been processed and the corresponding audit records created and inserted in the correct audit table. There are no records in Audit Table 3, a single record in Audit Table 2 and the remaining audit records have been inserted in Audit Table 1.

4.3. The Selected ETL System

The objective of the previous sections is to present the CPN models that were implemented separately for three important ETL processes: the SKP, the CDC and the SCD-H, which also contains the SKP process. For this, the behaviour of each process was explained and the corresponding model described. A simulation of each model was also performed and displayed, in order to present and analyse the behaviour of each modelled process. In this section the previously implemented models, that represent standard ETL operations, are used as independent packages or sub-modules in order to create and simulate a larger ETL system based on a practical example. The objective is to build the model for an ETL scenario through the composition of the smaller existing modules.

Let us consider the following Data Mart (DM) (Figure 64), which will be used as the practical example in which the modelling of this ETL scenario is based on.

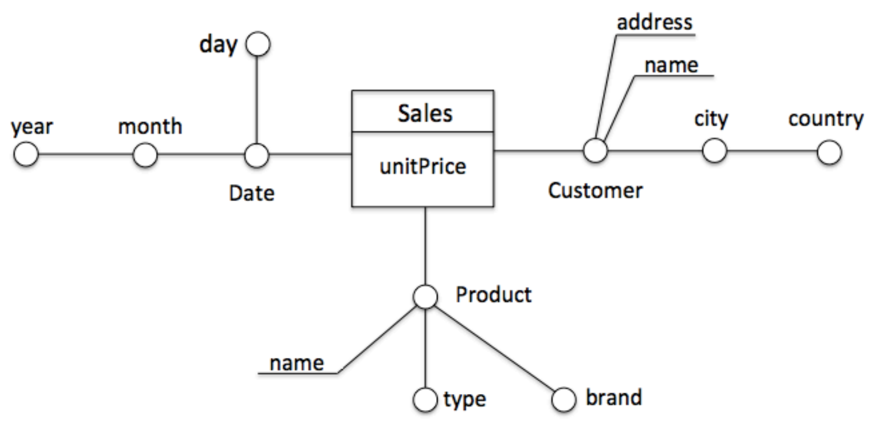


Figure 64 - Sales Data Mart Fact Schema

In Figure 64 is presented the schema for a simple Sales DM implemented with the Dimensional Fact Model, a conceptual modelling language used to support the design of Data Marts (Golfarelli and Rizzi, 2009). The Sales fact table and the dimensions Time, Customer and Product form the DM. The fact table is responsible for registering the many-to-many relationships between the existing dimensions. Therefore, each existing fact record (or primary event) aggregates the information of a product sold to a customer on a given date and the price for which the product was sold, which is represented by the `unitPrice` measure in the fact table. The Date dimension is composed by three dimensional attributes – `day`, `month` and `year` – and two hierarchies formed by the relationship of these attributes: date→day, and date→month→year. The Customer dimension is formed by two dimensional attributes, `city` and `country`, that form the customer→city→country hierarchy, and two descriptive attributes, address and name, that are used to better describe the customer dimensional attribute in the hierarchy. Finally, the Product dimension is also formed by two dimensional attributes – `type` and `brand` – but this time they form two independent hierarchies: product→type, and product→brand. The descriptive attribute name is used to give additional product information.

4.3.1. The ETL System CPN model

In order to integrate the relevant data into the Sales data mart it must initially be extracted from the operational sources. In this example it is assumed that there is a single operational source feeding the DM with data, which is audited through a transaction log file. The CDC is the first ETL process to occur, which is responsible for reading the source's transaction log and extract the relevant data into the corresponding audit tables in the DSA. Each dimension in the DM must have a corresponding audit table in the DSA. This model is based in a real-time ETL scenario, meaning that any modification in the transaction log will trigger the CDC process, which is then responsible for separating the extracted data accordingly with the existing audit tables. After the data has been extracted into the DSA it can then be loaded into the DM. In order to accomplish this all the dimensions need to be populated prior to the fact table. Because the fact records consist of a sequence of surrogate keys, and the corresponding measures, they must be the last records to be loaded into the DM. These surrogate keys are generated and mapped with the corresponding natural keys during the SCD-H process into the respective lookup tables. These same tables are also used during the SKP process, responsible for loading the fact records into the fact table, making it dependant on the existing SCD-H processes. Being so, there must exist three independent SCD-H processes that use the audited data in the Time, Customer and Product audit tables, being responsible for populating the Time, Customer and Product Dimensions. The last process to occur, the SKP, uses the audited data that is destined to the Sales fact table and is responsible for substituting the natural keys in the audited records for the surrogate keys created in the previous SCD-H processes. No data can be loaded into the Sales fact table until lookup records are made available for use in the SKP process.

