UNIVERSITÉ DE NEUCHÂTEL

FACULTÉ DES SCIENCES ÉCONOMIQUES

TOWARDS BRIDGING THE GAP BETWEEN REPRESENTATION AND FORMALISM IN THE CONTEXT OF SYSTEMS LIFE CYCLE MANAGEMENT PROCESSES

PhD Thesis submitted to the Faculty of Economics and Business

Information Management Institute

University of Neuchâtel

For the degree of PhD in Computer Science

by

Eric SIMON

Accepted by the dissertation committee:

Prof Kilian Stoffel, University of Neuchâtel, thesis director

Prof Valéry Bezençon, University of Neuchâtel, president of the jury

Dr Paul Cotofrei, University of Neuchâtel

Dr Thibault Estier, University of Lausanne

Dr Willy Picard, Poznan University of Economics, Poland

Defended on 11 December 2013

Faculté des Sciences Economiques
Avenue du 1er-Mars 26
CH-2000 Neuchâtel
www.unine.ch/seco

LETTRES ET SCIENCES HUMAINES
SCIENCES
DROIT
SCIENCES ECONOMIQUES
THEOLOGIE

UNIVERSITÉ DE NEUCHÂTEL

IMPRIMATUR POUR LA THÈSE

Towards Bridging the Gap Between Representation and Formalism in the Context of Systems Life Cycle Management Processes

Eric SIMON

UNIVERSITÉ DE NEUCHÂTEL

FACULTÉ DES SCIENCES ÉCONOMIQUES

La Faculté des sciences économiques,
sur le rapport des membres du jury

Prof. Kilian Stoffel (directeur de thèse, Université de Neuchâtel)
Prof. Valéry Bezençon (président du jury, Université de Neuchâtel)
Dr. Paul Cotofrei (Université de Neuchâtel)
Prof. Thibault Estier (Université de Lausanne)
Prof. Willy Picard (Poznan University of Economics)

Autorise l'impression de la présente thèse.

Neuchâtel, le 27 février 2014

Le doyen

Jean-Marie Grether

■ Téléphone : +41 32 718 15 00 ■ Fax : +41 32 718 15 01 ■ E-mail : secretariat.seco@unine.ch ■ www.unine.ch

Keywords: Systems life cycle management, SLCM, Petri nets, finite state automata, social protocols, business process model and notation, BPMN.

Summary

In the context of systems development life cycles (SDLC), a gap exists between the representations of the involved methodologies as process on the one hand, for example using business process model and notation (BPMN), and the formalisms that would provide the level of analysability necessary to validate the corresponding processes on the other hand beyond mere execution, for instance Petri nets. This doctoral thesis aims at bridging this gap by proposing a model in-between these two extremes that is simple yet expressive enough to be able to represent the processes, either directly or by translating BPMN diagrams to the model, while retaining enough formalism to allow its mapping to Petri nets, which enables the execution of the diagrams but also opens the door to automatic or semi-automatic validation of some properties of the systems using well-known algorithms in graph theory or methods that are specific to Petri nets. The model consists in a graphical extension of finite state automata theory, allowing synchronisation and composition of sub-processes. The model is then translated to the corresponding Petri net for execution. A further mapping, from BPMN diagrams to the model, allows a structural analysis of the described processes. Practical examples illustrate some of the possibilities and limitations of the approach, and open the discussion about possible future theoretical or practical research around these ideas.

Mots clés : gestion du cycle de vie des systèmes, SLCM, réseaux de Petri, automates à états finis, protocoles sociaux, modélisation de processus métier, BPMN.

Résumé

Dans le contexte de la gestion des cycles de vie des systèmes (SDLC), on observe un fossé entre, d'une part, les représentations utilisées pour modéliser les méthodologies sous forme de processus, par exemple en utilisant *business process model and notation* (BPMN), et d'autre part les formalismes qui offriraient les possibilités d'analyses nécessaires à la validation des processus correspondants, comme par exemple les réseaux de Petri. Cette thèse de doctorat vise à combler ce fossé en proposant un modèle quelque part entre ces deux extrêmes qui soit à la fois suffisamment simple et expressif pour représenter les processus, soit directement, soit en traduisant les diagrammes BPMN dans ce modèle, tout en conservant un niveau de formalisme suffisant pour permettre sa traduction dans des réseaux de Petri, ce qui permet également l'exécution des diagrammes, mais ouvre en outre la porte vers la validation automatique ou semi-automatique de certaines propriétés des systèmes en utilisant des algorithmes connus en théorie des graphes ou des méthodes propres aux réseaux de Petri. Le modèle consiste en une extension de la théorie des automates à états finis permettant la synchronisation et la composition de sous-processus. Le modèle est ensuite traduit dans le réseau de Petri correspondant pour exécution. Une correspondance supplémentaire, cette fois de diagrammes BPMN vers le modèle, permet une analyse structurelle des processus décrits. Des exemples pratiques illustrent quelques-unes des possibilités et limitations que présente cette approche, et ouvrent la discussion vers de possibles futures recherches théoriques ou pratiques liées à ces idées.

Everything must be made as simple as possible. But not simpler.

Albert Einstein

Acknowledgements

A Ph.D. thesis is often seen as a lonely progression towards some elusive goal. But no such adventure could be undertaken without the contribution and presence of many different people.

First, I would like to thank Kilian, for hiring me as his assistant many years ago, and for believing in me. My thanks also to Paul, for his scientific guidance, and for his friendship. I shall also remember my former office mates and colleagues Gina, Claudia, Iulian, Thorsten, Christophe, Marcelo, Dong, Tudor and Fabrizio, for the joyful atmosphere, the lunches, and the philosophical discussions we had with each other.

The work was made possible by the generous funding of both the University of Neuchâtel, and the Hasler Foundation. I am very grateful to these institutions.

A very special thought goes to my parents, and to Eléonore, for their moral support, their encouragements, and their understanding over the past several years.

Eric Simon, Neuchâtel, 11 December 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the context of systems life cycle management (SLCM), a clear gap exists between the methodologies devised to manage the development or business processes on the one hand, and the formalisms available to represent and analyse the involved processes on the other hand.

On the first side, concerned with management, the methodologies that are applied throughout the entire life cycle of a system are meant to guide managers and development teams during the successive phases of the project, rather than to guarantee a predefined, fail-proof process. They provide clear decision points, measures and guidelines to facilitate management activities such as governance, regulatory compliance, planning, budgeting and reporting.

But the methodologies are meant to be read and interpreted, not validated or executed. They are informal.

Some do *look* very formal, especially those based on the philosophy of "big design up front". The waterfall model [50] is certainly the most famous among such proposals, and it is still at the roots of some modern project management methods, although it has long been criticised for its linear nature, which it not suitable for software development, because of the essential qualities and complexity of the involved processes [10].

Other attempts take into account the iterative nature of software development, like the spiral model [7], or are based on adaptive models, like rapid application development (RAD) [26], and the many methods grouped under the general denomination of "agile" [4]. Lately, HERMES 5 [60] (Figure 2.6), the latest version of the Swiss IT project management method, incorporated these ideas as well by simplifying itself to the extreme, but the more informal the methodology, the more subject to interpretation it becomes. This is arguably a good thing for developers, and to some degree also to project

managers, but it causes the whole process to become very difficult – if not impossible – to describe, let alone to validate.

On the second side, concerned with the technical aspects of software in particular, and of systems in general, well-established formalisms exist to model the essential properties of both the actual software systems and the processes involved to produce and maintain them, or at least some of their key features[1]. In addition to the representation itself, sound mathematical foundations give the possibility to validate properties and to do correctness proofs, under certain assumptions or restrictions inherent to the object under scrutiny.

Well-known examples of interest here are automata theory in general, and the theory of virtual finite state machines and event driven finite state machines in particular. These theories allow the execution of a software specification from a formal representation. These techniques are often applied to develop safety critical applications or control software. In the same domain, Petri nets [35] are routinely applied to represent and analyse concurrent or real time systems, in order to ensure a high level of reliability. They have been applied to work flow validation as well [63, 64, 67, 41]. Even more abstract systems could be applied to do model checking, like μ-calculus [15].

Still in the context of safety critical applications in particular, and software systems with strong reliability requirements in general, such as embedded systems for example, other formalisms were developed, often with accompanying tool kits, like the Vienna Development Method (VDM), and its specification language VDM-SL and later VDM++ [6], Raise, i.e. Rigorous Approach to Industrial Software Engineering [47] and its specification language: the Raise Specification Language (RSL) [46] or the B-Method [1], derived from Z notation [2], now an ISO standard (ISO/IEC 13568:2002). Theories based on first-order logic have been appearing in publications as well [25].

But although these representations have value in the technical world, they are of limited use in the management world, i.e. in the context of software life cycle management. They can be applied, but they are often ill-suited for the task, having been built for other purposes and communities of users. Simply put, they seem to be too complex to cope with in the context of process work flows in the large.

The world of management developed its own representations of processes, like the Business Processes Model and Notation (BPMN), now in its second

[1]Here, a system is one possible outcome of a process, most notably a software or IT system. A process is the set of activities used to develop, maintain and decomission a system or to achieve any other outcome.

incarnation [58]. But BPMN, the most favoured representation, lacks the mathematical foundations necessary for the validation of properties such as those available in formal models like those mentioned above. It is essentially graphical in nature, not mathematical. It is aimed at communication, not validation, like other process notations [13].

There lies the essence of the gap between these two worlds. One side is technical, concerned with the representation and validation of properties in a well defined scope, while the other side is managerial, and although it has a long tradition of using graphical representations of all sorts, these are always aimed at people, which means they are far too fuzzy or informal to allow any automatic or semi-automatic validation.

In other words, nothing seems to exist to formalise the actual processes involved in the life cycle of systems with sound mathematical properties, allowing automatic or at least semi automatic validation, while at the same time being simple enough to use as a graphical representation in the managerial community.

A few steps towards formalisation have been taken by large organisations or administrations, which are the first to suffer from this gap, their business processes and regulations being their very nature. Documents such as the US Department of Justice's definition of SDLC or the ISO/IEC 12207 Standard for Information Technology for Systems Life Cycle Management (SLCM) [22] are among such attempts. These standards suffer from two main disadvantages resulting from their sheer complexity: first, they are very difficult to follow to the letter as such, even when accompanied by complete check lists, as is the case in the standardisation business, which somehow defeats their very purpose as tools for regulatory compliance; and second, they are very difficult, if not impossible, to validate or even to verify.

The idea presented in this thesis is to take one step further in bridging the gap between the methodologies and managerial representations on one side, with special consideration to the special case of BPMN, and the mathematical models for validation on the other side, in particular Petri nets. This gap is represented in Figure 1.1, where the relevant models are plotted in relation to two dimensions: their level of simplicity, and their level of analysability, as it can be understood from the above discussion.

To achieve this goal, the first step was to provide managers, or any other non technical stakeholder, with a very simple representation providing the bare minimum for a graphical representation of processes. This first model was inspired by previous research conducted in the domain of Picard's social protocols [36] and is described in Chapter 2. The idea was to somehow enforce a systems thinking approach to problem solving, something that is routine

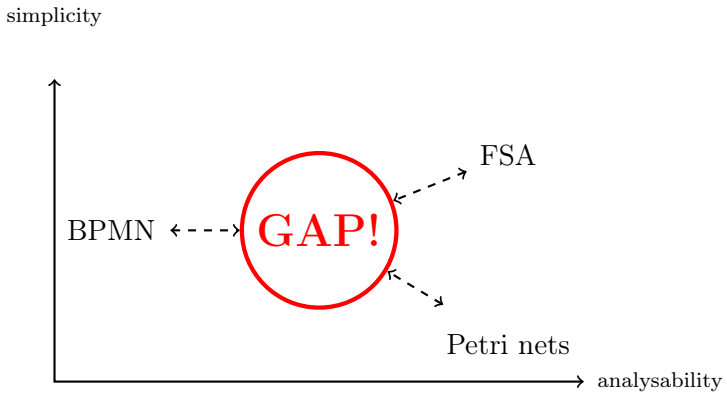


Figure 1.1 – A diagram showing the gap between several representations used in the two worlds (management and technical), with respect to the two dimensions: simplicity, and analysability, i.e. the provided system property validation possibilities. Note that SLCM is absent, as it is clear that it can be represented as BPMN, which can be seen as the frontier of the management world towards formalisation in this context.

in the technical world, but not usually emphasised on the managerial side.

The second step was to show that this model, although itself lacking the mathematical properties necessary to allow validation, could nevertheless be mapped to a well-established mathematical representation with the necessary properties, in this case Petri nets [35].

The third and final step was to show that BPMN could be mapped to our model, with some restrictions, thus effectively bridging the gap in Figure 1.1, under certain conditions. This permits, for example a purely graphical exploration and validation of the representations, the model being much simpler than BPMN in terms of the number of elements. Furthermore, the model is a very promising direction for semi-automatic analysis or validation, since all the algorithmic proofs or analysis techniques applicable on Petri nets can potentially be applied.

This thesis is laid out as follows: First, Chapter 2 presents the foundations on which the research is based: software development life cycle (SDLC), business process model and notation (BPMN), finite state automata theory (FSA), Petri nets, and Picard's social protocols [36]. Second, the model is presented in Chapter 3, in the form under which it was published in [51], together with its mapping to Petri nets, as was proposed later in [52, 53]. Those two proposals with their associated publications constitute the first

two contributions to bridging our gap. Chapter 4 shows a first example of application in the context of SDLC, using the prototype that was developed to validate the model and its mapping to Petri nets, prototype which is described in Appendix A. Then, a mapping of a subset of BPMN to our model is described as well, which constitutes the third contribution of this research. Four examples of application to real world work flows are presented and discussed in Chapter 5. This last part is the result of a joint effort with Paul Cotofrei in the context of the research funded by the Hasler Foundation (see Acknowledgements). Further research by Cotofrei was conducted to actually go one step further in this context by automatically infering business processes diagrams from legal or regulatory documents. This closely related research was published in "Business Process Modelling for Academic Virtual Organizations" [12]. Finally, possible future directions of research are investigated in the conclusion (Chapter 6).

Chapter 2

Preliminaries

This chapter introduces the concepts and theories that form the foundation of this thesis.

First, the systems development life cycle (SDLC) is introduced as the context of this research, with emphasis on the underlying methodologies.

Second, finite state automata theory (FSA) is presented, followed by the business process model and notation (BPMN) and then Petri nets, which constitute the core representations used through this thesis.

Finally, Picard's social protocols [36] are presented, as they inspired the first step towards bridging the gap between the two worlds (managerial and technical) in this context (Chapter 3).

2.1 The Systems Development Life Cycle

The systems development life cycle (SDLC) is the name given to all activities constituting the processes of building and maintaining information systems. In the specific context of engineering, it is often called the software development life cycle. It consists of both the process itself, and the methodologies applied to develop the systems.

2.1.1 A Bit of History

Many attempts have been made to formulate methodologies in order to be able to develop increasingly more complex systems in an organised and reliable manner. Most of the fashionable methodologies nowadays, in particular in the specific context of software systems development, and the more general domain of information systems life cycle management, are to some degree anchored in systems thinking, a field of systems dynamics founded in 1956

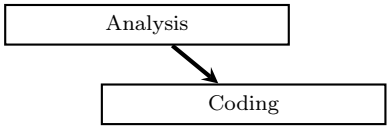


Figure 2.1 – The two steps of the software development process, as described by Royce in his seminal paper on the waterfall model.

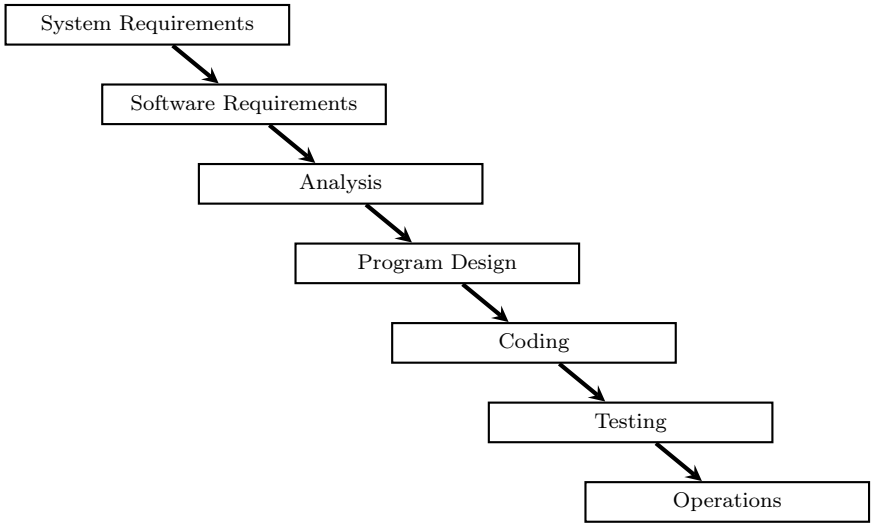


Figure 2.2 – The seven phases of the waterfall model, as described by Royce.

by MIT professor Jay Forrester for testing new ideas about social systems. This property is essential as it is the only way by which one can approach some level of reliability, considering the extreme complexity of the involved processes.

The first methodology, or model for software development, and by extension systems development, that is relevant in this context, is the waterfall model, proposed by Royce in 1970 [50]. In his seminal publication, Royce starts by conceptualising the process of software development as a process in two steps: First, analysis, and second, coding, as shown in Figure 2.1. He then proceeds in a top-down manner to break down this very simple conceptual process into a perfectly ordered sequence of seven phases, each dependent upon completion of the preceding one, as shown in Figure 2.2.

Modified versions of the waterfall model exist. Royce himself criticises his own proposal and improves his initial model with a "final model", in which he adds feedback from code testing to design, and from design back to requirements, as shown in Figure 2.3. He also emphasises the importance of the following factors: involving the customer, testing, and documentation

throughout the entire process (figure 2.4).

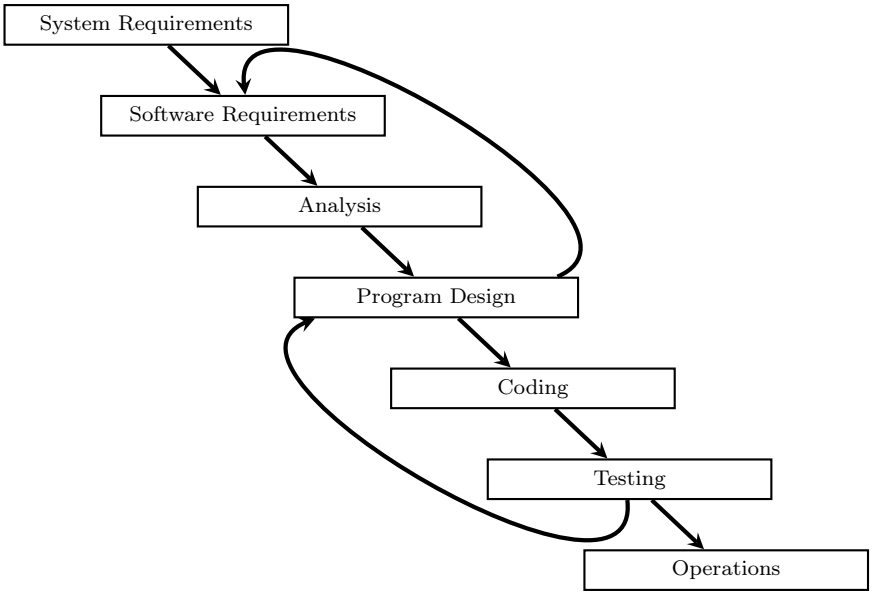


Figure 2.3 – The phases of Royce's final model, with the two critical back arrows.

Nowadays, it is argued that the model and its underlying philosophy of "big design up front", while having undeniable advantages for some complex systems, like for instance its emphasis on documentation, is suitable only for projects which are stable, in other words projects that don't have changing requirements. This is the case of equipment manufacturing for example, but the approach is not well adapted to the design and development of all modern software systems, where the requirements are very often not precise in the first place, and are subject to change.

Later on, confronted with the limits of the waterfall model for software development identified by Royce, people in the development trade tried to formulate a methodology that would better take into account the iterative nature of software or system development, and could accommodate both top-down [28] and bottom-up approaches, thus coping with the inevitable changes in requirements during a system's life cycle. Iterative development means developing in phases, starting with a design goal, and ending with the client reviewing the progress. In the mid-eighties, Boehm published an article about the spiral model, not the first to discuss iterative development in this context, but the first after Royce to explain why iteration matters so

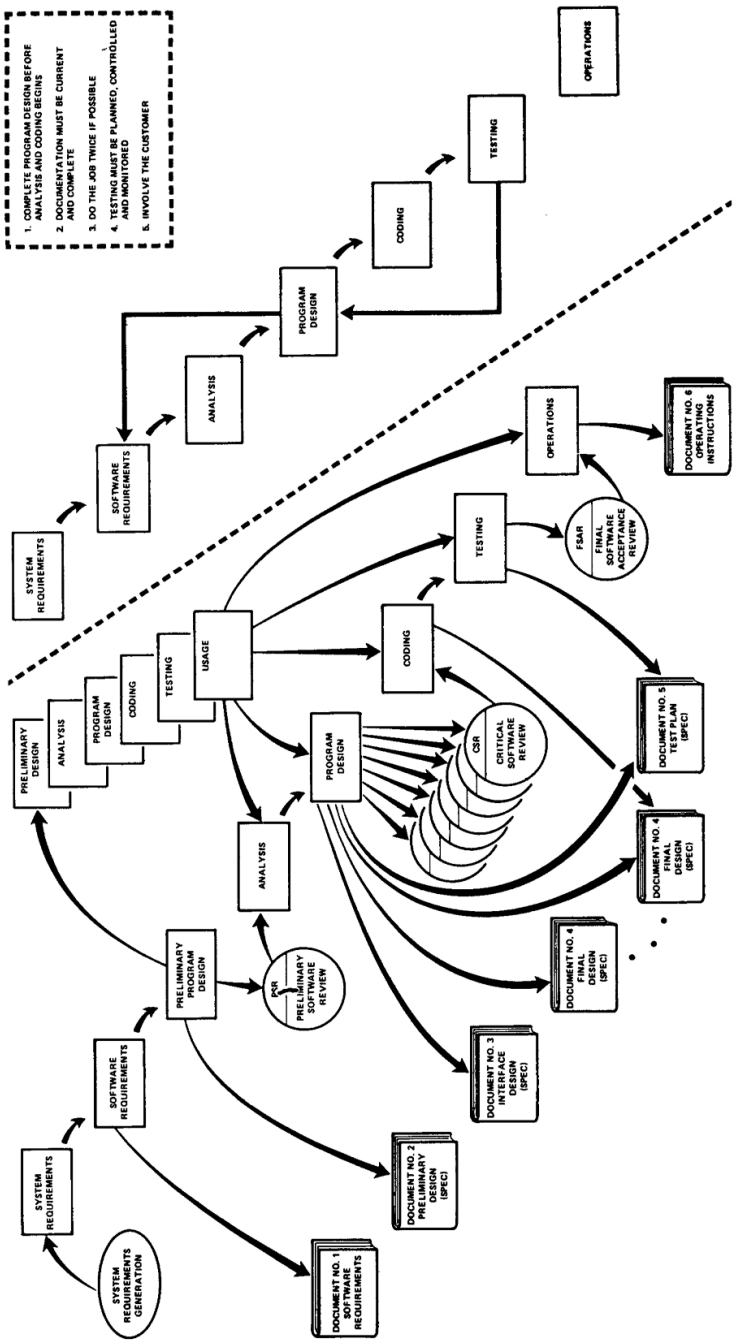


Figure 2.4 – The complete break-down of the Waterfall Model, as Royce himself described it in his 1970 publication [50] (figure 10).

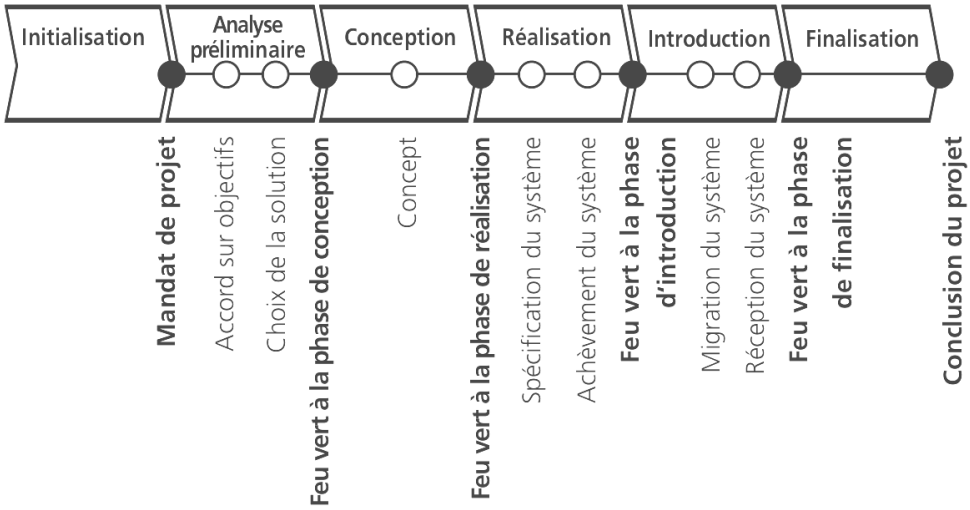


Figure 2.5 – HERMES 2003, the Swiss project management method, which was the standard for the Swiss Administration until 2013, with its phases and decision points (in French). Note the similarities with the waterfall model in Figure 2.2.

much [7]. The spiral model was adopted by the US Military, for its ability to handle large, complex and critical systems in highly inflexible administrative environments.

It is interesting to note that many "modern" IT project management methods in the industry, in particular in corporate IT, are still very much inspired by Royce's waterfall model, despite criticism. HERMES, the standard of the Swiss Federal Administration [61], for example, consists of a more or less iterative version of a waterfall in six phases, now reduced to model only four phases in its fifth version [60] (Figures 2.5 and 2.6). The tendency is to speak of "frameworks", and not "methods" or "methodologies" any more. It is always difficult to see through the haze of buzz words in corporate IT, but this might indicate a decreasing trust in the gospel that such methods are applicable, even for managers.

The V-Model, which is widespread in the German Administration, is no different in essence. It emphasizes testing, by relating each development phase to a validation equivalent on the other side of the V, as shown in Figure 2.7. In the commercial world also, the Rational Unified Process (formerly Rational, now IBM), ends up to be very similar, at least with respect to our current considerations.

A last and interesting attempt to further extend the waterfall and spiral

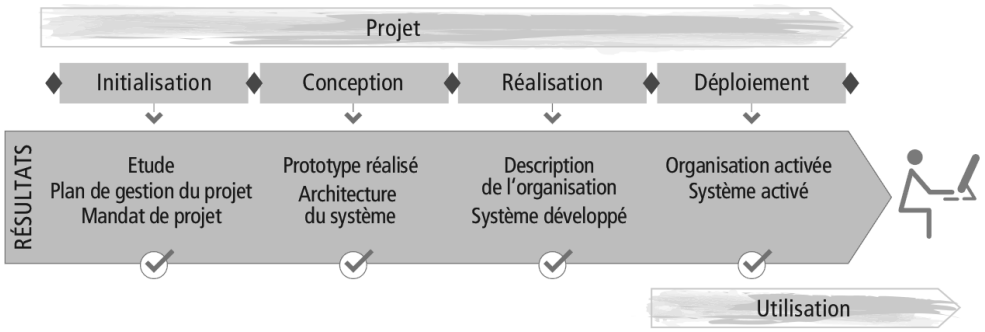


Figure 2.6 – HERMES 5, the Swiss project management method, with its phases and decision points (in French). Note the trend to reduce the number of phases and provide for more flexible frameworks, exemplified with the differences between this figure and the previous version of HERMES shown in Figure 2.5.

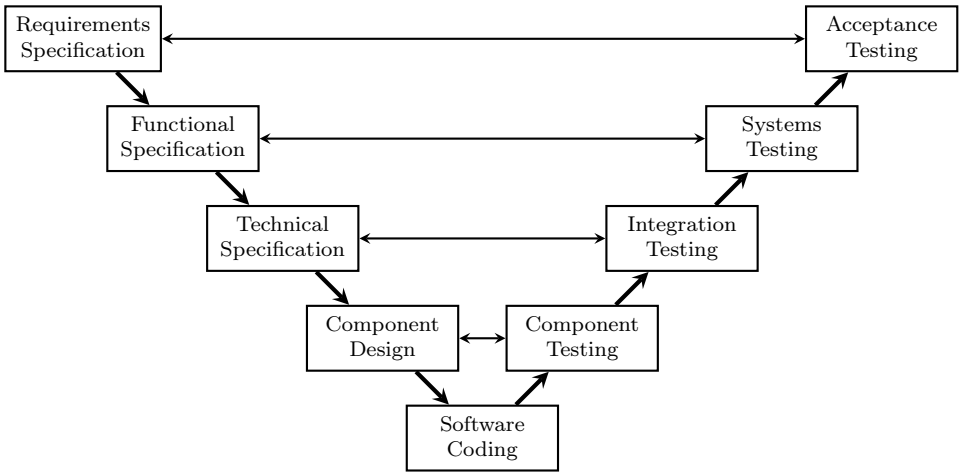


Figure 2.7 – One version of the V-Model. Note the linearity, despite the relations between activities on the right branch with those on the left branch of the V.

models led to the chaos model, defined by Raccoon[1] [45]. The idea is that all phases of the life cycle apply to all levels of a project, so a chaos strategy is devised that takes into account these properties, hence bridging the gap between managerial models (waterfall, spiral) and software development methodologies. For example, individual lines of code are treated as a level requiring design, implementation and integration, and so are functions, modules, systems and finally the whole project. This fractal structure, inspired by chaos theory, despite its obvious cynicism, is arguably closer to how software is actually developed. Indeed, it shows that the development of any sufficiently complex system consists in many interrelated levels of problem solving. Any attempt to bridge the gap between management and development has to take these properties into account, or it is doomed to fail. This essential property of software development was already identified by Brooks in his book "The Mythical Man-Month" [9] and his paper "No Silver Bullet: Essence and Accidents of Software Engineering" [10], where he advocates that software is not built, but grown.

Also during the eighties, at IBM, James Martin developed a completely different approach called rapid application development (RAD), that he formalised in a book in 1991 [26]. The focus is on delivering quality as fast as possible. The speed is achieved through the use of Computer Aided Software Engineering (CASE) tools, and the quality through the early involvement of users in the analysis and design phases. The approach enables an early prototype delivery, with reduced features, usually, and the subsequent incremental addition of more features.

Rapid Application Development consists in six core elements:

1. **Prototyping:** developing a feature-light application in a very short amount of time.

2. **Iterative development:** adding features in short life cycles, feeding the new user requirements into the next release.

3. **Time boxing:** supporting iterative development by pushing off features to future versions.

4. **Team members:** emphasizes that teams should be small and composed of experienced members.

5. **Management approach:** management should be very involved in keeping life cycles short and enforcing deadlines.

[1] pseudonym

6. **RAD tools:** development speed is more important than the cost of tools, so one should use the latest technologies.

The major drawback of this approach is the reduced scalability of the resulting systems, a consequence of the continuous enhancement of an early imperfect prototype. This requires heavy re-factoring, continuous non-regression testing and the system quickly becomes impossible to maintain. Also, it has limited value as such for this context, as it is almost entirely unstructured.

During the nineties, moving further away from the plan-driven and predictive waterfall model towards more adaptive methods, a variety of methods have been defined and grouped under the general framework denomination of agile software development, since 2001 [5]. The basic idea was to further refine iterative methods by dramatically reducing the time between releases, or milestones, now measured in weeks or even days instead of months. Another common denominator of these methods is that work is performed in a highly collaborative manner, with many concurrent activities and synchronisation of dependencies.

The well-known agile software development methods include extreme programming [4], Scrum [56], agile modelling, adaptive software development [19], Crystal methodologies, dynamic systems development method [14], feature driven development [34], lean software development [40], agile unified process, most of which are arguably modern corporate IT consultant hypes. Nevertheless, interesting for our case, as they involve well defined roles and processes.

It is worth mentioning that the same "agile" approach has been applied to the documentation and data life cycles, as was the case for Royce forty years ago.

2.1.2 Towards Formalisation

Regardless of the criticism of the methods mentioned in the previous section, it is not unusual to find one method used within another on some scale or other. Indeed, an essential property of all these models is that they define some common phases or activities, thus forming building blocks, at different levels of granularity.

For example, a developer might use the waterfall model on a very small scale for the development of his module, while an agile software development method is applied by the team for the whole project. The reverse is also highly probable, in particular where financial constraints impose a waterfall management of the project with decision points, whereas the developers work with agile methodologies, and short release cycles. So not only are

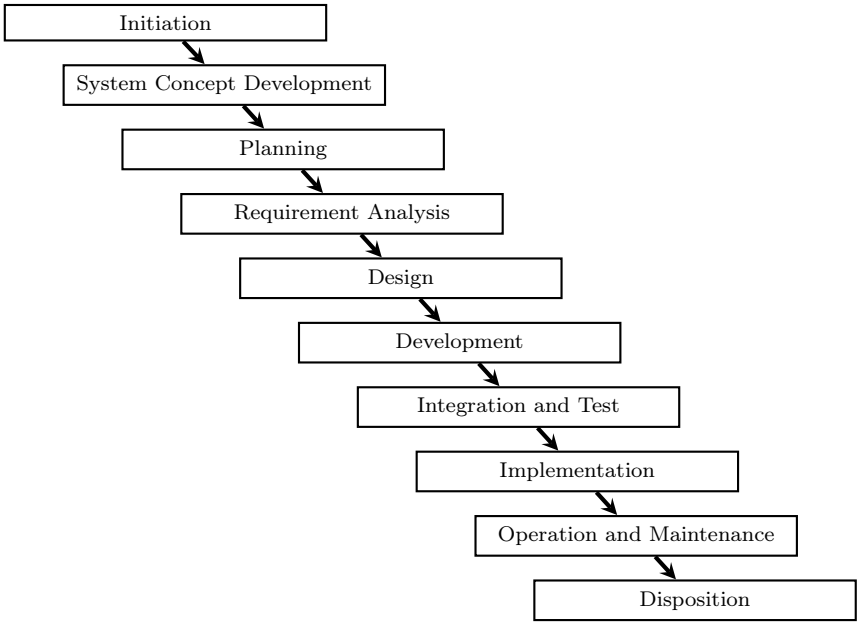


Figure 2.8 – The ten steps of the SDLC as proposed by the Department of Justice.

methodologies applicable at all levels of software development, as stated in the chaos model, they are also intertwined and combined. This shows the need for a clear definition of common subprocesses, or building blocks, in all approaches, that could be formalised and combined to create any new method, customised for the specific needs of the project managers or entire teams.

A first step in this direction has been taken by big administrations, notably by the US Department of Justice, who defined the Systems Development Life Cycle (SDLC) [59]. The involved systems approach to problem solving, made up of several phases, in this case ten, is shown in Figure 2.8.

The systems development life cycle also includes documentation as deliverables that must be generated during each phase. The same approach is used in HERMES, the Swiss IT project management method (or framework), with a linear process similar to the waterfall model, and emphasis on documentation and decision points [61, 60]. Figure 2.6 shows the phases of the method and its decision points. Note that HERMES is meant to be tailored, so by default no documentation is mandatory, apart from the description of the projet itself and its organisation (the project manual).

Although it is an essential step towards a more formal methodology, it is intended to be read, interpreted, validated and applied by human beings.

In order to have a system able to represent and validate such models automatically or semi-automatically, which is the ultimate goal of this research, a good formalisation with appropriate state of the art knowledge representation techniques is necessary, something that couldn't be found in this domain, at least not explicitly.

Following today's call for standards and best practices in software and systems development, the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) established a joint technical committee in 1987 with the scope of standardization in the field of information technology systems. This committee initiated the development of an International Standard, ISO/IEC 12207 for software life cycles [22], that led to its IEEE/EIA industry implementation in 1998 [21].

The standard establishes a top-level software life cycle architecture. The life cycle begins with an idea, or a need, that can be satisfied wholly or partly by software, and ends with the retirement of the software, as in the SDLC or HERMES. The architecture is built with a set of processes and interrelationships among them. The processes are modular, minimally coupled, and the strict responsibility of a party, or role, in the software life cycle. That means that processes are as independant from each other as possible, and reused as building blocks where possible, and that to each activity or process, a role can be assigned.

This standard is meant for specific projects, typically led by project managers using check lists for the validation of the various phases, also a core activity of HERMES for instance. It is, however, the closest thing to a formal definition to be found to this day for this specific context, so it was chosen as the starting point for the practical validation of our approach (Chapter 4).

2.2 Finite State Automata

The theory of finite state automata (FSA), or finite state machines (FSM), is a well-established model, used mainly for the representation of linear behavioural processes, as in computer programs. This section gives the basic definitions of finite state automata, as it relates to our context. More formal details, exhaustive classifications of various FSA, and their properties, are extensively described in the appropriate literature [17, 18, 11].

A finite state automaton consists in a finite set of states, and a finite set of transitions. The latter indicates a change of the machine from one state to another, and is triggered by a certain condition or event. An automaton can only be in one state at a time, and only one transition can be activated at a time (determinism). Non-deterministic finite state automata

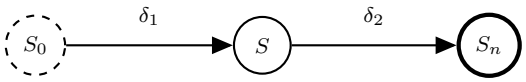


Figure 2.9 – A traditional graphical representation of a finite state automaton with three states $S = \{S_0, S, S_n\}$ and two transitions $\Delta = \{\delta_1, \delta_2\}$. S_0 is the only initial (or start) state, S is the single intermediate state, and S_n is the only final (or end, or accepting) state.

(NFA) also exist, but these are irrelevant in practical settings, such as the context of this thesis. It has been demonstrated that any non-deterministic finite state automaton can be transofrmed into a deterministic one through an algorithm: the Rabin and Scott powerset construction algorithm [44].

DEFINITION. A **finite state automaton** over an alphabet Σ is a quintuple $A = \{\Sigma, S, \Delta, I, F\}$, where

- S is a finite set of states, with $S \neq \emptyset$;
- $\Delta : S \times \Sigma \to S$ is a finite set of transitions;
- $I \subseteq S$ is the set of initial states, with $I \neq \emptyset$;
- $F \subseteq S$ is the set of final states, which can be empty.

Graphically, finite state automata are represented using circles and arrows. Initial (or start) states are represented with a dashed line, intermediate events as solid circles, and final states with a thick line, or a double line. This is the notation that was chosen in the remainder of this document. Transitions are simple arrows, sometimes with a label. Figure 2.9 shows a simple state machine with three states and two transitions as an illustration of the notation.

2.3 Business Process Model and Notation

Business process model and notation (BPMN), previously known as "Business process modelling notation", is a graphical representation of process work flows, and a widely accepted standard to model any business activities in the managerial world. It was initially developed by the Business Process Management Initiative, and has been maintained by the Open Management Group since 2005 [58].

BPMN consists in a minimal set of graphical elements, used to construct diagrams called Business Process Diagrams (BPD). There are four categories of elements: flow objects, connecting objects, swim lanes, and artefacts.

These elements are described below, and a summary with the matching graphical representations is provided in Table 2.1 (page 22).

1. **Flow objects:** Events, activities, gateways
2. **Connecting objects:** Sequence flow, message flow, association
3. **Swim lanes:** Pool, lane
4. **Artefacts:** Data object, group, annotation

Events. An event denotes something that happens (compared to something that is done = an activity). It is represented with a circle (table 2.1), and can contain icons denoting the type of event (e.g., an envelope for a message, or a lightning bolt for an error). Events can be "catching", in which case they receive something (typically a message) and start a process, or "throwing", in which case they send something (typically a message) when a process ends. There are three types of events, similar in representation to the common notation of finite state machines (section 2.2): start, intermediate and end events.

Start events. A start event is a trigger for a process. The consequence is that it can only be the "catching" type.

Intermediate events. An intermediate event represents what is happening between start and end events. Therefore, it can be "catching" or "throwing".

End events. An end event represents the end of a process. It can only be the "throwing" type. For instance, it can launch another process by sending a message to a catching event.

Activities. An activity represents the actual work being done. It is similar to the label on a transition in a finite state machine. It's graphical representation is a rectangle with rounded corners, with a label inside. The rectangle's line differs, depending on the the type of activity and may contain an icon. There are four types of activities: tasks, sub processes, transactions, and call activities.

Tasks. A task represents an atomic unit of work. The criterion for atomicity is that it cannot be broken down to a further level of detail.

Sub processes. A sub process is the mean to provide a zooming feature in a diagram. The idea is to hide/reveal additional levels of detail by collapsing/revealing the more detailed process when clicking the plus sign in the activity. A sub process must have a start event, and one or several end events, and the flows from the parent process must not cross its boundary.

Transactions. A transaction is a special kind of sub process where all contained activities must be treated as a whole. It is similar to transactions in the context of persistence (databases): all activities must all be completed to proceed further, and if any one of these fails, they must all be undone (roll-back). The graphical representation is a double border surrounding the sub process.

Call activities. A call activity is a point where a global process (or task) is reused. It provides the ability to reuse elements, combine them, and is an essential feature of process work flow modelling, like the sub process. The graphical representation is a thick border around the rectangle.

The sub process and call activities are very interesting features, as they provide composition and synchronisation mechanisms, which is essential to the systems thinking approach to problem solving which constitutes a leitmotiv of this research, and is the core idea behind the first contribution presented in Chapter 3.

Gateways. A gateway is a point of forking or merging of process paths. Its representation is a diamond shape, with an icon inside denoting the type of condition. This is similar to older popular work flow notations. There are seven different types of conditions, hence seven gateways: exclusive, event based, parallel, inclusive, exclusive event based, complex, and parallel event based.

Exclusive. The exclusive gateway provides an alternative. Only one of the paths can be taken as in an exclusive or (XOR).

Event based. The event based gateway provides a condition based on the evaluation of an event to determine the path to be taken.

Parallel. The parallel gateway provides a branching point for parallel paths without condition.

Inclusive. The inclusive gateway provides alternative flows, where all paths are evaluated.