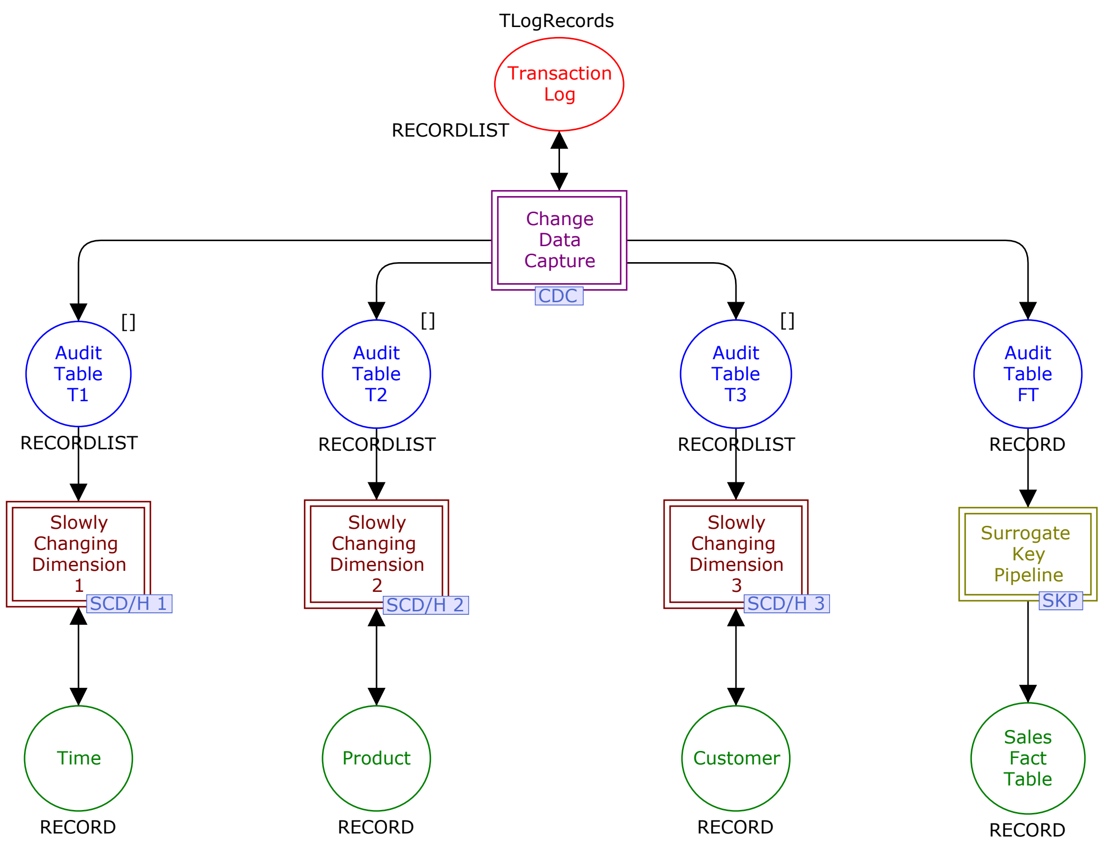


Figure 65 – The ETL system CPN model

Figure 65 shows the implemented CPN model of the ETL system used to populate the Sales DM presented previously. This CPN model is composed by nine places and five substitution transitions, each of these transitions hides the details of the sub modules presented in the previous sections that model the corresponding ETL processes. The place Transaction Log, displayed in red, is used to model the transaction log of the operational source's database that will be used to audit its data. The four blue coloured places are used to model the relational tables in the DSA. Each audit table is used to store audited data from the corresponding table in the operational source, captured through the transaction log, that will later be loaded into the respective dimension or fact table. The green coloured places are used to represent the DM's dimensions – Time, Product and Customer – and the Sales fact table. A single CDC module is used to extract information from the Transaction Log, whenever it is modified in the operational source, decode and load it into the correct audit table in the DSA. There must be an independent SCD-H process for populating each dimension. In this case three SCD-H modules are needed. To accomplish this, the original SCD-H module was cloned and used separately in this CPN model. Each of these clones can then be modified or adapted to the needs of each ETL scenario. In this case the Surrogate Key Gen sub-module, found in the Insert sub-module of each SCD-H module (Figure 66) was slightly modified.