Exclusive event based. An exclusive event based gateway is simply the combination of exclusive and event based gateways described above. The condition is the event being evaluated and the path to be taken is determined based on this. The paths are mutually exclusive.

Complex. A complex gateway can be used to model a complex synchronization mechanism. For example, a mechanism where more than two choices depend on ranges of values (if $0 < x \leq 5$ then choose path 1, else if $5 < x \leq 10$ then choose path 2, else if ... then choose path $n - 1$, else choose path n).

Parallel event based. The parallel event based gateway is the combination of the parallel and the event based gateways. Parallel processes are started based on an event, but since the parallel gateway works without condition, there is no evaluation of the event.

Connecting objects. Connecting objects, or connections, are used to connect flow objects (events, activities, gateways) with each other. They are of three types: sequences, messages, and associations.

Sequence flows. A sequence flow simply shows the order in which activities are performed. Its representation is a solid line and an arrowhead. The sequence flow may be specialised: to indicate one of several conditional flows from an activity, a diamond is added at the start of the arrow. Another special case is to denote the default flow from a decision (or activity) with conditional flows, in which case a slash (small diagonal line) is prepended to the arrow.

Message flows. A message flow is used to represent the passing of messages across boundaries (pools, see below). Its representation is a dashed or dotted line, an open circle at the start (sometimes a diamond, in which case it is not to be confused with the continuous line of the conditional sequence flow), and an empty arrowhead at its end. A message flow is to be used strictly across pools, and can never be used within the same pool.

Associations. An association is used to associate an artefact or text to a flow object. Its representation is a dotted line. An association can include a direction, in which case an arrowhead is added: toward the artefact to represent an output, from the artefact to represent an input, or both to indicate it is an input as well as an output.

Swim lanes. Swim lanes are a way to organize elements into categories. They are represented as rectangles with names, separated by solid lines. The only way to link elements from one swim lane to another is by message flow. Swim lanes are of two types: pools and lanes.

Pools. A pool groups the major participants of a process. Usually, it is used to separate different organisations. It is represented as a rectangle with a label. A pool can contain one or more lanes, like an actual swimming pool. Visually, a pool can be open, showing all internal details, or collapsed to hide the detail, in which case it is only a rectangle spanning the entire diagram.

Lanes. A lane groups activities inside a pool according to function or role. It is represented as a rectangle spanning the entire width of the pool (or height if the diagram is vertically arranged). A lane contains the flow objects, connecting objects and artefacts.

Artefacts. Artefacts are available to bring more information into the diagram, thus rendering it more readable. There are three pre-defined artefacts: data objects, groups and annotations.

Data objects. Data objects show which data is required or produced in an activity. Its representation is a cornered sheet, like a document in many file systems explorers. It is often used for documentation artefacts.

Groups. A group is a visual grouping of activities. It does not affect the flow in any way. Its representation is a rectangle, with rounded corners and dashed or dotted lines.

Annotations. An annotation does not affect the flow either, but it brings more information about flow objects, for example a comment. Its representation is a big left bracket spanning the entire text of the annotation.

A simple example of a BPMN diagram is depicted in Figure 2.10 on page 23. It is important to note, that BPMN permits the extension of the basic set presented here with other artefacts, if deemed necessary.

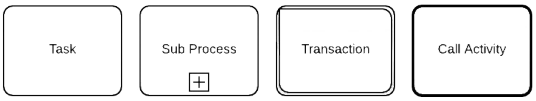
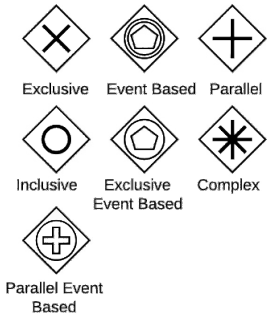
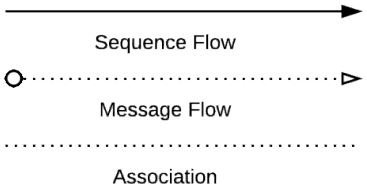
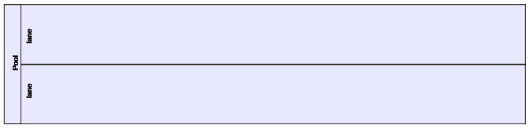
Category	Elements	Graphical notations
Flow objects	Events	
	Activities	
	Gateways	
Connecting objects		
Swim lanes		
Artefacts		

Table 2.1 – BPMN Elements

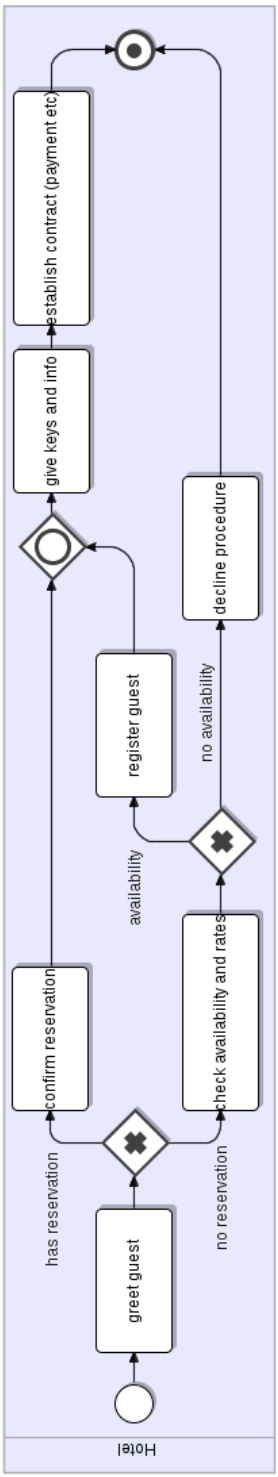


Figure 2.10 – An example of a BPMN diagram model: a simplified hotel check-in process.

2.4 Petri Nets

Petri nets are a well-established formalism used for the modelling and analysis of concurrent systems [35]. This section gives a basic definition of Petri nets and their relevant associated concepts: multisets, pre-/post-sets, marking, firing, computation. More detailed or various definitions can be found in the extensive literature [35, 48, 49, 29, 63].

DEFINITION. A **multiset**, over a non-empty finite set S, is a mapping defined as $m : S \to N$. The non-negative number $m(s)$, where $s \in S$, represents the number of occurrences of the element s in the multiset m. The set of all the multisets over S is denoted by $\mathbb{N}^S$.

DEFINITION. A **Petri net** is a tuple $N = (P, T, F, W)$, where

- P is a finite set of places and T is a finite set of transitions, such that $P \cap T = \emptyset$;
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation;
- $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the weight function for N, such that $W_{(x,y)} = 0$ iff $(x, y) \notin F$.

The Petri nets defined above are also known as place-transition Petri nets (or P/T Petri nets).

DEFINITION. If $x \in P \cup T$, then:

- The **pre-set** of x is the set: $\bullet x = \{y | (y, x) \in F\}$;
- The **post-set** of x is the set: $x\bullet = \{y | (x, y) \in F\}$.

DEFINITION. A **marking** of a Petri net $N = (P, T, F, W)$ is a mapping $M : P \to \mathbb{N}$. It assigns a number of tokens to each place $p \in P$.

DEFINITION. A **marked Petri net** is a pair $\mu = (N; M)$, where N is a Petri net and M is a marking of N.

DEFINITION. Let $N = (P, T, F, W)$ be a Petri net, M a marking of N and $t \in T$ a transition from N:

- t is enabled in marking M if $W_{(p,t)} \leq M_{(p)}, \forall p \in P$;
- If t is enabled in marking M, then t can fire, yielding a new marking M' , where $M'_{(p)} = M_{(p)} - W_{(p,t)} + W_{(t,p)}, \forall p \in P$.

The firing rule can be extended to sequences of transitions:

DEFINITION. A **computation**, or maximal execution sequence, is a finite execution sequence which ends with a marking in which no transition is enabled, or an infinite execution sequence.

DEFINITION. A marking M' is **reachable in one step** from another marking M if there exists a transition $t \in T$ such that $M \rightarrow_T M'$. By extension to a sequence of transitions, a marking M' is **reachable** from another marking M if there exists a transition sequence $\sigma \in T^*$ such that $M \rightarrow_T^* M'$, where $\rightarrow_T^*$ denotes the reflexive transitive closure of the transition relation $\rightarrow_T$.

DEFINITION. The **set of reachable markings** $R(\mu)$ of a marked Petri net μ from an initial marking M_0 is the set of all reachable markings from M_0: $R(\mu) = \{M' | M_0 \rightarrow_T^* M'\}$

DEFINITION. The **reachability graph**, or **state space**, of a marked Petri net μ is the transition relation $\rightarrow_T$ restricted to its reachable markings $R(\mu)$.

Graphically, a Petri net is represented as a graph of places and transitions. Places are represented as circles, and transitions as rectangles with inbound and outbound arrows. Both are labelled. The inbound, and outbound arrows show the number of tokens that are respectively consumed, and produced by the transition. Tokens in places are usually represented by small circles or bullets, or by a number in brackets or parenthesis.

Figure 2.11 shows the graphical representation which was chosen for the remainder of this document, and which is also what is being generated by the prototype described in Appendix A.

Formally the Petri net N in Figure 2.11 is defined as :

$$N = (P, T, F, W)$$

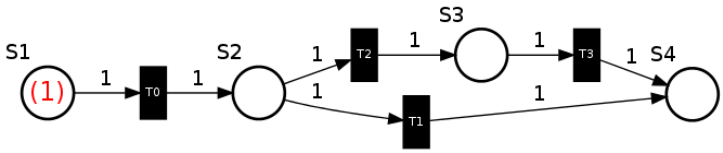
where

$$P = \{S1, S2, S3, S4\}$$

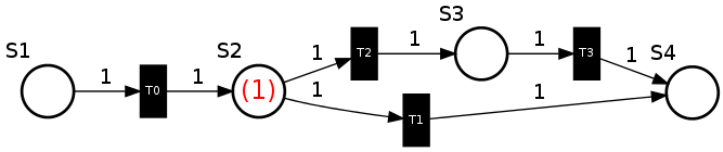
$$T = \{T0, T1, T2, T3\}$$

$$F = \{(S1, T0), (T0, S2), (S2, T1), (T1, S4), (S2, T2), (T2, S3), (S3, T3), (T3, S4)\}$$

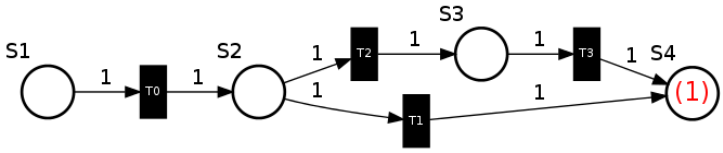
The weight function W assigns 1 to a couple (Si, Tj) iff $(Si, Tj) \in F$, otherwise 0.



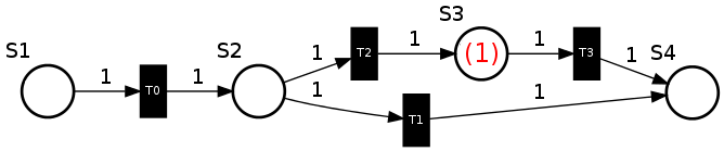
The initial state, with a single token in S1. The marking is $M_0 = \{S1 = 1, S2 = 0, S3 = 0, S4 = 0\}$. Only transition T0 is enabled.



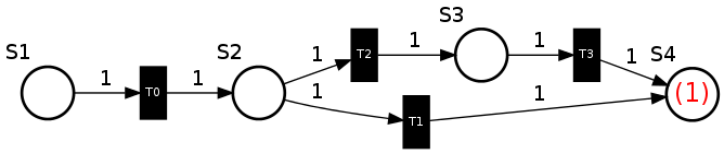
After transition T0 has fired, the token is in S2. The marking is $M_0 = \{S1 = 0, S2 = 1, S3 = 0, S4 = 0\}$. Both transitions T1 and T2 are enabled.



First possibility: after transition T1 has fired, the token is in place S4. The marking is $M_0 = \{S1 = 0, S2 = 0, S3 = 0, S4 = 1\}$. No transition is enabled, the maximal execution sequence has been reached, so the computation ends.



Second possibility: After transition T2 has fired, the token is in place S3. The marking is $M_0 = \{S1 = 0, S2 = 0, S3 = 1, S4 = 0\}$. Transition T3 is enabled.



After transition T3 has fired, the token is in place S4. The marking is $M_0 = \{S1 = 0, S2 = 0, S3 = 0, S4 = 1\}$. No transition is enabled, the maximal execution sequence has been reached, so the computation ends.

Figure 2.11 – An example of a graphical representation of a simple Petri net, in its successive possible states depending on which transition fires.

2.5 Social Protocols

Social protocols were proposed in 2006 by Willy Picard as a model for adaptive human collaboration processes [36]. They are presented in this section as they formed the starting point for our research.

The social protocols provide a formalism to model interactions between humans. The formal definition of a social protocol, given in Picard's publication [36], can be summarised as the following abstract concepts, presented in a bottom-up approach:

DEFINITION. An **action** a is the execution of a task.

DEFINITION. A **role** r is a label.

DEFINITION. A **behavioural unit** is a group $BU = R \times A$. For example: "The manager writes the final report" is a behavioural unit where the role is "manager" and the action is "write the final report".

DEFINITION. A **state** s is a label associated with a given situation in a collaborative process.

DEFINITION. A **transition** is the combination of a starting (source) state and an ending (destination) state with a behavioral unit: $t = (bu, s_{source}, s_{destination})$.

Once these preliminary definitions are in place, one can define a social protocol as the set of all states and transitions.

Picard adds a desirability function Δ, adding external constraints on the execution of the behavioural units, thus specifying the probability that the transition will be activated. The social protocol is then defined as:

DEFINITION. A **social protocol** is a sextuple

$$\Sigma = \{S, S^{start}, S^{end}, R, A, \Delta\}$$

where

- S is the set of all states s.
- S^{start} is the set of starting (source) states s_{source}, with $S^{start} \subset S$.
- S^{end} is the set of end (destination) states $s_{destination}$, with $S^{end} \subset S$
- $S^{start} \cap S^{end} = \emptyset$.

- R is the set of roles r.
- A is the set of actions a.
- $\Delta : T \to [0, 1]$ is the set of desirability functions associated with the behavioural units formed by the elements of R and A.

This forms a (non-deterministic) finite state automaton as described in Section 2.2, with the addition of some external constraint on the choice of transitions provided by the desirability function.

The interpretation of a social protocol is that the process is moving from state to state via the execution of behavioural units. But one has to be careful that a behavioural unit may only be executed by a collaborator labelled with the appropriate role.

This last property is very interesting as it allows for the modelling of SDLC processes as described in documents such as the DoD definition of SDLC [59], the IEC/IEEE 12207 document [22] or HERMES [61, 60], already discussed in Section 2.1.2.

The desirability function is an interesting idea, as it allows for the modelling, for example, that the transitions leading to success are more desirable than the transitions leading to failure. Moreover, it provides a mechanism to model negotiation.

Negotiation consists in changing the desirability function, or even changing the structure of a model. This is deemed necessary because human systems are constantly changing and adapting. For instance, with experience, actors in a process can observe that a new transition is needed, or that some transitions are all of a sudden, less "desirable" than others.

It is interesting to note at this point that the desirability function as such disappeared from the model in subsequent publications by the same author [37, 38, 39]. It influenced our initial approach in a very significant manner; in fact, it was the main reason why we chose Picard's social protocols in the first place, although it became irrelevant in the course of our research as well, as we focused on providing a representation that is as simple as possible, and the potential tailoring of processes is outside the scope of the model itself. It could nevertheless be an important research path to pursue, in relation to our context, as it is an essential feature of human activity, as Picard states [36].

Chapter 3

The Model

This chapter presents the first step of this thesis' contribution, i.e. the model to represent SDLC processes. Although it was inspired initially by the social protocols presented in Chapter 2, it consists merely in a finite state machine (FSM), to which two extensions are made to represent parallel synchronised sub-processes on the one hand, and allow for scalability on the other hand. The first section presents the argument that led from Picard's idea to this very simple model. The subsequent two sections explain the two necessary extensions, followed by a formal definition of the model, with the necessary definitions. The last section presents the mapping of the model to Petri nets.

3.1 Argument

As already stated in the introduction, when it comes to dealing with processes, there is a clear gap between the management world and formalism. The essence of this gap lies in the different points of view and the needs of each of these worlds.

Management is concerned with processes in terms of people, results, and possibly representation as a means to communicate ideas. There is usually no need to scientifically prove the models, as common agreement usually suffices. In management, BPMN is one de facto process modelling standard. These models are agreed on, stored for later use or used as ERP workflows or other management systems.

On the other hand, formalisations provide these proofs. The mathematically sound representations, such as Petri nets or finite state automata, allow many interesting analysis or validation of properties at the formal level. They provide both a good and compact graphical representation, but if one was to attempt to model processes such as those in the DoJ SDLC document, the

IEC/IEEE 12207 document [22], or even HERMES 5, it would be impossible to ensure that the model really copes with reality, as it quickly become far too complex to be tractable.

One reason for this is the difficulty for people in general, and people in the management world in particular, to apply systems thinking to the formalisations with theoretical models such as finite state automata or Petri nets, like they would with BPMN if they were well-trained business analysts for example. Indeed, BPMN provides the mechanism for composing processes with sub-processes which facilitates the breaking down of the problem into sub-problems: the sub-process activity flow object (table 2.1 on page 22).

Finite state automata and Petri nets lack this composition (or zooming) mechanism. Nothing prevents people from breaking down problems into models using those formalisms. This is actually what experienced people involved in modelling would do. But the representation do not encourage such methodologies, and we argue that this is a serious limitation for this particular context. Recursive finite state machines [3] do provide it, and even in a very elegant generalised way. But they are very difficult to use in this context, due to the strict definitions permitting recursion, which in turn is not necessarily the way people think, unless they have been trained as mathematicians or computer scientists.

The same lack of simplicity de facto rules out the concepts of abstraction and folding in Petri nets [16] as well.

Moreover, the very nature of systems life cycle management processes, or any business process for that matter, implies the possibility to have parallel, concurrent activities. BPMN allows this, with swim lanes and coordination mechanisms through messages (table 2.1 on page 22). Petri nets too, as they were designed exactly for such problem, but they are still quite complex, and not commonly seen in the management world. Finite state automata simply lack this property, so they are not applicable.

The first step in starting to bridge the gap was to come up with a simple enough model that could represent processes with both the necessary properties of composition and of concurrency, while retaining at least some mathematical properties to allow for the validation of the properties of the work flow at a graphical or algorithmic level. Social protocols (section 2.5) seemed to be a very promising idea, as they were based on mathematical foundations, in this case finite state automata theory, and they were aimed at modelling and even negotiating processes. They suffer from the same limits as the finite state automata in general, i.e. the lack of composition and concurrency mechanisms.

Note that Professor Picard later addressed this last problem by pro-

posing to model his social protocols with coloured Petri nets [23]. This is described in details in "Modelling Multithreaded Social Protocols with Coloured Petri Nets" [38]. The purpose of the development of social protocols is to model human-to-human interactions over a network from the ground up, while retaining a strong mathematical foundation, and explore the possibilities offered by successive refinements of the model, for example a direct structural validation [37] or the application to agile paradigms of software development [39].

The use of coloured Petri nets in this context implies that the semantics of what is being produced is attached to the states themselves. Adding types would permit the explicit representation of what is being produced, a specific document or a software module for example, but this information would have to be explicit in the simple representation of the processes, and that would add complexity and burden to the users, by making the proposed model more complicated [52, 53]. This possible approach was therefore not investigated in our research, not because it was not interesting or not applicable, but because it was too anchored in the formalism world and not exactly aligned with our line of research, and it seemed more promising to attempt to bridge the gap between the two worlds with a simpler and more pragmatic proposal.

After all, if one is to attempt to bridge the gap between two very different cultures, as is the case here, it would be a mistake to try to force the aspects of one culture on the other, and hope that by convincing managers that they should behave like scientists, their problems would be solved. To have better chances of success, it is better to try to retain as much of the characteristics of the two cultures, and connect these somehow. In this case, this means retaining the pragmatism of the managerial world on one side, and the formalism of the technical world on the other.

This led to the first idea, that was to retain the simplicity of finite state automata theory as it was applied in Picard's first model of social protocols, i.e. with actions and roles, and extend it with only the two necessary properties to model composition, and concurrency. This is the essence of the model presented in this chapter: component state machines (CSM) and scalable state machines (SSM), which together provide a simpler representation than BPMN, with less elements, but with a substratum of mathematical properties, that we will argue in Section 3.5 is not lost by the addition of the extensions [51], and therefore allows a mapping to Petri nets, which in turn permits interesting validation of properties that the model itself doesn't allow.

Petri nets are not the only theory with sufficient mathematical soundness to allow analysis and validation of properties. Other representational

possibilities for our model were investigated, namely: conceptual graphs [55] and description logic. The two theories possess sound mathematical foundations, and are expressive languages, but this very last feature is what rules them out. Representing processes using conceptual graphs or description logic (or any other first-order predicate logic for that matter), adds to the burden of representing dynamic activities explicitly, while Petri nets were designed from the ground up to fulfil that requirement. In addition, Petri nets have a very pragmatic graphical representation that fully reflects their formal foundations.

The theory of μ-calculus, on the other hand, is applied to model checking for systems [24], in particular those non-deterministic systems operating in "critical contexts" [15]. The modal logic of μ-calculus is even more complicated than first-order predicate logic though, so it rules out this approach too, although it would be interesting to investigate its differences and potential advantages over Petri nets for validation.

It would nevertheless be interesting to investigate more complex formalisms, like coloured Petri nets, if only to verify our assumptions about simplicity as a means of bridging the gap between the two cultures.

3.2 Synchronisation

To model development processes in the context of SDLC, an essential feature to include is some sort of synchronisation mechanism between several parallel, or concurrent, sub-processes or activities. Finite state automata theory doesn't include such possibilities: there is no AND on the nodes, only ORs (alternatives).

The first extension is a simple "rendezvous" type of synchronisation mechanism to model parallel activities and hold subsequent dependant activities until all pre-requisites are completed.

Note that it is different from the "rendezvous" as it is defined in the context of parallelism. The latter synchronises threads (processes) that continue after having met at that point. In our case the parallel processes themselves do not continue, it is the whole process that is held until the parallel processes all reach a final state.

Graphically, the synchronisation is represented by an AND in a rectangle between the set of final states of the synchronised automata on the one hand, and the set of source states of the dependent activities on the other hand.

It is very important at this point to realise that the extension is not formally part of the mathematical model itself, but only a convenient way to represent concurrent activities using finite state automata. There is no such

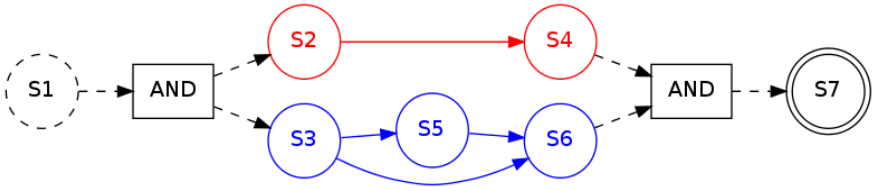


Figure 3.1 – Synchronisation between two parallel automata: The AND on the left means that both red and blue state machines are started after S1. The AND on the right means that both final states of the red and blue state machines must be reached in order to proceed to the final state S7 of the whole process. The dashed arrows emphasise the fact that those arcs are not part of the model.

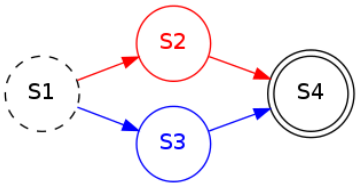


Figure 3.2 – Representation of an alternative (an OR) in finite state automata theory. The alternative is already part of the model, as only one outbound path can be chosen at any given state, in this case either the red or the blue one.

thing as an AND in finite state automata theory. The arrows are therefore dashed, to emphasise the fact that the arcs are not transitions of the state machine, and do not carry meaning about a role or an action.

Figure 3.1 shows an example of such a synchronisation, and the chosen notation, consistent with the traditional notation for finite state automata described in Section 2.2. In this particular example, a whole automaton is composed of states S1 to S7, and can itself be subdivided into two components: the red and blue automata respectively. The synchronisation is twofold: first S2 and S3 are triggered by an AND with two outbound arrows, and second S4 and S6 must both be reached in order to proceed to the final state S7. The red and blue state machines are therefore parallel activities that can be conducted separately, but must both be finished before the whole process can continue.

An arbitrary number of sub-processes $n > 1$ can be synchronised in this manner.

Note that there is no need for an OR (XOR), as an alternative is already part of the state machine theory: a state with multiple outbound arcs represents such an alternative, dependant on the conditions expressed by the arcs, as in Figure 3.2.

It could be argued that non-deterministic FSA already include the AND, as they allow for *zero* to n transitions from a given state by definition, but

we find this representation to be simpler to use and to map to Petri nets, as one clearly sees, graphically, the difference between an AND and an OR (XOR), which is not the case in non-deterministic FSA.

3.3 Components

If one sets aside recursive state machines [3], finite state automata theory doesn't provide any clear decomposition and composition mechanisms allowing for the design of sub-processes separately and their subsequent combination into more complex processes. This mechanism is an essential feature of process modelling though, as it allows a systemic approach to problem solving, which is the only way to have a chance to cope with the sheer complexity of the processes involved in this context. So this feature has to be added, but in the simplest possible way: by allowing the replacement of an arc by another finite state automaton representing the corresponding sub-process.

This extension can be seen as a sort of recursive definition, similar in essence to that of recursive state machines [3], which allows for the replacement of activities (arcs) in an FSM by another FSM. In other words, it allows the viewing of an activity as a simple transition, or as a more detailed process.

It was found to be much simpler than the definition of a recursive state machine though. It is based only on the semantic equivalence of two different state machines seen at different levels of detail. In this sense it is more appropriate to speak of "zooming" or "scalability", so we use the terminology "scalable state machine" (SSM) and "component state machine" (CSM) in the remainder of this document when we refer to the extended FSM used in our model.

Intuitively, as a prelude to a more formal definition of the model, there are two conditions to satisfy for this property to hold:

1. The source states (entry nodes), and the destination states (exit nodes), must be unequivocally identified and respectively identical for the considered transition T and the finite state automaton representing the sub-process it represents. This corresponds to the requirement of a well-defined interface in the definition of a recursive state machine, with the limitation that we consider only one entry state and one exit state.

2. The role associated with an arc must be in phase with the "overall role" of the component automaton. A certain freedom exists as to how to define this "overall role", depending on the semantics of the process.

3.4 Formal Definition

DEFINITION. A **scalable state machine** is a finite state automaton, i.e. a quintuple

$$\Sigma = (S, s_{src}, S_{dst}, A, T)$$

where

- S is the set of all states.
- $s_{src} \in S$ is the source state, or entry state.
- $s_{dst} \subseteq S$ is the set of destination states, or exit states.
- A is the set of activities.
- $T : S \times A \to S$ is the state-transition function.

DEFINITION. An **activity** $\alpha \in A$ is a tuple (r, a), where

- A **role** r is a label; it identifies the users or entities that perform the action. A role can be an abstract thing played by many users (a team) or software agents.
- An **action** a is the execution of a task. Usually, in the context of SDLC or BPMN, such a task can be interpreted as the production of a deliverable, such as a document or a piece of software.

Note the similarity with the formal definition of social protocols given in Section 2.5. Only the desirability function is absent, as it was abandoned, for reasons explained in Section 3.1.

DEFINITION. A **component state machine** is a scalable state machine Σ with exactly one source state s_{src} and exactly one destination state $s_{dst} \in S$.

DEFINITION. A scalable state machine $\Sigma = (S, s_{src}, S_{dst}, A, T)$ is **semantically equivalent** to another scalable state machine Σ', in which the transition $t = (s^t_{src}, \alpha^t, s^t_{dst}) \in T$ has been replaced by a component state machine $\Upsilon = (s^\Upsilon_{src}, S^\Upsilon_{dst}, A^\Upsilon, T^\Upsilon)$ if and only if the following two conditions hold:

1. Source and destination states are respectively identical for the CSM Υ and the transition t it replaces: $s^t_{src} = s^\Upsilon_{src}$ and $s^t_{dst} = s^\Upsilon_{dst}$.
2. A meaningful overall role r^Υ and a meaningful overall action a^Υ of the SSM Υ that corresponds to the role r^t and action a^t in activity $\alpha^t = (r^t, a^t)$ that can be defined, respectively.

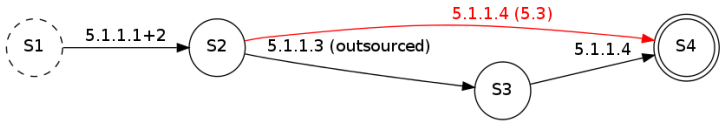
Condition 1 implies the uniqueness of the destination state. An arc has exactly one source and one destination state. In effect, the overall process can be represented as a SSM, with multiple destination states, but any activity itself is only represented by a CSM, i.e. it must have only one destination state. This makes sense in our context. A process can very well have two distinct destination states, one for success and one for failure for example, but any arc in itself must lead to a well-defined, unique state, or else the composition doesn't make sense.

In general, condition 2 always holds as a role is only a label and one can define a meta-role, capturing the semantics of all the roles involved in the new SSM Υ replacing transition t , or simply use the last activity completing the task, i.e. the last arc reaching the destination state of Υ. For example, if three roles are involved on the arcs of a process, namely "software architect", "developer" and "tester", the collapsed transition t representing the SSM Υ could have "development team" as an overall role, or simply "tester" as the person responsible for the acceptance of the process (the last transition). The same is true for the action.

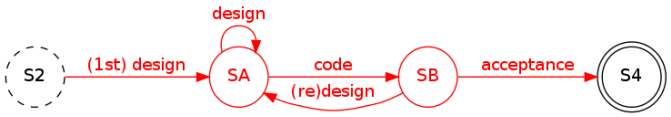
Figure 3.3 illustrates the process of composing SSM in the model defined above, using real examples from the IEC/IEEE 12207 document [22].

Using the full definition of recursive state machines would allow for much more flexible compositions. However, for the problem at hand, it is not necessary. The semantics behind the SLCM or BPMN activities is about producing deliverables or results, and alternative destination states can therefore always be combined into one single ending state which signifies the acceptance of the considered process after some possibilities that would otherwise be considered as final states have been reached. In other words, a scalable state machine (with multiple destination states) can be transformed into a component state machine (with only one destination state) with no loss of functionality. In the success/failure example, one possibility consists in defining another state meaning the end of the project and making it the only final state.

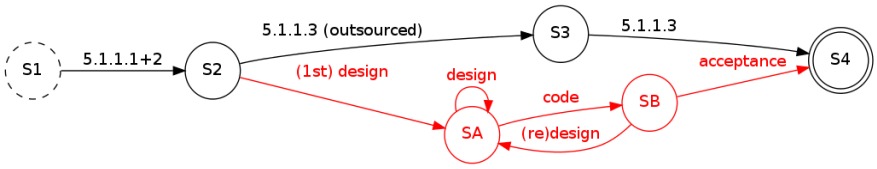
Furthermore, as stated in the introduction and the argument in Section 3.1, the idea is to keep the formalism as simple as possible, with the aim to allow non-specialists to model processes relevant to their business, and combining automata with interfaces of multiple entry and exit states recursively is far more complicated, even though it is mathematically much more elegant and powerful than the proposed model.



Part 1: a FSM at a low level of detail.



Part 2: the red arc between S2 and S4 in Part 1 can be specified at a higher level of detail as this FSM, a CSM.



Part 3: The resulting FSM after replacing the red arc in part 1 by the CSM in Part 2.

Figure 3.3 – The process of replacing a transition by a more detailed FSM (a CSM) inside a FSM. The arc in red in Part 1 is replaced by the whole CSM of Part 2, which respects the two necessary conditions about the boundary states and roles, and the result is a more detailed view of the process, shown in Part 3.
NB: the inversion of the different paths and colours is due to the automatic layout engine of the prototype used to represent and execute the processes.

3.5 Mapping the Model to Petri Nets

This section describes the second step of the contribution to bridge the gap between our two worlds. The previous sections of this chapter describe a simple model based on finite state automata theory and extended with concurrency and composition mechanisms. This model is sufficient for the context of modelling the simple processes of SDLC as described in the IEC/IEEE 12207 document [22]. The strength of the model, its simplicity and pragmatism, is also its weakness. The two extensions are not really part of the mathematical system itself, and in order to represent explicitly these processes and validate their properties, or do correctness proofs, another model is needed that satisfies the following properties:

1. The new model must take into account the possibility of parallel activities, while retaining the other features of concurrency and composition.

2. A mapping must exist between the proposed model and the new model that doesn't break any condition or include new information requiring human intervention.

Petri Nets, presented in Chapter 2, Section 2.4, satisfy these conditions, and in their simplest form.

It is also well known that finite state automata can be seen as a special case of Petri nets, where each transition in the automaton, represented as a box in the case of Petri nets, can have one and only one incoming arc and one and only one outgoing arc.

The transformation, or translation, of a finite state automaton into a Petri net, which we call "the mapping" is then trivial, as it goes from the more specific to the more general model. This is exemplified in Figure 3.4, using the same example as in Figure 3.3.

What remains to be proven is that the two extensions presented in Chapter 3, Sections 3.2, and 3.3 respectively, do not break any property that would prevent the mapping, which is the purpose of the following sections of this chapter.

3.5.1 Synchronisation

What happens when we represent the synchronisation mechanism ("rendezvous" or AND) between the set of the destination states of some SSM on one side and the triggering of initial states of other SSM on the other side as a Petri net? In the theory of Petri nets, such a synchronisation is simply a transition (a rectangle) with multiple incoming arcs, one for each of the

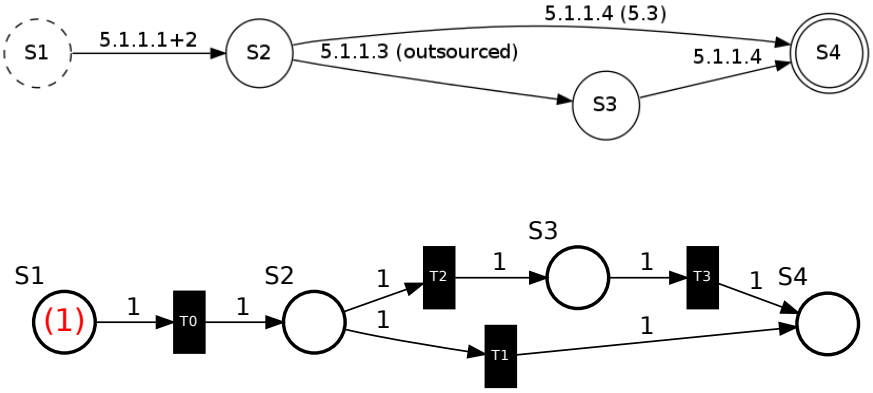


Figure 3.4 – The mapping, or translation, of a finite state automaton to a Petri net is a trivial process, as the former is a special case of the latter.

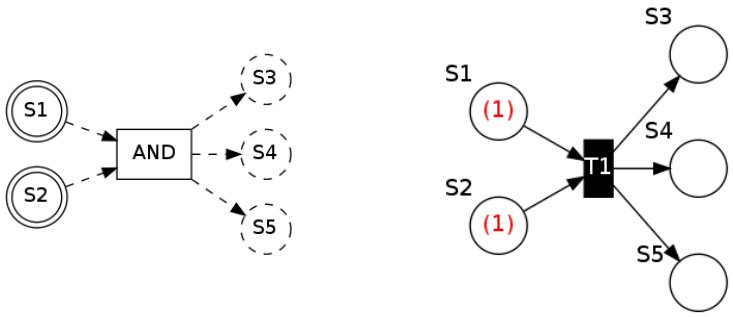
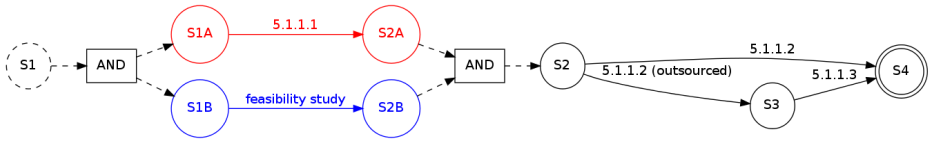
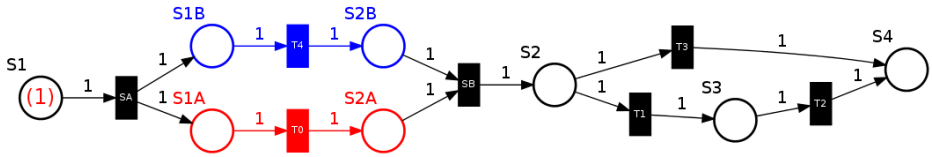


Figure 3.5 – The logical AND shown on the left is mapped as a new transition T1 with multiple incoming and outgoing arcs in the corresponding Petri net on the right. Note the tokens in the two final states on the left, symbolised by red numbers 1 in parenthesis, meaning that the transition is ready to fire, i.e. to be executed.



Part 1: A FSM with synchronization, similar to the one in Figure 3.1 on page 33.



Part 2: The resulting Petri net.

Figure 3.6 – Example of a Petri net resulting from the mapping of a FSM with synchronisation. The mapping is straightforward.
NB: the inversion of the different paths and colours is due to the automatic layout engine of the prototype used to represent and execute the processes.

places (states) that has to contain a token in order to enable the transition, and where the outgoing arcs, possibly only one or even none, enables the places (states) that have to be activated. The example in Figure 3.5 shows an AND in a finite state machine and the corresponding Petri net notation where the AND becomes transition T1 with corresponding inbound and outbound transitions.

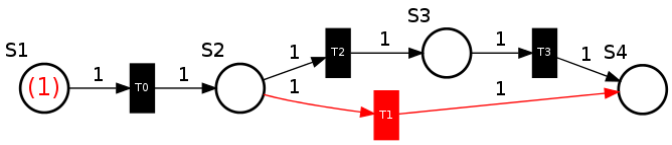
Formally, the AND between a set of final states S^f of a SSM Σ and a set of initial states S^i of another SSM Σ' is a transition t of the Petri net with the particular flow relation where all corresponding places mapped from $s^f \in S^f$ and places $s^i \in S^i$ are linked by t, producing the flow relation $F = (S^f \times \{t\}) \cup (\{t\} \times S^i)$. Figure 3.6 shows a concrete example of the mapping of a CSM to a Petri Net.

3.5.2 Composition

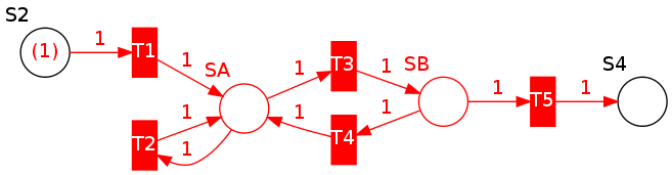
In the mapping of finite state automata to Petri nets, a state-transition (an arc) of the automaton becomes a transition in the corresponding Petri net. Since scalability deals only with the (semantic) equivalence of two SSM at two different level of detail, a SSM where a transition has been replaced by a CSM is nothing more than a special case of FSM and as such can be mapped directly to another Petri net. This is shown in Figure 3.7, using the same example as in Figure 3.3 (page 37).

Note that it cannot be said that this second Petri net is equivalent to the first one, as the equivalence of Petri nets is something completely different that has to do with the way the nets behave (isomorphism) and implies among other things that two equivalent systems always have the same number of cases, events and steps, which is clearly not the case here.

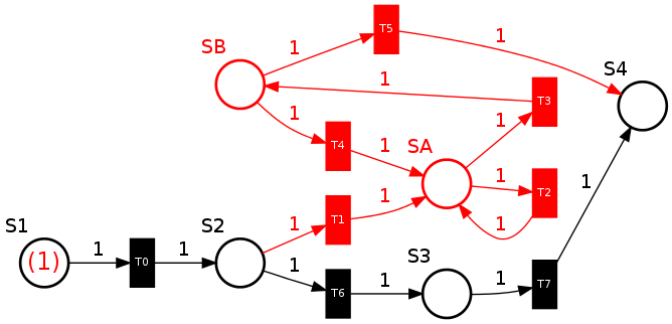
In other words, the Petri net is simply the (automatic) mapping of the whole state machine, resulting in the combination and synchronisation of as many components as the design requires.



Part 1: The Petri Net corresponding to the FSM in Part 1 of Figure 3.3, page 37.



Part 2: The Petri net correponding to the FSM in Part 2 of Figure 3.3, page 37.



Part 3: The resulting Petri Net corresponding to the FSM in Part 3 of Figure 3.3, page 37.

Figure 3.7 – The Petri Nets corresponding to Figure 3.3 (page 37). The parts in red are the parts that are replaced, and illustrate the resulting Petri Nets after the replacement of a transition in a FSM by a more detailed CSM. Note that the composition is not done at the level of Petri Nets, but at the level of FSM.
NB: the inversion of the different paths and colours is due to the automatic layout engine of the prototype used to represent and execute the processes.

Chapter 4

Application to SLCM

This chapter depicts examples of the application of the model to SDLC processes, described in the IEC/IEEE 12207 document [22][1]. To easily edit the work-flows using the graphical representation presented in Chapter 3, and to perform the mapping and execution of Petri nets, a prototype was developed, which is described in Appendix A. The figures in this chapter were generated with this prototype.

4.1 Introduction

The IEC/IEEE 12207 document [22] is structured with a top-down approach. First, primary life cycle processes are identified, described, and assigned a number (5), which constitutes the root of the nomenclature and the classification. The numerical classification follows the hierarchy of the processes, and is reflected in the nomenclature. For example, process "5.1.1" is the third level of classification below the root (5).

Other methodologies use the same approach, like HERMES [61], but they fail to classify sub-processes hierarchically, putting a greater burden on model interpretation.

[1] For reference, version 12207.2-1997 of the document was used.