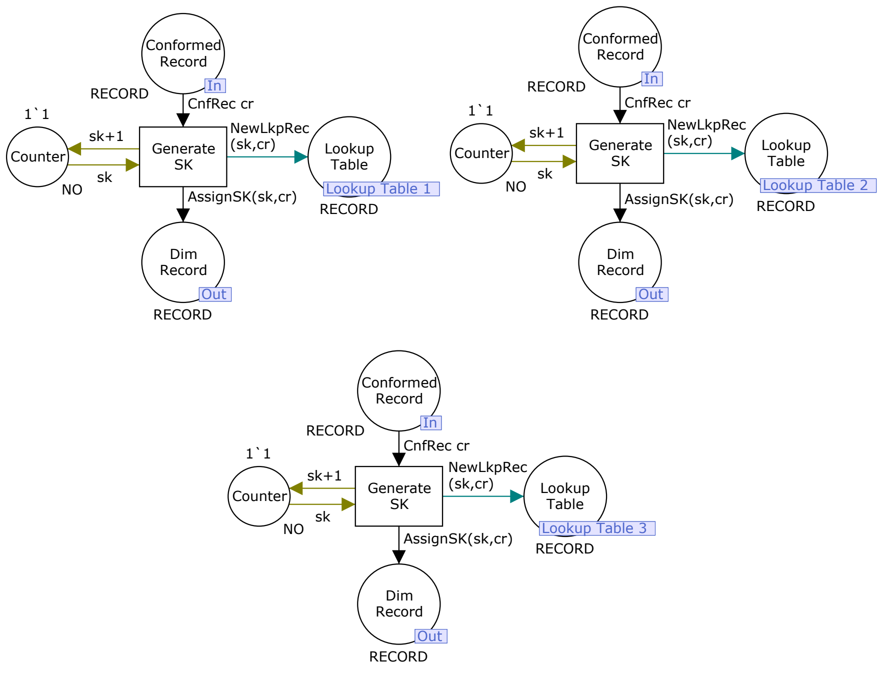


Figure 66 - Surrogate Key Gen Sub-modules of the SCD-H 1, SCD-H 2 and SCD-H 3 modules

The places representing the lookup tables managed in the SCD-H modules are now part of independent fusion sets. The Lookup Table 1 fusion set holds the lookup records corresponding to the Time dimension, the Lookup Table 2 fusion set holds the lookup records corresponding to the Product dimension and the Lookup Table 3 fusion set holds the lookup records corresponding to the Customer dimension. The place Fact Records in the DSA serves as input for the last used module, the SKP, used to load the fact records into the Sales fact table once all the dimensions have been populated. The execution of this module is dependent on the execution of the SCD-H modules because the fusion places representing the Lookup tables in this module are part of the same fusion sets defined in the SCD-H modules (Figure 67).