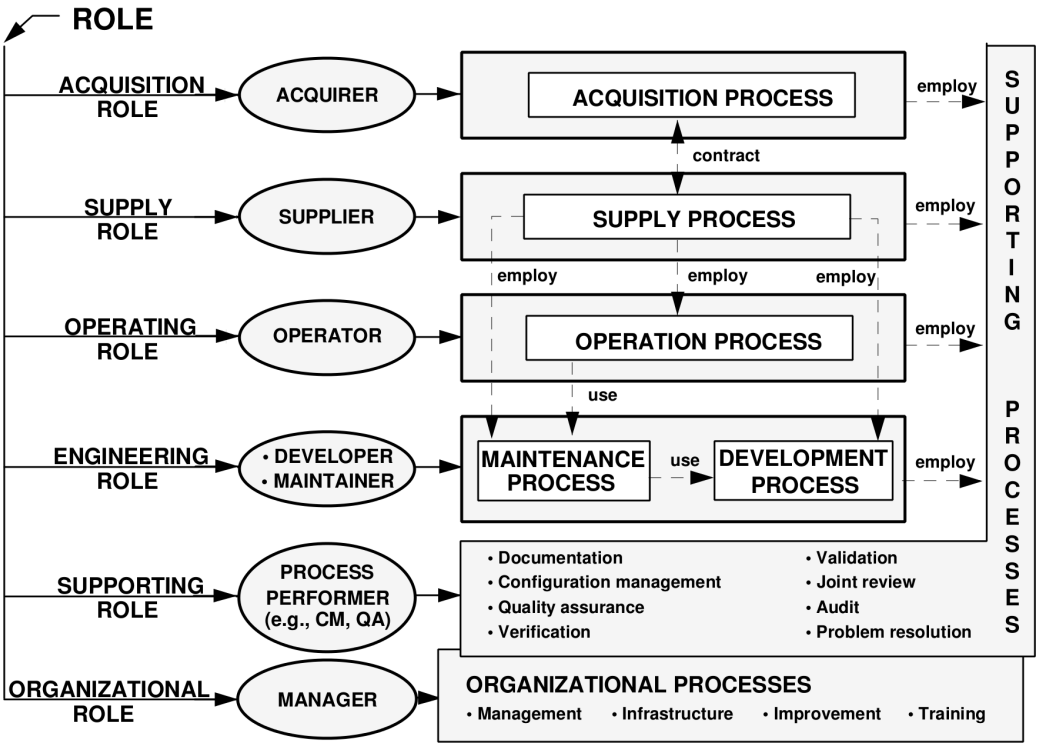


Figure 4.1 – Graphical representation of the relationships of primary life cycle processes and roles, taken from the IEC/IEEE 12207 document [22].

The primary life cycle processes, described and graphically represented in Figure 4.1, are the following:

1. Acquisition
2. Supply
3. Development
4. Operation
5. Maintenance

Following the classification, they are named “5.1 Acquisition”, “5.2 Supply”, etc.

The five primary processes are accompanied by supporting and organisational processes, numbered (6) and (7), which is also the case in many other management methods, as some things are transversal. Those supporting processes are continuous activities that run in parallel of the sequential development or other operations of the life cycle itself. They are difficult to

model, but one can always provide a custom process respecting the various guidelines of these processes, as the methodology encourages tailoring.

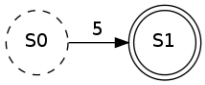
Each sub-process involves activities, performed by roles, and produces an output, for example a result (report, software milestone, acceptance). What's also interesting is the reuse of other processes, which the document refers to as "invoked processes". This property was beneficial to test our system, as it is exactly the composition mechanism derived from Systems Thinking.

The problem arises when some sub-processes, or activities, are in fact mere guidelines or subject to interpretation.

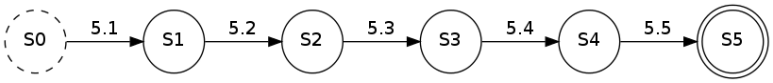
The following sections show one possible interpretation of the document in terms of our model, and its mapping to Petri nets for graphical validation (and possibly future automatic or semi-automatic validation), concluding with a discussion of this particular case.

4.2 Modelling the Processes

As Royce puts it in his paper on the waterfall model [50], a development process can be described in terms of two steps: first, analyse, and second, code. Similarly, the life cycle can be summarised at bird's-eye view as: managing the application's life cycle. This is actually what happens if we represent the SLCM process as only what the IEC/IEEE 12207 document [22] calls "Primary process" (5), which with our prototype becomes:



The root process is then subdivided into five sub-processes, so this particular automaton, viewed at a higher level of detail in our system, becomes:



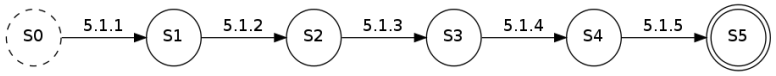
Let us take a closer look at each one of these five sub-processes. Process 5.1, "Acquisition", is described as a process involving the tasks of a role "acquirer". It is composed of five sub-processes, classified 5.1.x, where x is the number in the list below, according to the nomenclature specified above:

1. Initiation
2. Request for tender preparation
3. Contract preparation and update

4. Supplier monitoring

5. Acceptance and completion

The corresponding sub-process in our model and system is then similar to the previous one, only with other labels on the edges:



Note that for the moment, we haven't yet composed the automata, but if we were to do so, state S5 of sub-process 5.1 would become state S1 of process 5.

Now, if we look a bit further down the structure of the nomenclature, and analyse sub-process 5.1.1, it starts to get interesting. Process 5.1.1, initiation, is made up of nine activities, described in text from page 6 to page 8 of the IEC/IEEE 12207 document [22]. These contain some constraints on the way things should be done, for example 5.1.1.2 which states:

> *"The acquirer will define and analyze the system requirements. The system requirements should include business, organizational and user as well as safety, security, and other criticality requirements along with related design, testing, and compliance standards and procedures."*

Other activities are no activity at all, but a note specifying that another sub-process should be invoked somewhere, for example 5.1.1.5 states:

> *"The Development Process (5.3) should be used to perform the tasks in 5.1.1.2 and 5.1.1.4."*

This is why the resulting automaton lacks some sub-processes (numbered arcs), as they are not directly represented as activities. Figure 4.2 shows the resulting automaton. Note the similarity to the example that was used in Chapter 3, Figure 3.3, that was used as an example for exactly the composition mechanism that is of interest here. Activity 5.1.1.6-b "Develop yourself", as well as 5.1.1.6-c/d which provide the choice to develop internally, externally, or both respectively (not represented on the diagram), can be replaced by customised processes such as the one in Figure 3.3 (page 37).

In 5.1.1, we see an alternative: either develop yourself (5.1.1.6-b), or acquire a commercial off-the-shelf software (COTS) (5.1.1.6-a).

It goes on like this for the next sub-process, 5.1.2 "Request for tender preparation", which is subdivided into four activities. The interesting element at this point, is activity 5.1.2.2, which states:

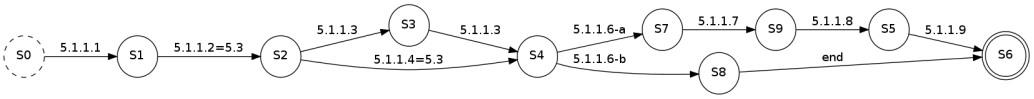
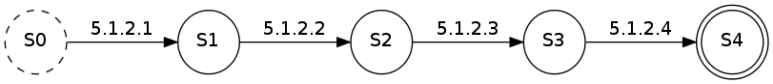


Figure 4.2 – Activity 5.1.1 with two alternatives proposed in the document: First the possibility to choose whether to sub-contract requirement analysis, and second to choose between internal or external development. Note that 5.1.1.6 actually allows six possibilities, some being combinations of the other, consequently our interpretation only provides a single alternative (two possibilities).

> *"The acquirer should determine which processes, activities, and tasks of the International Standard are appropriate for the project and should tailor them accordingly. Especially, the acquirer should specify the applicable supporting processes (clause 6) and their performing organizations, including responsibilities (if other than supplier), so that the suppliers may, in their proposals, define the approach to each of the specified supporting processes. The acquirer will define the scope of those tasks that reference the contract."*

This kind of activity leaves the manager with the possibility to tailor her own process, which is key. We will come back to this later, as it is exactly where the proposed model has an important role to play.

But first, let us continue drilling down the document and see what we encounter. Activity 5.1.2, "Request for tender preparation" is trivial to model in the way it is described:



However, the preparation of a such a document might involve internal procedures, like the successive reviewing and acceptance of hierarchy or the four-eyes principle, in which case the activity could be modelled at a yet higher level of detail, specific to the organisation. One example for a pretty standard administration would be:

1. Produce a draft of the document for reviewing.
2. Send the draft for reviewing to the assigned reviewer, and change it according to comments until there is agreement (cycle).
3. When an agreement is met, send it to the CIO for acceptance.
4. The CIO accepts the document (end of the process).

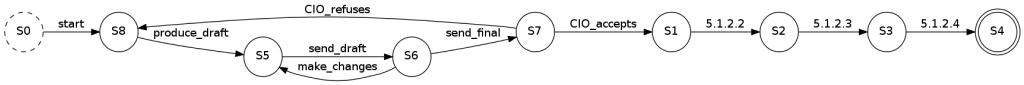


Figure 4.3 – Process 5.1.2 with a customised 5.1.2.1 sub-process. Note the "start" between S0 and S8, necessary for S0 to be an initial state for the model. Without this "start", coming back to it from S7 would break the composition mechanism.

This process is modelled in Figure 4.3, where activity 5.1.2.1 has been replaced by the specific subprocess.

Process 5.1.3 involves negotiation directly in the description. One possible interpretation of it is modelled in Figure 4.4. The most interesting part in this case is the fact that when one reads the document, in particular the last activity 5.1.3.4, one realises that the process can cycle forever. Also, activity 5.1.3.5 states:

> *"Once the contract is underway, the acquirer will control changes to the contract through negotiation with the supplier as part of a change control mechanism. Changes to the contract will be investigated for impact on project plans, costs, benefits, quality, and scheduling."*

This means that activity 5.1.3.5 could be modelled as a cyclic monitoring/controlling mechanism, in which case one has to be careful about when the activity ends. We will come back to this later when we compose all the elements and see what the model looks like for the whole process.

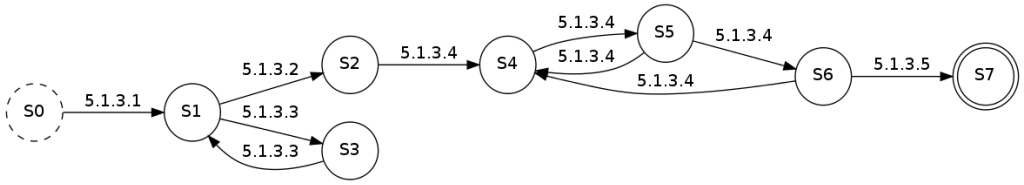
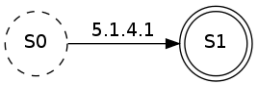


Figure 4.4 – Process 5.1.3. Note that activities 5.1.3.3 and 5.1.3.4 had to be modelled with several states and arcs, as they comprise several steps. This is subject to interpretation, but here 5.1.3.3 is a simple review process (loop), and 5.1.3.4 models a simple negotiation with a third party (initial document, then negotiation until verification, then acceptance or refusal).

Activity 5.1.4, "Supplier monitoring", consists of only two elements, one of which is a monitoring activity, that has to conform to support processes in the document, and the other is a guideline:

> *"The acquirer will cooperate with the supplier to provide all necessary information in a timely manner and resolve all pending items."*

So 5.1.4 is a single step in our case:



This activity can be refined into something cyclical as well. The same problem as for activity 5.1.3.5 arises: it has to be defined throughout the process where this activity takes place, as it is an ongoing process. Figure 4.5 describes a possible, simple interpretation for the purpose of this example.

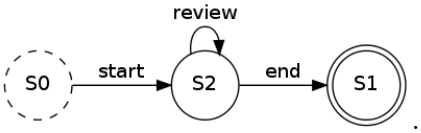
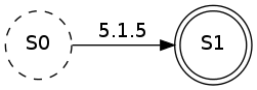


Figure 4.5 – Interpretation of process 5.1.4. It represents a simple cyclic activity. It could be defined according to the support processes, but this is irrelevant at this point of the example.

Activity 5.1.5 is "Acceptance", and defines guidelines for an acceptance mechanism, as well as responsibilities. It is similar to the quality gates in HERMES. For our purpose, it is modelled as a single activity:



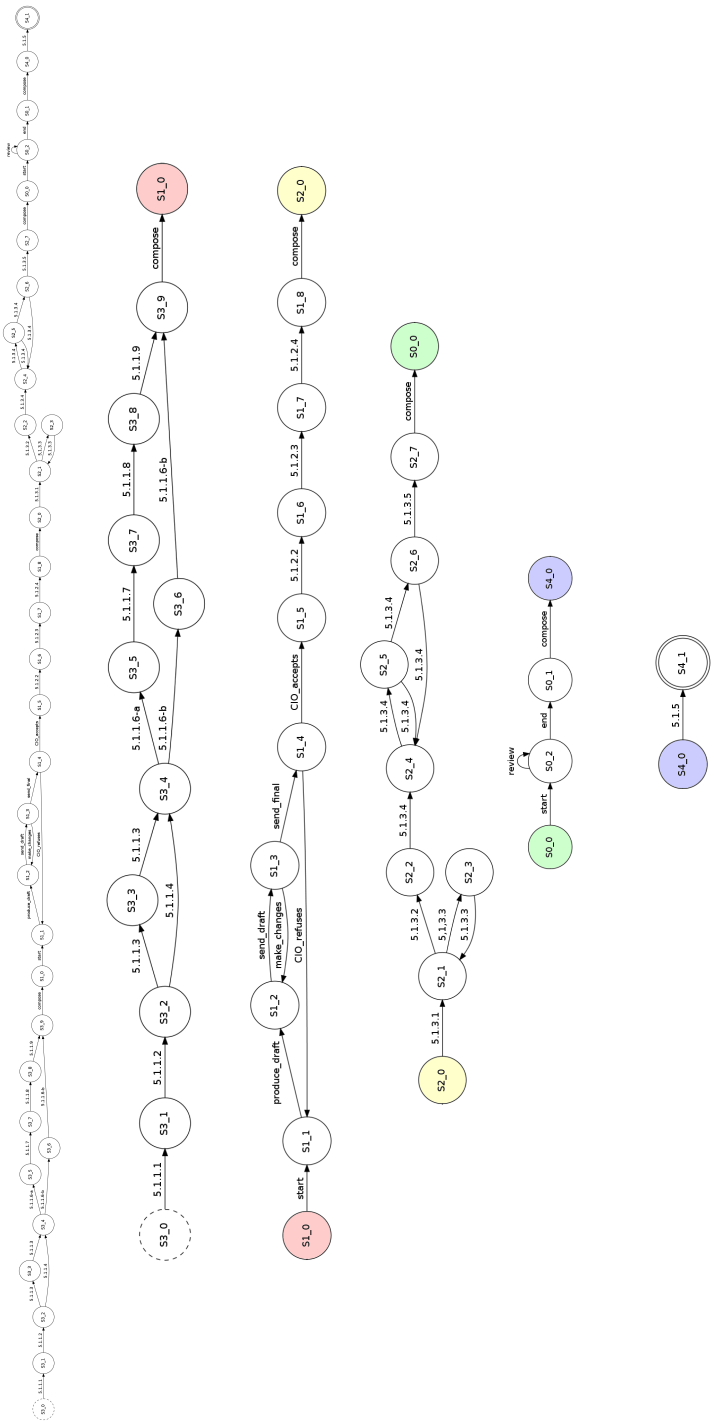
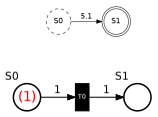
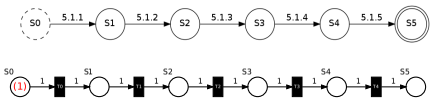


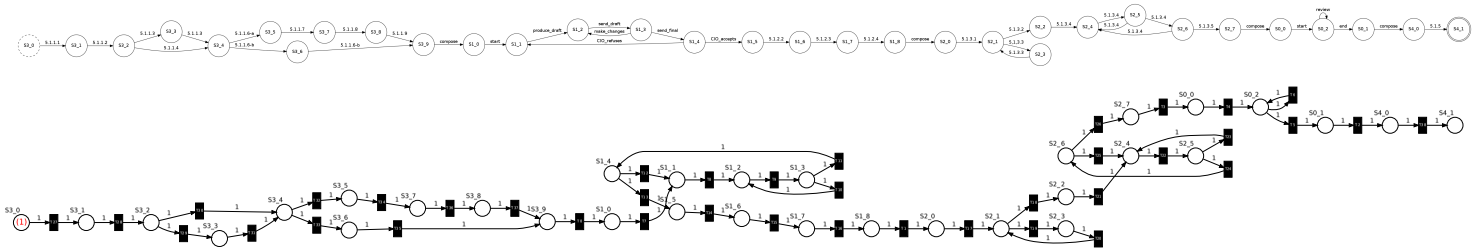
Figure 4.6 – Complete representation of 5.1 with the composition of the various activities described earlier. Note the linearity of the process. The process has been split into five parts for readability, using colours for readability. *Note: the colours were added manually.*



Activity 5.1.



Activity 5.1 at the initial level of detail.

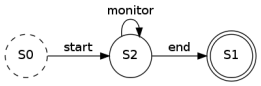


Activity 5.1 at the highest level of detail.

Figure 4.7 – Complete representation of 5.1 at different levels of detail, with corresponding Petri nets. The Petri nets contain a token in the first position (place). The complete Petri net contains cycles, which can be detected by running it or with cycle detection algorithms, but it retains the linear nature of the modelling algorithm, i.e. it will eventually end with the token on the far right.

4.3 Adding Concurrent Activities

The model of Activity 5.1 as it is depicted is linear with only alternatives and no concurrent processes. Activity 5.1.3.5, as mentioned above, is actually a continuous monitoring activity. One way to model this is:



It has to be synchronised with the whole process at some point. An option could be to start it at the end of Activity 5.1.3, and ensure that it ends only after the whole process but before the acceptance stage, resulting in the model in Figure 4.8.

By taking a closer look at the whole Petri net, and at the different paths that can be taken, without fully exploring the state space, one can start by exploring the paths interactively by running the Petri net using the simulator described in Appendix A and selectively firing transitions, which could result in Figures 4.9 and 4.10.

The same can be done with any other activity, and other points of synchronisation can be added in the middle of processes, to model quality gates, for example (HERMES [61, 60]). One just has to be careful to model them appropriately at the semantic level, but at least purely at the representational level, mistakes can be detected very easily by looking at the execution of the Petri net, either interactively, or systematically, by exploring the net's state space. Some analysis can also be done algorithmically, by detecting the Petri net's network properties, like cycles or dead-ends, under certain conditions. Some possible leads are presented as future work in the conclusion (Chapter 6).

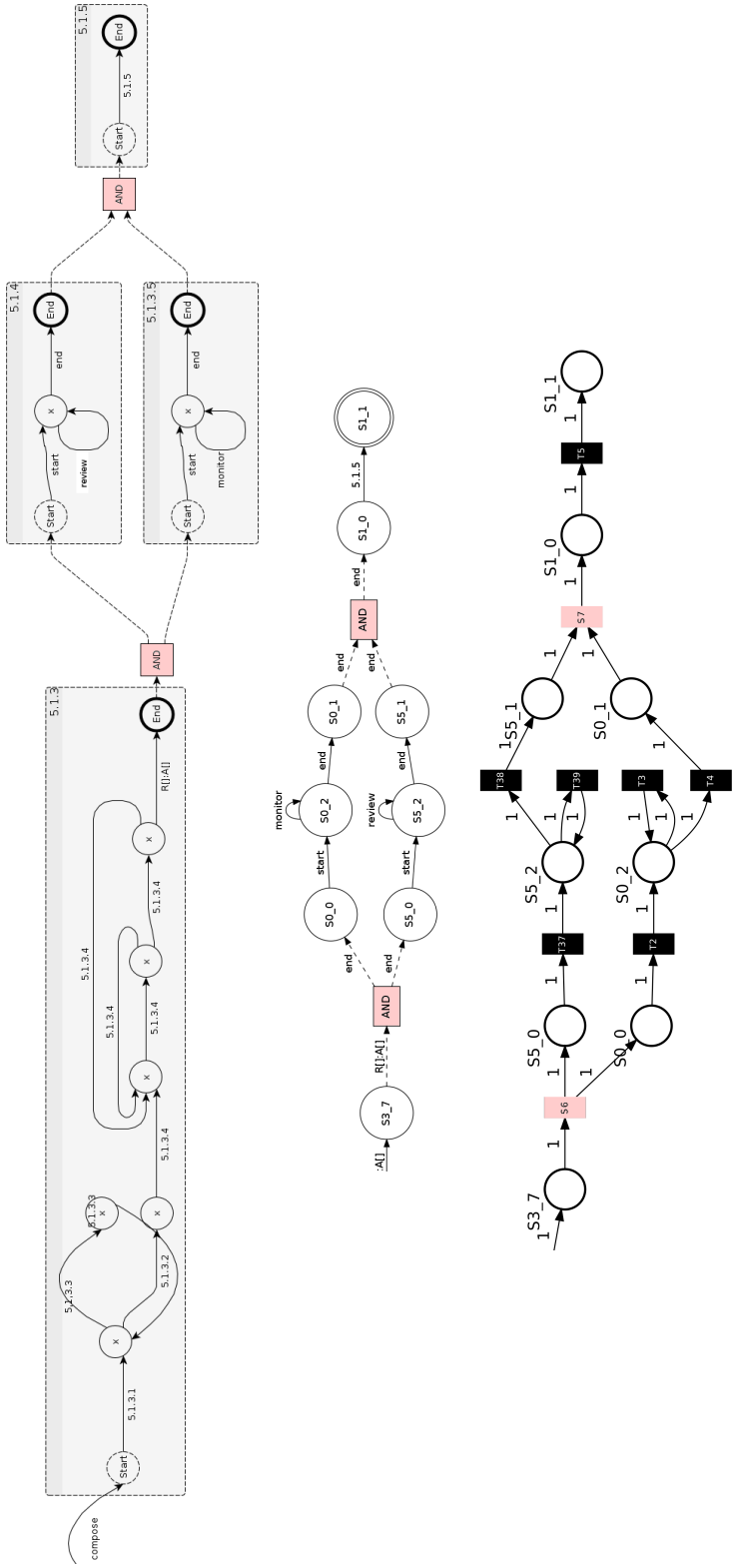


Figure 4.8 – Activity 5.1.3.5 is now a customised process, synchronised with the rest. The top is a screen-shot of the relevant portion of the model in the editor, the middle part is the representation in the engine, and the bottom part is the corresponding portion of the Petri net.

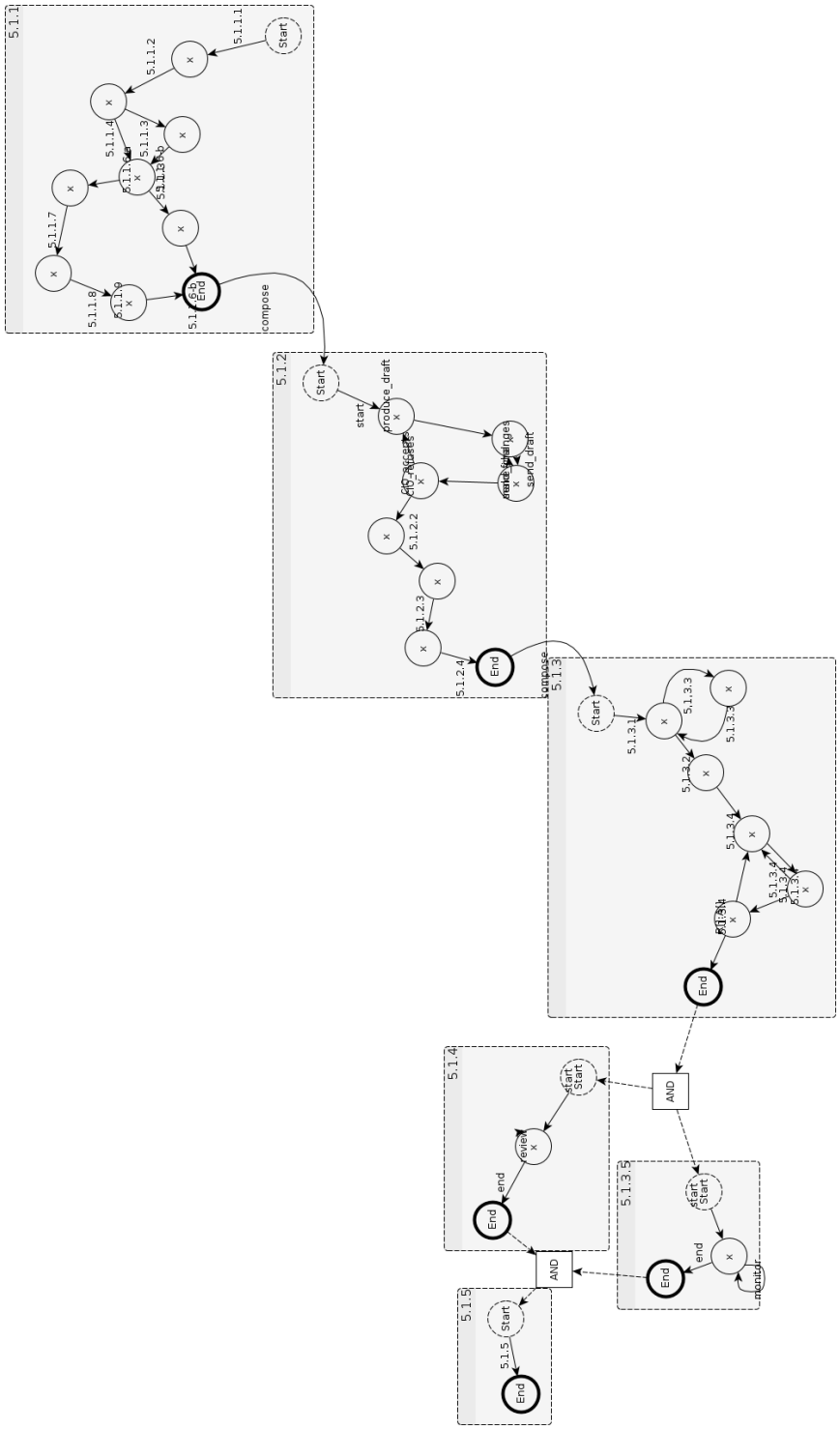


Figure 4.9 – Activity 5.1, complete with 5.1.3.5 customized and synchronised. Note that the start is on the upper right corner of the figure, as laid out automatically by the "organic" layout engine of the editor. With a hierarchical layout, the figure would be ordered left to right, but too elongated to fit on the page.

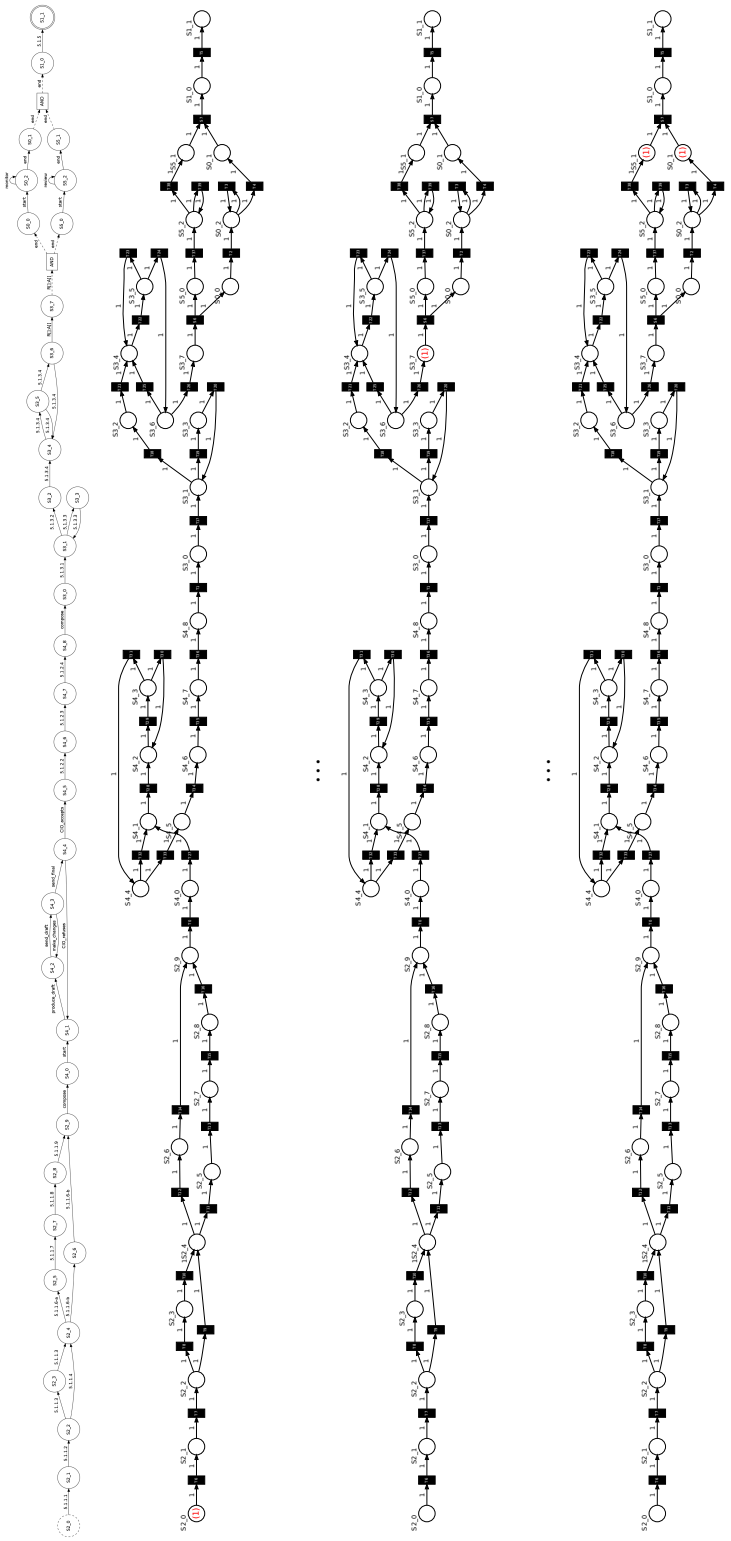


Figure 4.10 – Activity 5.1, complete with 5.1.3.5 customised and synchronised. The last two parts illustrate the synchronisation mechanism in a Petri net (notice the token.)

4.4 Discussion

The preceding sections illustrate the complete modelling of Activity 5.1 in the IEC/IEEE 12207 document [22]. The remaining activities 5.2 to 5.5 can be interpreted and modelled in exactly the same way, and extended with custom processes when necessary. The example illustrates the use of the proposed model and the main advantage it provides over other representation techniques: composition and synchronisation. The entire model has been produced based on the above mentioned document, but apart from possible differences in interpretation, including Activities 5.2 to 5.5 in the example does not add any value.

Furthermore, the SDLC process is very linear by nature. There are cyclic activities here and there, but on the whole, the composition resembles Figure 4.10, only on a bigger scale. After all, the process was designed that way: a project is by definition time constrained, and therefore its phases must be as well. The very essence of software development, as Royce has already stated (Chapter 2, Figure 2.1, page 8) and the fractal nature of the way the process is designed engenders this property.

Better to look at in more detail, though, is the actual customised processes that can be plugged into the model, thanks to the composition mechanism described in Chapter 3. Tailored methodologies are common practice, and suffer from the fact that they cannot be precisely validated in theory, but only in practice, and moreover with much difficulty.

The next section shows some of the main advantages of the model when it comes to detecting some of the most common problems of modelling small, toy-like processes that illustrate the essential features of the custom, tailored activities that could arise in practice in this context.

4.4.1 Detection of Graphical Errors

The system and the mapping allow the detection of purely graphical mistakes in the representation. For example, the process described in Figure 4.11 on the top might seem perfectly valid. However, the corresponding Petri net in the middle shows that there is a problem: a node overlaps another in the representation. The Petri net gets blocked in a particular state and doesn't allow the token to reach the final state, which is illustrated in the bottom part of the figure. This error was actually detected by the engine, since it is only graphical, but at the editor's level, i.e. at the interactive level, it goes unnoticed, which shows the necessity to complete the pure representation with at least some automatic translation or validation.

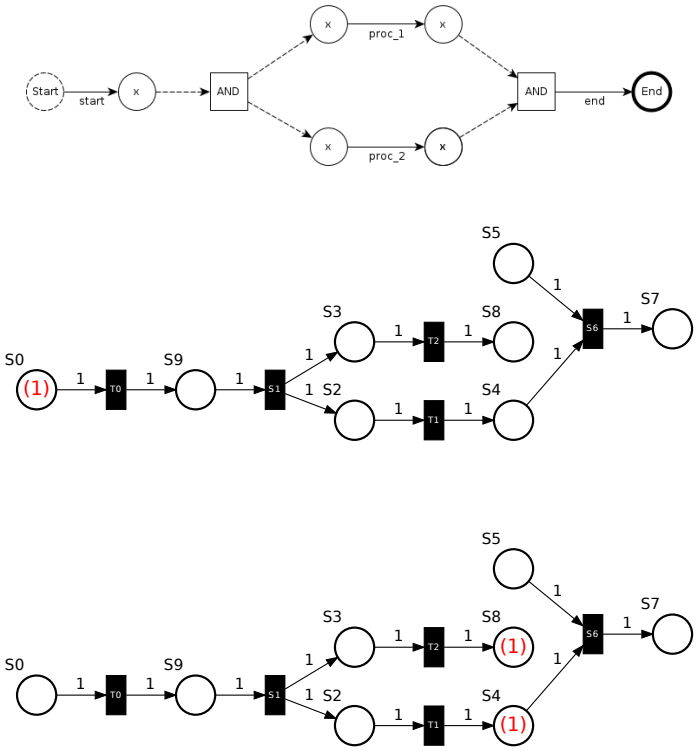


Figure 4.11 – The process modelled at the top may seem perfectly valid to the human eye. However, when mapping it to the corresponding Petri net, we realise that it contains a problem. In this particular case, node S5 and S8 overlap with each other in the editor.

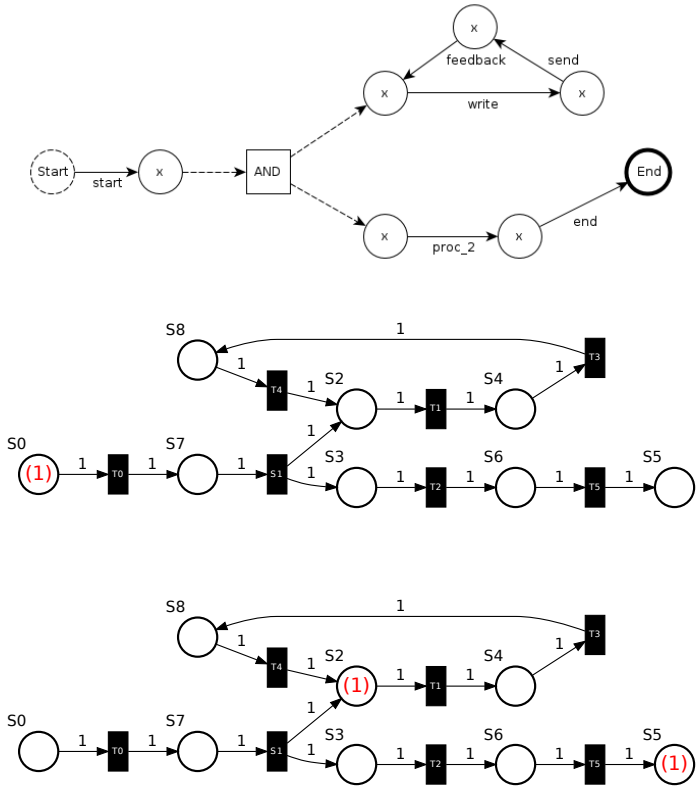


Figure 4.12 – The cycle at the top can be spotted easily in this purely illustrative example. But with more complicated sub-process compositions, it would be harder to detect graphically. The two states of the Petri net in the middle and at the bottom show the cycle, which can also be detected by cycle detection algorithms.

4.4.2 Detection of Cycles

Cycles, in the context of processes, as seen from the human point of view, are of several kinds. Some, like the monitoring added and synchronised in activity 5.1 (section 4.3, Figures 4.8, 4.9, and 4.10), do not pose any real problem in themselves. At the semantic level, they can take a very long time to end, at worst, which might even be desirable depending on the nature of the process (emphasis on quality control, or financial controlling for example). But there are cyclic activities that lead to endless loops and block the process, like the simple example in Figure 4.12.

The last example is trivial, and the cycle is detected at design time (unless some arrow overlaps another), but some more complicated cycles could be harder to detect. For instance, there could be whole sub-processes in-

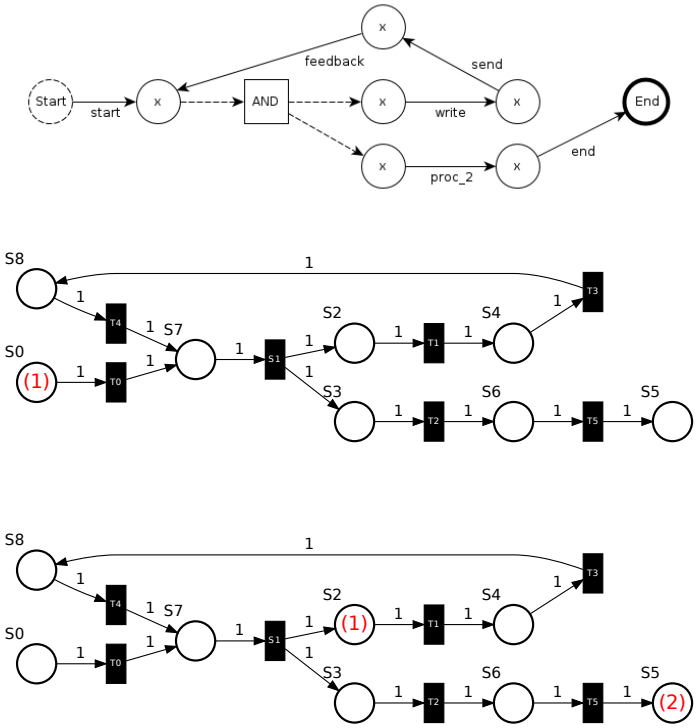


Figure 4.13 – Compared with Figure 4.12, the cycle is now the product of a mistake in synchronisation. The corresponding Petri net is interesting, as it generates several tokens that end up in the final place at the state described at the bottom of the figure.

volved, that are perfectly valid in the sense of our restrictions when viewed independently, but combined in such a way that the process actually never ends. Figure 4.13 is an example of such a mistake in the synchronisation mechanism.

As mentioned in the introduction of this section, this research limits itself to exploratory detection at the graphical level, be it in the model or the resulting Petri net, and at the Petri net's execution level. There are algorithms for cycle detection in (directed) graphs in general, like depth-first traversal and more elaborate solutions like Tarjan's strongly connected component algorithm [57], applicable to Petri nets in particular, which would provide interesting future work for automatic validation, mentioned in the conclusion (Chapter 6).

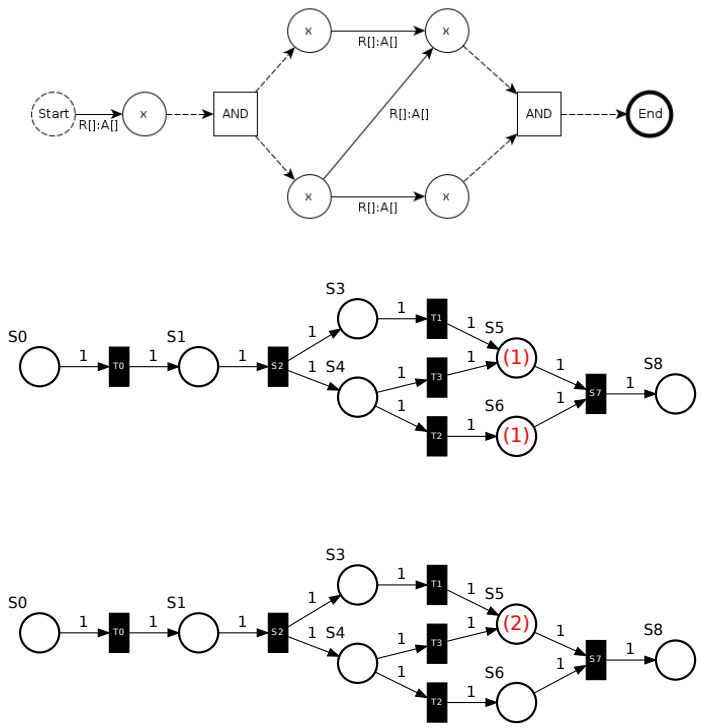


Figure 4.14 – The top part shows a model of a process with a badly constrained alternative (no re-synchronisation, which actually breaks the rule of composition). The problem here is that one scenario finishes (the middle one, when the two tokens are consumed by the last transition), the other not (the bottom one, which produces two tokens in S5 and therefore cannot continue).

4.4.3 Detection of Badly Constrained Alternatives

One could try to express an alternative at some point between continuing a sub-process or branching into another sub-process, like in Figure 4.14. This breaks the necessary conditions for a component state machine at this stage (section 3.4), but it is bound to happen in modelling real processes. Depending on the choice between T2 and T3 (alternative), the network ends, or remains blocked with two tokens in S5. This is detected at execution time, and it would be helpful to investigate possible algorithmic detection of such cases in a more general way, possibly using liveliness and reachability properties in the graph, although practically speaking the reachability problem has the disadvantage of being EXPSPACE-hard [30], even though it is clearly decidable in theory [27].

4.4.4 Advantages over Other Representations

There are many other variants of the problems listed in the last three sections that arise when modelling processes in this way: unreachable states (disconnected graphs), intractable alternatives similar to the one in Figure 4.14, multiple imbrications of cycles. In this respect, the representation itself is not better than other possibilities like BPMN or other work-flow diagrams. But it has two main advantages:

1. The model puts constraints on the way automata can be composed together and synchronised. This is a limitation for some processes, but in the particular context of SDLC processes it is a main advantage. Errors like badly constrained alternatives or even purely graphical problems can be detected easily, either during the mapping itself or at run-time.