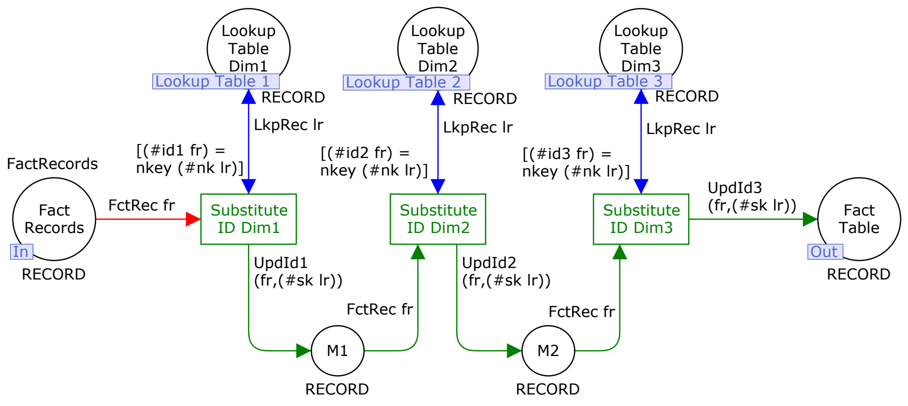


Figure 67 – The SKP Module

4.3.2. Simulating the ETL System CPN model

In the initial marking (*M0*) of this model (Figure 68), there is one token in the places Audit Table T1, T2 and T3 with the colour [] (i.e., an empty list) meaning that no records have been audited yet. There is also one token in the place Transaction Log representing the list of records in the operational source's transaction log. The initial colour of this token is defined through the `TLogRecords` value. There are a total of 14 transactions registered through 36 records in this list; all of the transactions are inserts and in each audited table in the operational source (i.e., T1, T2, T3 and FT) are inserted three different records. Below are presented the fist and the last transaction in the initial marking of the place Transaction Log:

```
val TLogRecords=
1`[TLogRec({lsn=1, tid=1, opr="LOP_BEGIN_XACT", tname="Insert", endtime="NULL",
      aun="NULL", rlc="NULL"}),
TLogRec({lsn=2, tid=1, opr="LOP_INSERT_ROWS", tname="NULL", endtime="NULL",
      aun="dbo.T1", rlc="0x10006C0"}),
TLogRec({lsn=3, tid=1, opr="LOP_COMMIT_XACT", tname="NULL", endtime="20/06/2012
      10:20:11", aun="NULL", rlc="NULL"}),
TLogRec({lsn=34, tid=14, opr="LOP_BEGIN_XACT", tname="Insert", endtime="NULL",
      aun="NULL", rlc="NULL"}),
TLogRec({lsn=35, tid=14, opr="LOP_INSERT_ROWS", tname="NULL", endtime="NULL",
      aun="dbo.FT", rlc="0x10006C0"}),
TLogRec({lsn=36, tid=14, opr="LOP_COMMIT_XACT", tname="NULL", endtime="23/06/2012
      11:42:11", aun="NULL", rlc="NULL"})]
```

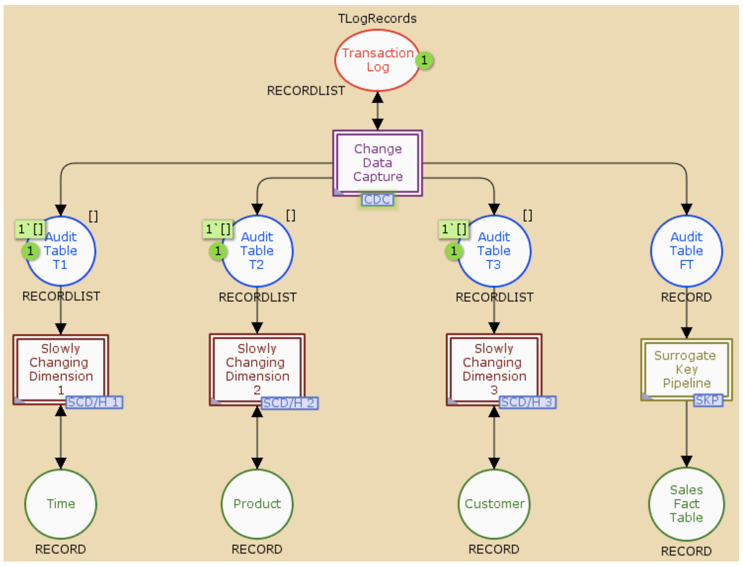
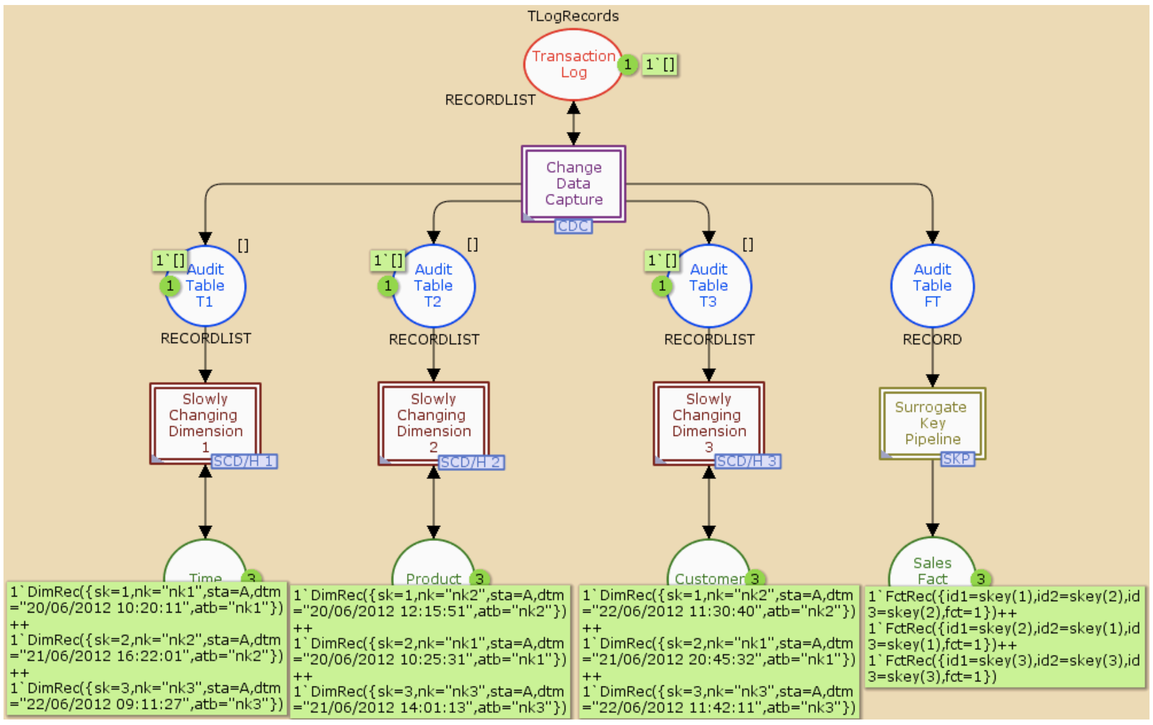