2. Other representations can be executed, but do not put any constraints (finite state automata, Petri nets as such, or even UML sequence diagrams), or they constrain the possibilities, but cannot be executed or analysed formally (traditional work-flow representations, BPMN). So this approach provides a good trade-off between power of expression, and tractability. Moreover, the semantics of the transitions, although not explicit in the representation, can easily be reconstructed, so things like multiple tokens or lost tokens can become acceptable properties if they actually mean something.

More investigation would be needed to actually provide a strict analysis of all the possible properties, constraints and applicable algorithms under such conditions in the context of this model. Some possible leads are given in the conclusion (Chapter 6).

Chapter 5

Application to BPMN

The two contributions presented so far in this thesis, together with the example in Chapter 4, already reduce the gap between representation and formalism in the context of SDLC (figure 1.1 (page 4)). The model can be seen as a proposal that stands somewhere in the middle of the diagram, providing a good trade-off between simplicity on the one hand, and analysability or expressivity in terms of capabilities for system property validation, on the other hand. This is illustrated in Figure 5.1.

The gap *is* indeed reduced, but not entirely bridged. The de-facto standard for business process modelling is BPMN, and the community is not likely to adopt a new notation or representation, even a very simple one, especially if it was developed purely in an academic setting, which is the case for this model. More work is needed to effectively bridge the gap in more practical settings, by analysing a possible mapping between BPMN and our model.

There is some amount of research that already goes in the direction of bridging the gap in the specific context of BPMN. For instance, in the context of web services validation a language called BPEL, or WS-BPEL (for web-services business process execution language) exists, that could serve the purpose of analysing interesting properties in the same way we did in the preceding chapter by running the Petri nets interactively. Some research suggests that a type of mapping is possible [68].

Moreover, with BPMN 2.0, the execution of conform diagrams[1] could lead to the same result without the burden of having to map to another model. This would have to be investigated more thoroughly of course, but one argument in favour of our approach is that the mere mapping provides interesting insight into the diagrams, by reducing their complexity in terms of graphical elements, and by removing the semantics, which enable purely

[1] Conformance is defined in the BPMN specification document [58]

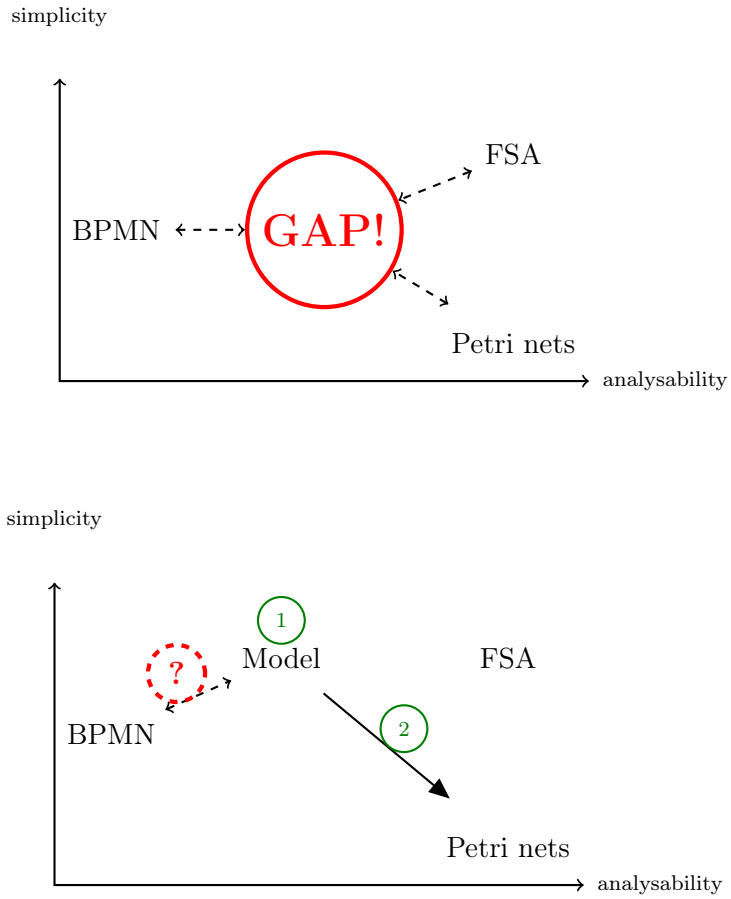


Figure 5.1 – A diagram showing the two first contributions in relation to the two desirable characteristic properties of the two worlds (managerial and technical): simplicity, and analysability, and how they contribute to reducing the gap presented in the introduction, page 4. The gap is indeed reduced, but still exists, as the model doesn't represent a de-facto standard for business process modelling, and is not likely to become one in this form.

formal verifications, using graph-theory algorithms, Petri-net specific methods, or even mathematical proofs [41]. The same argument would hold for BPEL, provided the mapping can be entirely constrained, which is not entirely clear [31].

Also, the management world has been quite active with Pi-calculus in the past two decades, and as Smith and Fingar put it: "Workflow is just a Pi process" [54]. This lead is still active [42, 43], but very controversial, as many things simply cannot be expressed correctly, which was demonstrated in several publications by van der Aalst: "Why Workflow is NOT just a Pi process" [66], and "Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype"" [65]. The argument may well be settled soon by the mapping of WS-BPEL to Petri net as proposed by Ouyang et al. [33].

This chapter presents such a mapping to Petri nets, which at this point constitute the indisputable best-established formalism in the context of workflow modellling and validation, as the literature suggests [62, 63, 64, 67, 41].

A few examples of application to real BPMN processes are presented at the end of the chapter, that show promising results, as well as future research leads.

5.1 Mapping BPMN to the Model

There have been attempts to map BPMN directly to Petri nets, with some success [62, 63, 64, 67, 41]. The problem encountered is the expressivity of BPMN, that allows for the complex conditions in the gateways, which are difficult to model. Moreover, BPMN is open to extensions, which renders the mapping impossible altogether. Finally, the semantics behind BPMN diagrams is lost in the process.

This section explores the mapping of BPMN to our model, which suffers from the same limitations as mentioned in the previous paragraph since the model is in itself a special case of Petri nets with a simpler notation. The proposal consists in a partial mapping, but nevertheless provides some interesting insights for graphical validation of the processes at the level of our model, and in the mapping to the Petri net itself.

For example, it is trivial to map simple structures made of start, intermediate, and end events, like the one in Figure 5.2. Different types of events, like "error", "cancel" or "signal" among others, which convey particular semantics, are treated as regular events, which means semantics is lost in the process.

An activity is essentially the same, but one has to be careful to introduce

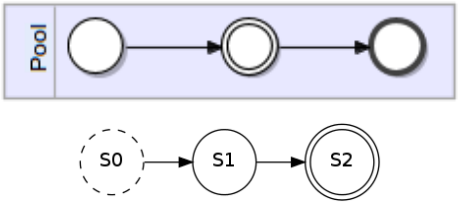


Figure 5.2 – The three BPMN elements of the "event" type mapped to the model. Note the difference in notation for intermediate and end event and states between BPMN and finite state automata representation.

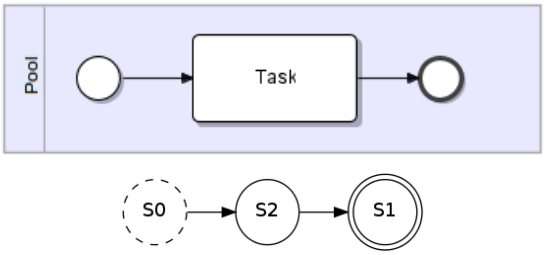


Figure 5.3 – The BPMN elements of the "task" type mapped to the model. A task becomes an intermediate state. This is necessary to allow sub processes to be represented with correct start and end states in our model.

an intermediate state, to ensure the possible mapping of the sub process activity. Figure 5.3 illustrates the mapping of a task, the others are essentially the same.

The specific problem of error-handling with an intermediate error event attached to the boundary of an activity has been left out of the mapping. It could nevertheless be mapped as in Figure 5.4, where the catching of an error during the execution of a task produces an additional outbound arc, i.e. an alternative.

Gateways are more complicated. Exclusive gateways are mapped as alternatives, as in Figure 5.5. Parallel gateways are essentially the synchronisation mechanism, so they are mapped to the AND node of our model, as in Figure 5.6.

Message-based gateways of the same types pose no problem, provided one forgets about the difference in semantics between message and normal flows. Our system simply maps message flows as normal transitions and synchronisations (see the examples in Section 5.2).

Messages in general are synchronisation mechanisms between pools, so message flows are treated the same as sequence flow, which means we forget about the semantics of message-passing. Figure 5.7 illustrates such a

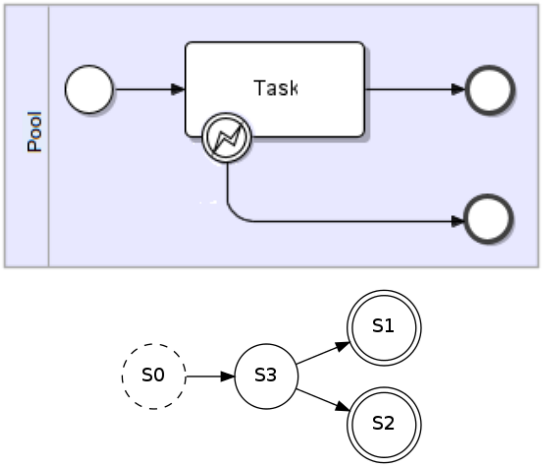


Figure 5.4 – The BPMN elements of the "intermediate error event" type attached to the boundary of an activity, mapped to the model. This particular case of error handling has not been included in the prototype, but can be mapped as an alternative.

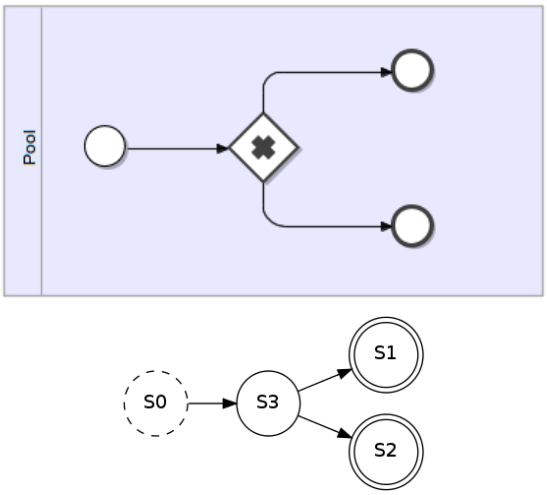


Figure 5.5 – The BPMN elements of the "exclusive gateway" type mapped to the model. It is trivial: an alternative. The additional state allows for the differentiation of the composition and synchronisation end and start states.

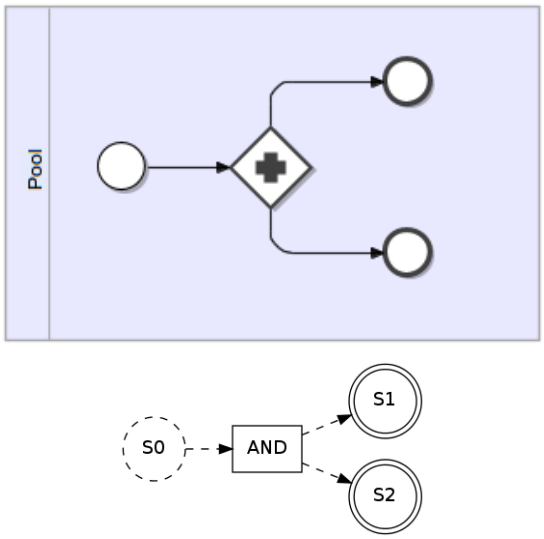


Figure 5.6 – The BPMN elements of the "parallel gateway" type mapped to the model.

synchronisation between two parallel activities using messages and parallel gateways.

But inclusive and complex gateways pose a problem. The model cannot represent them. One would have to map directly to a Petri net with the appropriate number of tokens to provide possible paths, depending on the semantics of the condition in the gateway, which is also problematic, as one would have to impose restrictions. In any case, the mapping in our model takes only one possible path into account, which is not a problem for simply exploring the network interactively, but becomes a serious limitation if one is to apply algorithms for system property validation on the resulting Petri net.

Sub-processes and transactions have been left out of the mapping. The former is concerned with the level of detail at which the process is seen, so the mapping would have to map the entire process, with all sub-processes expanded, and the latter is roll-back mechanism that is impossible to translate in the model, as it describes things happening at the semantical level.

Artefacts are not mapped as anything at this point, but data artefacts can be mapped either to final states (places) or to tokens if one were to map directly to a Petri net. The problem is again the semantics of the BPMN, which is completely lost during the mapping, as the examples in the following section illustrate.

There also exist extension of BPMN diagrams, for example choreography diagrams in BPMN 2.0, which allow the synchronisation between processes

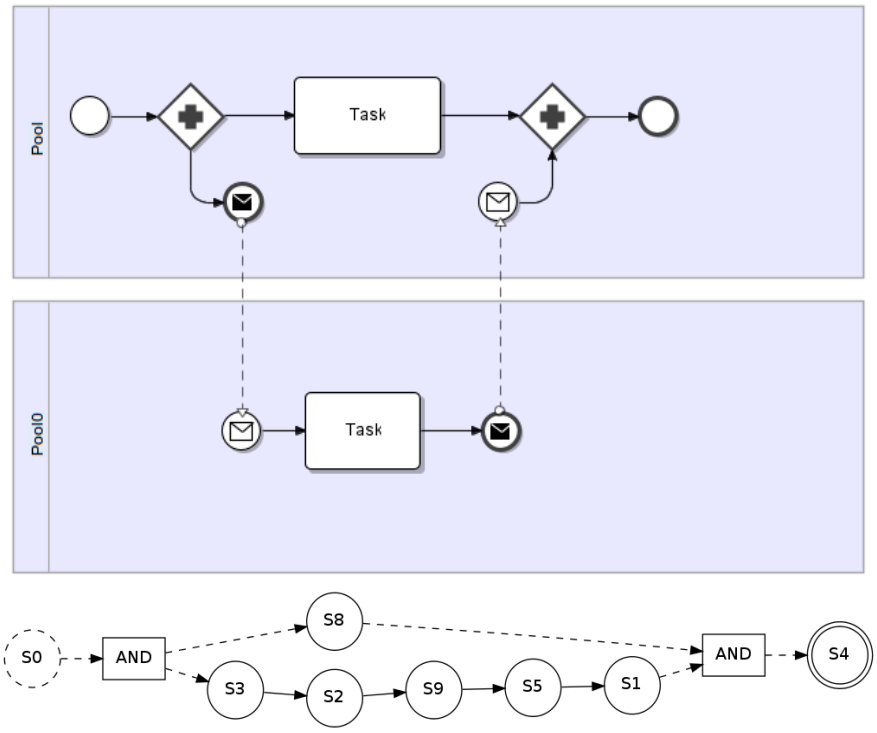


Figure 5.7 – The BPMN elements of the “message” (flow) type mapped to the model. It is considered as a normal sequence flow, but we then lose the semantics, which poses the problem of interpretation later if one attempts to go back from the Petri net to the original model to figure out where the error is.

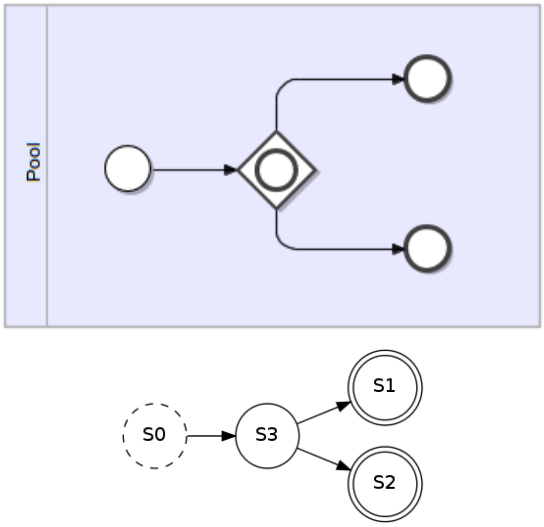


Figure 5.8 – The BPMN elements of the "inclusive gateway" type mapped to the model. It is mapped as an alternative, which is a limitation, as all paths can be evaluated.

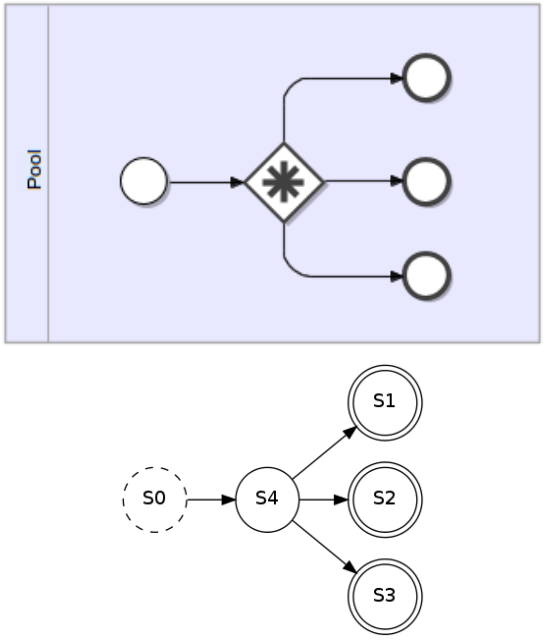


Figure 5.9 – The BPMN elements of the "complex gateway" type mapped to the model. It is mapped as an alternative, but the choices could be more complex, consequently another limitation.

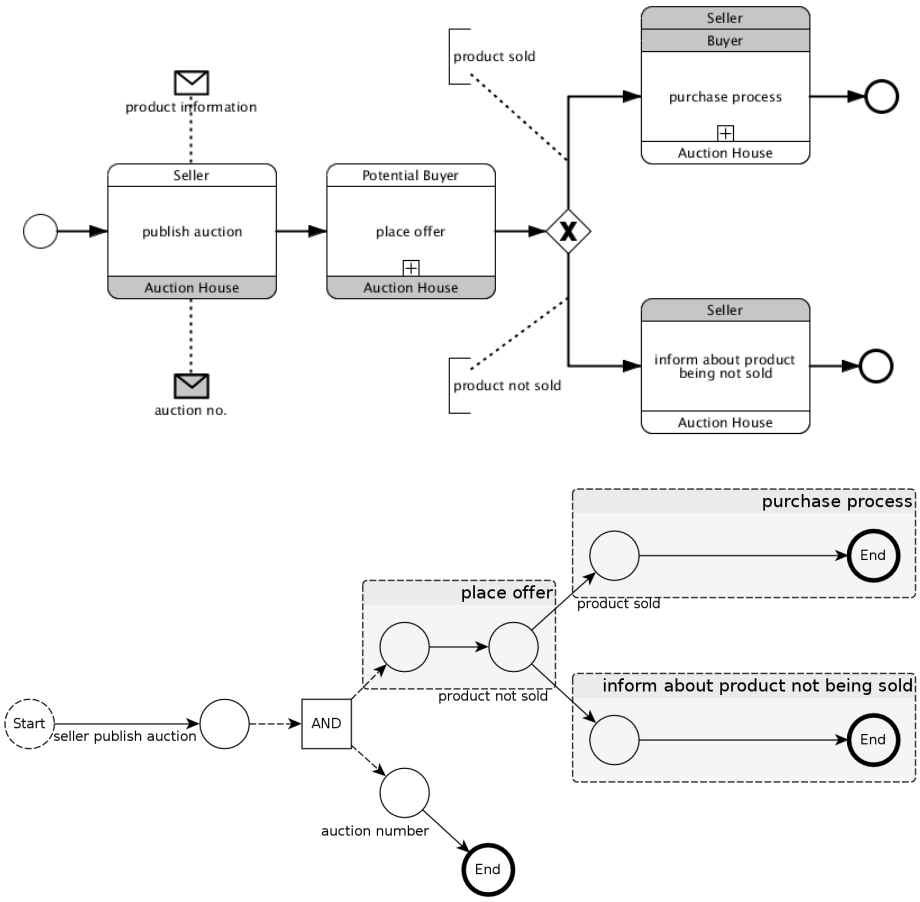


Figure 5.10 – An example of BPMN choreography diagram from the BPMN 2.0 Tutorial at BPMNcommunity [8], and its possible mapping to the model. The lack of semantics could be a problem for more complex diagrams, and this would require more investigation.

accross pools. An example of choreography taken from the BPMN 2.0 Tutorial at BPMNcommunity [8] is presented in Figure 5.10. Note that this possible mapping has not been realised in this research, and although it seems to pose no problem provided the messages can be transformed into regular transitions, the added value of the resulting semanticless diagram remains to be investigated.

5.2 Examples

The four examples presented in this section are real cases of university regulatory documents, modelled as BPMN. They encompass processes related to courses:

1. Student course registration (figures 5.11 and 5.12).
2. Faculty creation of a new educational programme (figures 5.13 and 5.14).
3. Implementation of a new educational programme (figures 5.16 and 5.17).
4. Update of an existing educational programme (figures 5.18, 5.19, 5.20 and 5.21).