Figure 68 – The ETL System CPN model - Initial marking *M0*

In the final marking of the simulation, *M124*, there are three fully processed dimensional records in each of the DM's dimensions – Time, Product and Customer – and also three fact records, formed by three surrogate keys and one measure, in the Sales Fact Table. The existing token in the place Transaction Log has now the colour '[]', meaning that all the records have been read from the log by the CDC process.



```
1`DimRec({sk=1,nk="nk1",sta=A,dtm
="20/06/2012 10:20:11",atb="nk1"})
++
1`DimRec({sk=2,nk="nk2",sta=A,dtm
="21/06/2012 16:22:01",atb="nk2"})
++
1`DimRec({sk=3,nk="nk3",sta=A,dtm
="22/06/2012 09:11:27",atb="nk3"})
```

```
1`DimRec({sk=1,nk="nk2",sta=A,dtm
="20/06/2012 12:15:51",atb="nk2"})
++
1`DimRec({sk=2,nk="nk1",sta=A,dtm
="20/06/2012 10:25:31",atb="nk1"})
++
1`DimRec({sk=3,nk="nk3",sta=A,dtm
="21/06/2012 14:01:13",atb="nk3"})
```

```
1`DimRec({sk=1,nk="nk2",sta=A,dtm
="22/06/2012 11:30:40",atb="nk2"})
++
1`DimRec({sk=2,nk="nk1",sta=A,dtm
="21/06/2012 20:45:32",atb="nk1"})
++
1`DimRec({sk=3,nk="nk3",sta=A,dtm
="22/06/2012 11:42:11",atb="nk3"})
```

```
1`FctRec({id1=skey(1),id2=skey(2),id
3=skey(2),fct=1})++
1`FctRec({id1=skey(2),id2=skey(1),id
3=skey(1),fct=1})++
1`FctRec({id1=skey(3),id2=skey(3),id
3=skey(3),fct=1})
```

Figure 69 – The ETL System CPN model - Final marking $M124$

Chapter 5

Conclusions and Future Work

5.1. Conclusions

During the development of a DWS, it is not common to adopt conceptual modelling methodologies in the implementation of ETL systems. The main efforts concerning the modelling of a DWS are mainly applied in the conceptual design of its schema. The existing approaches, regarding the specification of ETL processes, are either not supported by a single and strong modelling language or don't allow for their validation during the conceptual design stage. This causes the success rate of DWS implementation to drop, while increasing the rather large amount of resources needed to implement the ETL system. Thus, in this dissertation a formal specification approach based on CPN for modelling and validating DWS ETL processes is presented.

To initiate this study, one of the most relevant and used cases in ETL system implementation was selected, the SKP. This case is itself one of the most adequate processes to begin with; it is a small but very important task, which allowed for a smoother contact to the, until then unknown, concepts of the CPN modelling language. The research continued with the SCD-H process. This far more complex process involved an improved application of the mechanisms of the CPN modelling language, being the hierarchical concepts the most relevant ones in this case. Hierarchy allows for a superior organization, presentation and readability of the implemented model, since it is formed of several different smaller processes itself. Due to the complexity of this process and the higher number of types of records used during its execution, when compared to the SKP, the colour sets used to represent these different records had to be modified, which implied the restructuring of the previously implemented model. The final process to be conceptually modelled was the CDC, which was modelled according to some of the already defined colour sets. Once these three processes, now turned into independent modules, were modelled, they were used as packages for modelling an ETL system scenario involving several of the considered processes. This task was made easier as the SKP model had already been updated accordingly. Still some minor alterations were made in order to relate the existing modules into an ETL system model for the Sales DM scenario presented.

The ETL CPN model presented in the last chapter is only an example of the modelling capabilities of the CPN, as it is based on a very simple and small data mart, but opens great possibilities in this area. Once the CPN meta-modules are implemented and made available as packages, an increase of the DW size and complexity will not be directly proportional to an increase in the modelling complexity of an ETL system. In fact, in this stage, it is possible to easily build and validate a CPN model for any ETL system, independently of the number of the DM that form the DW, as long as it is solely composed by the tasks modelled so far; this is simply accomplished by reusing and relating these defined packages.

5.2. Future Work

The use of the CPN modelling language for the conceptual modelling of ETL systems brings many advantages during the implementation stage of the ETL in the development of a DWS. The CPN allow the system to be built as a hierarchical structure where each process is represented by a module that can be composed of several smaller modules. This is very useful when modelling these types of systems as they can be greatly simplified by adding different abstraction levels, which makes it easier for the designer to build large models while improving their readability and understanding. A correct specification of the ETL system, with the CPN modelling language, in the conceptual design stage allows for its impact, in the development of DWS, to be previously evaluated as the whole system can be easily implemented by adding and connecting the necessary pre-defined packages. The simulations of the execution of the model have also proven to be a great feature with many advantages. By allowing a careful analysis of the data flow, through automatic and interactive simulations, it is possible to validate the implemented models. This is especially important because by applying conceptual design and validation methodologies in the development of ETL systems, the main errors can be detected and corrected in an early stage of the development. By testing and validating the ETL system prior to its implementation, the risk of failure of the whole DWS can be greatly reduced, as well as the time and monetary resources needed to correct these same errors in a more advanced stage of the system implementation.

The study initiated in this dissertation cannot be considered to be complete, since the number of different processes that usually integrate an ETL system is high. Currently, the models describe the behaviour of the SKP, SCD-H and CDC processes, as well as their integration in a real case scenario, which is based in predefined values and some basic assumptions, as is the case of the list of transaction log records and their structure. This is not enough for every real world ETL system. Thus, it needs to be adapted so that it can be applied to a wide variety of ETL scenarios. The idea is that any ETL system can be build simply by the aggregation of the necessary packages.

Bibliographic References

Abelló, A., Samos, J., and Saltor, F., YAM2: a multidimensional conceptual model extending UML. Information System, 31(6), 541-567, 2006.

Cherkasova, L., Kotov, V., and Rokicki, T., On Scalable Net Modelling of OLTP. In Petri Nets and Performance Models. Proceedings of the 5th International Workshop. IEEE Computer Society Press, 270-279, 1993.

Delaney, K. and Randal, P. and Tripp, K., Microsoft SQL Server 2008 Internals. Microsoft Press, 2009.

English, L. P., Improving data warehouse and business information quality: methods for reducing costs and increasing profits. John Wiley & Sons, Inc., New York, NY, USA, 1999.

Figueiredo, J. C. A. D. and Kristensen, L. M., Using Coloured Petri Nets to Investigate Behavioural and Performance Issues of TCP protocols. In Department of Computer Science, Aarhus University. 21-40, 1999.

Golfarelli, M., The DFM: A Conceptual Model for Data Warehouse. Encyclopedia of Data Warehousing and Mining (Second Edition), John Wang (Ed.), IGI Global, 2008.

Golfarelli, M., Rizzi, S., Data Warehouse Design: Modern Principles and Methodologies, 1 ed. McGraw-Hill, Inc., New York, NY, USA, 2009.

Inmon, W., Building the Data Warehouse , John Wiley & Sons, 1996.

Inmon, W., Building the Data Warehouse, 4th ed., Wiley Publishing, Inc, 2005.

Jensen, K., An introduction to the theoretical aspects of coloured petri nets. In A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium. Springer-Verlag, London, UK, 230–272, 1994.

Jensen, K., A brief introduction to coloured petri nets. In Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems. TACAS '97. Springer-Verlag, London, UK, 203–208, 1997.

Jensen, K., An introduction to the practical use of Coloured Petri Nets. In Lectures on Petri Nets II: Applications, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets. Springer-Verlag, London, UK, 237–292, 1998.

Jensen, K., Kristensen, M., Wells, L., Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. Int. J. Softw. Tools Technol. Transf. 9, 213–254, 2007.

Jensen, K., Krinstensen, L., Coloured Petri Nets: Modeling and Validation of Concurrent Systems. Springer, New York, NY, USA, 2009.

Kimball, R., Caserta, J., The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleanin. John Wiley & Sons, 2004.

Kristensen, M., Jensen, K., Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad hoc Networks. In SoftSpez Final Report, H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, Eds. Lecture Notes in Computer Science, vol. 3147. Springer, 248–269, 2004.

Lorentsen, L., Touvinene, A.-P., Xu, J., Modelling feature interaction patterns in Nokia mobile phones using coloured petri nets and Design/CPN. In 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01) / Kurt Jensen (Ed.). DAIMI PB-554, Aarhus University, 1–14, 2001.

Murata, T., Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77, 4, 541–580, 1989.

Oberheid, H. and Soffker, D., Cooperative arrival management in air traffic control – A Coloured Petri Net model of sequence planning. In Proceedings of the 29th international conference on Applications and Theory of Petri Nets. PETRI NETS '08. Springer-Verlag, Berlin, Heidelberg, 348-367, 2008.

Perkusich, A., de Arajo, L. M., Coelho, R. D. S., Gorgnio, K. C., Ribeiro, E. D. L. G., and Lemos, A. J. P., Design and animation of Coloured Petri Nets Models for traffic signals, 1999.

Petri, C.A., Kommunikation mit Automaten. Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik, Bonn, 1962. English translation: Technical Report RADC-TR-65-377, Griffiths Air Force Base, New York, Vol. 1, 1966.

Popova, D., On Time Petri Nets. Elektronische Informationsverarbeitung und Kybernetik 27(4): 227-244, 1991.

Project MAC (Massachusetts Institute of Technology), Record of the Project MAC Conference on Concurrent Systems and Parallel Computation: held at the Houston House of the National Academy of Science Summer Conference Centre, Woods Hole, Massachusetts, June 5-7, Association for Computing Machinery, 1970.

Simitsis, A., Modeling and managing ETL processes. In VLDB PhD Workshop. Berlin, 2003.

Vassiliadis, P., Simitisis, A., Skiadopoulos, S., Conceptual modelling for ETL processes. In Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP. DOLAP '02. ACM, New York, NY, USA, 14–21, 2002.

Xu, J. and Kuusela, J., Analysing the Execution Architecture of Mobile Phone Software with Coloured Petri Nets. In Software Tools for Technology Transfer. Springer-Verlag, 133-143, 2001.

Zaitsev, D., An Evaluation of Network Response Time using a Coloured Petri Net Model of Switched LAN. In Proceedings of the Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, October 8-11, 2004, DAIMI PB - 570 / Kurt Jensen (Ed.). 157-166, 2004.

Web References

Aarhus University, Industrial use of CPN. [Online]
Available at < http://cs.au.dk/cpnets/industrial-use/>
[Accessed on 27 June 2012]

CPN ML , Overview of CPN ML Syntax, Version 3 . 0. [Online]
Available at <http://www.daimi.au.dk/designCPN/man/Misc/CpnML.All.pdf >
[Accessed on 25 June 2012]

Standard ML of New Jersey [Online]
Available at <http://www.smlnj.org/>
[Accessed on 26 June 20