Each example is described, modelled and then discussed in terms of advantages and disadvantages. This is not an exhaustive, scientific analysis of all possibilities, but a practical application that showed some valuable results.

5.2.1 Course registration

The first example illustrates the mapping from a BPMN diagram to the model, and then to a Petri net. It also shows the disappearance of semantics in the process. It is a real example of university course registration.

The BPMN diagram in Figure 5.11 is a very simple process. It involves two distinct entities: the Faculty and IT service (SITEL). These two entities are modelled as pools, as they communicate only through the passing of messages. In the Faculty, two main actors take part in the process: the student and the secretariat. These two actors are represented as swim lanes inside the Faculty pool, as they interact directly.

The process starts with the student wanting to register, and it ends when the student has received their computer account, and the student is registered in the IS-Academia database. The end of the process is actually an event that triggers another process: registration for the exam, which is mandatory in the Faculty. This shows the importance of the systems thinking approach to problem solving that was already part of our model and exhibited in the SDLC process (Chapter 4).

This process is very simple, and entirely linear, but it exhibits some key properties. The corresponding representation using our model, as well as the mapping to the Petri net, is shown in Figure 5.12 on page 75.

The AND between S17 on the one hand, and S4 and S1 on the other hand, imply the production of two tokens (in the figure, the top line of the model is mapped at the bottom of the Petri net, due to the automatic layout engine). So there are two possible outcomes: either one, or two tokens in the final place. The latter could be seen as the production of two actual distinct results, synchronised between the two pools with message passing: the registration confirmation and the IT account. In the former case, only the confirmation is created, as the IT account already exists since the student is already registered at the university.

The fact that semantics are lost during the mapping allows for purely structural or formal validation, but doesn't necessarily mean that it cannot be reconstructed a posteriori, like in the interpretation above. It would be necessary to conduct a systematic study of such cases and their actual significance if one were to validate not only the model, but its semantics, which would certainly be valuable (Chapter 6).

In short. This simple example shows that the mapping itself is no problem under these circumstances, and provides a valid executable Petri net with a single reachable place, that is a finite state space. The only problem is with the interpretation of the outcome when more than one token arrives at the final place, but it can potentially be reconstructed afterwards.

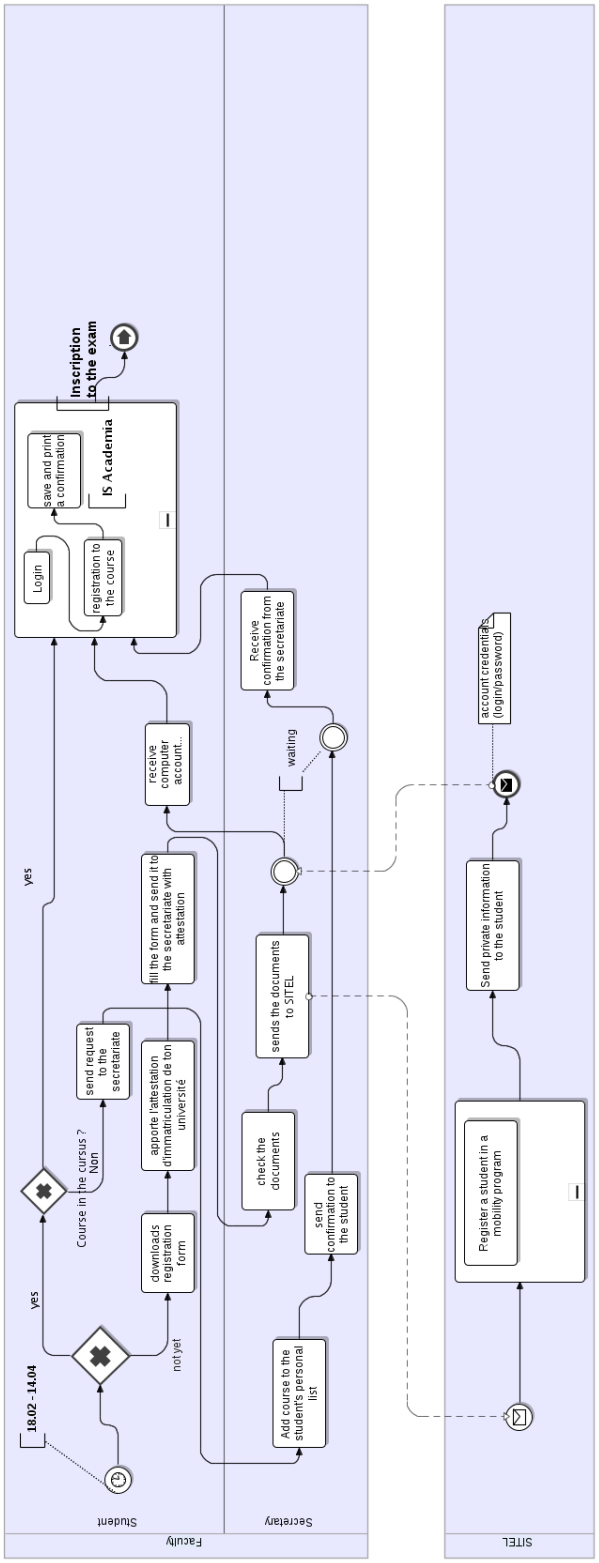


Figure 5.11 – Student course registrations.

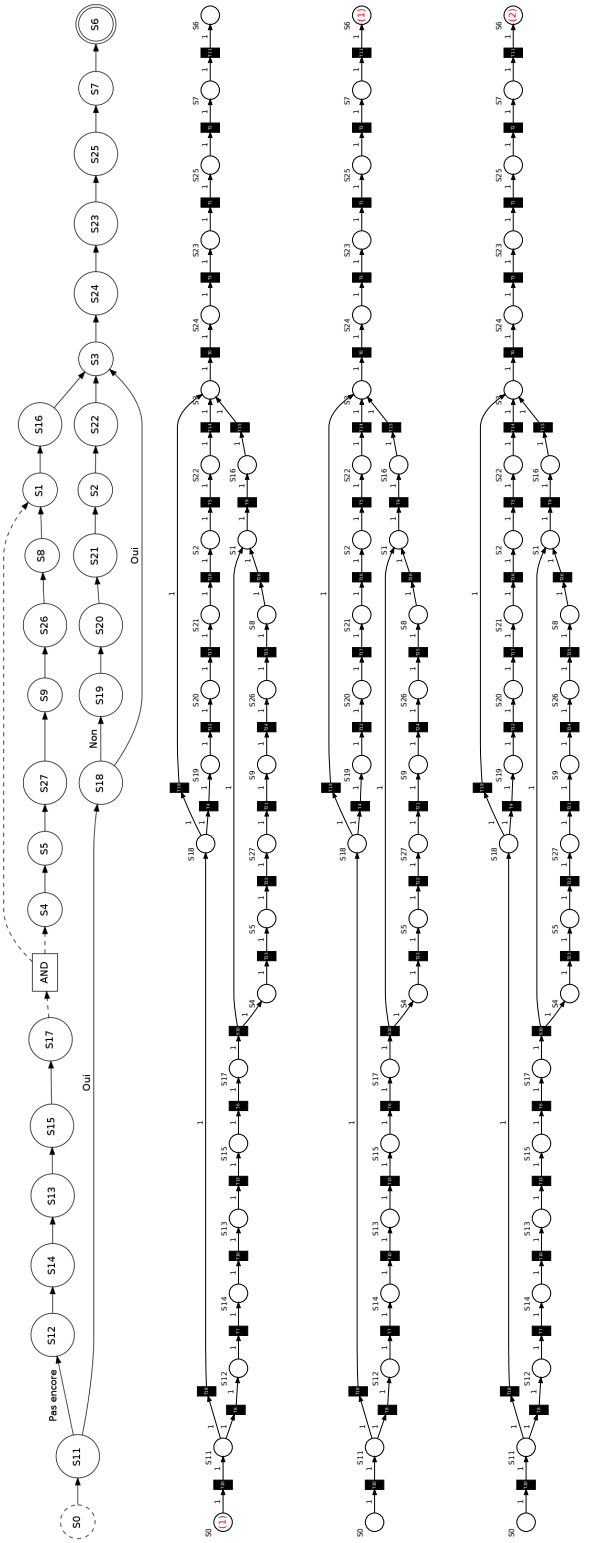


Figure 5.12 – The model shows a completely linear outcome, the process is trivial to map, and the Petri net has only one final place, which can be reached via one of two paths. Note the AND between S17 on the one hand, and S4 and S1 on the other hand, which causes two tokens to end up in the final place in the last situation.

5.2.2 Creation of a new educational programme

The second example is a bit more complex, and involves more than one final (reachable) place, which can be seen graphically or detected algorithmically. It also illustrates the composition of several concurrent activities.

The process is the creation of a new educational programme by the Faculty. It involves quite a few more entities than the previous example: the Faculty, of course, but also a commission, or expert group, and the Rector's office. As in the previous example, these fairly independent entities are modelled as pools. In the Faculty itself, three main roles are involved, these are modelled as swim lanes: the institute in charge of the programme, the Faculty Council and the Dean's Office.

The process is described as BPMN in Figure 5.13, together with the relevant mapping as a Petri net. The process exhibits no problem, although it is a lot less linear than the SDLC processes in Chapter 4 or the previous example.

It is important to take a look at the internal representation of the model to see how the concurrent processes are composed and synchronised in the model (figure 5.14). Unlike the previous example, all sub-processes are really dependent on each other, so they are synchronised back, which means that there is only one token at the end: either the programme is created, or not.

If it is created, then the process can trigger another process: programme implementation, which is the next example on page 80. Again, this is essential for breaking down the initial problem into sub-problems that are synchronised at specific decision points, like in the SDLC in Chapter 4.

In short. The preceding example illustrates the kind of graphs that result when mapping a process with more entities and roles (pools and swim lanes). It shows that the Petri net can be executed, and can be analysed in terms of system properties if need be. The three possible outcomes do not convey semantics, but this could be reconstructed a posteriori, like in the first example.

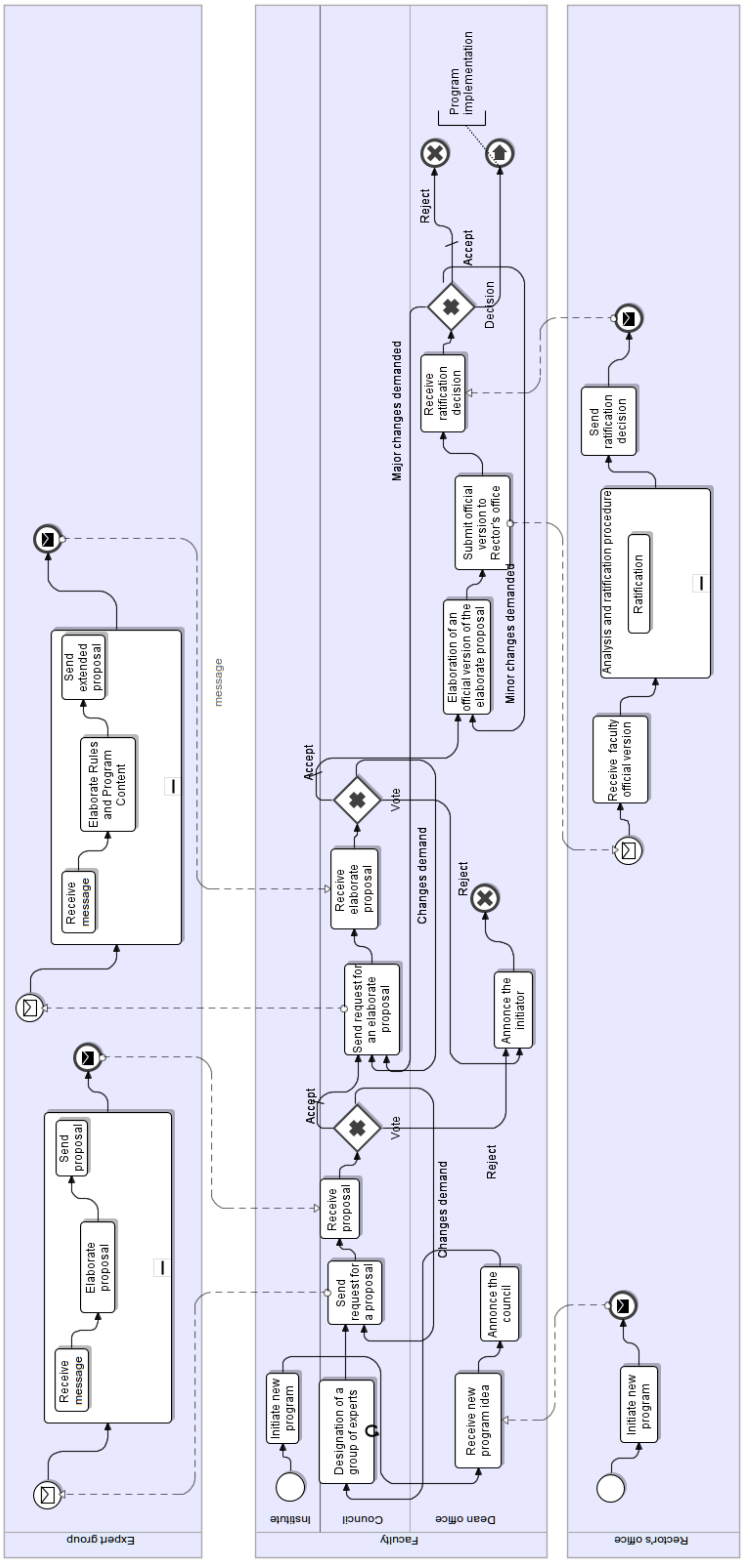


Figure 5.13 – Creation of a new educational programme by the Faculty as aBPMN diagram. Note the two possible outcomes: either accepted, or rejected, and the link with a subsequent process in the former case (programme implementation).

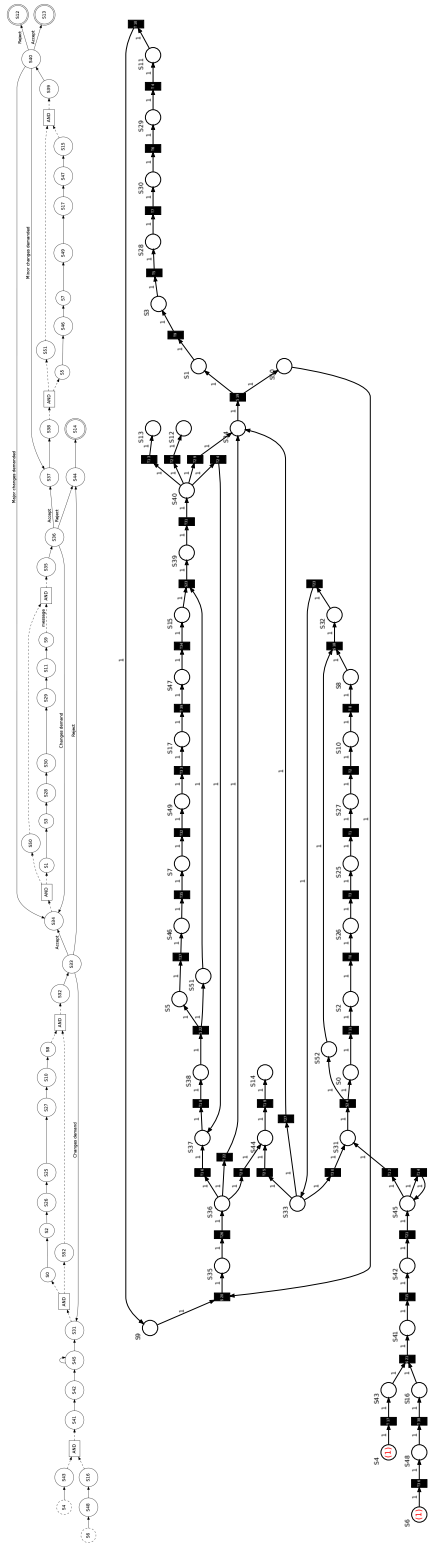


Figure 5.14 – The internal representation of the BPMN in Figure 5.13 is interesting as it shows how sub-activities are composed and synchronised. The corresponding Petri net is shown with an initial state, and its three possible final states can be viewed in Figure 5.15.

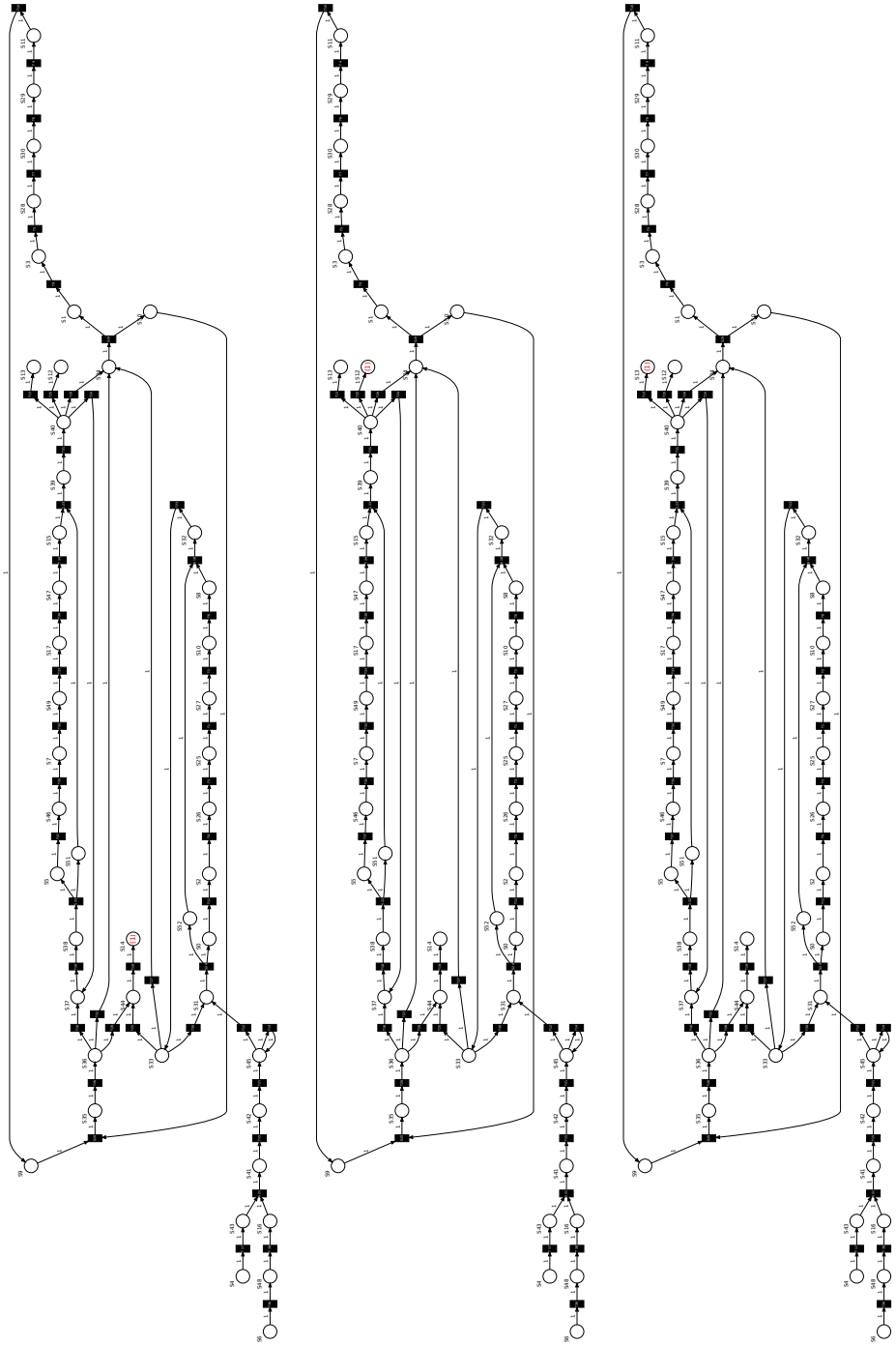


Figure 5.15 – The Petri net's (in Figure 5.13) three possible final states.

5.2.3 Implementation of a new educational programme

The third example is informative, as it illustrates what happens to the Petri net when message passing is mapped as a transition, although two final states have to be reached.

The process consists of the implementation of the educational programme created in the previous example, and involves the decision of a new entity: the University Council. Consequently the outcome is again an accept/reject alternative. The BPMN diagram is presented in Figure 5.16.

Figure 5.17 shows the corresponding model and Petri net. The interesting element here is the production of two tokens, but in distinct final places, when the course is implemented. As in the first example, the semantics can be reconstructed from the BPMN diagram itself: when the programme has been accepted by the University Council, both the Rector's Office and the Dean's Office are notified, which produces two tokens. The final states are reached and the process ends.

The Petri net loses its semantics, but not the desirable property that the two final places contain a token in the final marking.

In short. The preceding example is an informative case, as it illustrates what happens when message passing is involved, and lost in the mapping to the model and to the corresponding Petri net. In fact, only the semantics are lost: formally, the two sub-processes in the separate pools are still executed, with the two tokens in the separate final places, which is consistent with what can be observed at the model level.

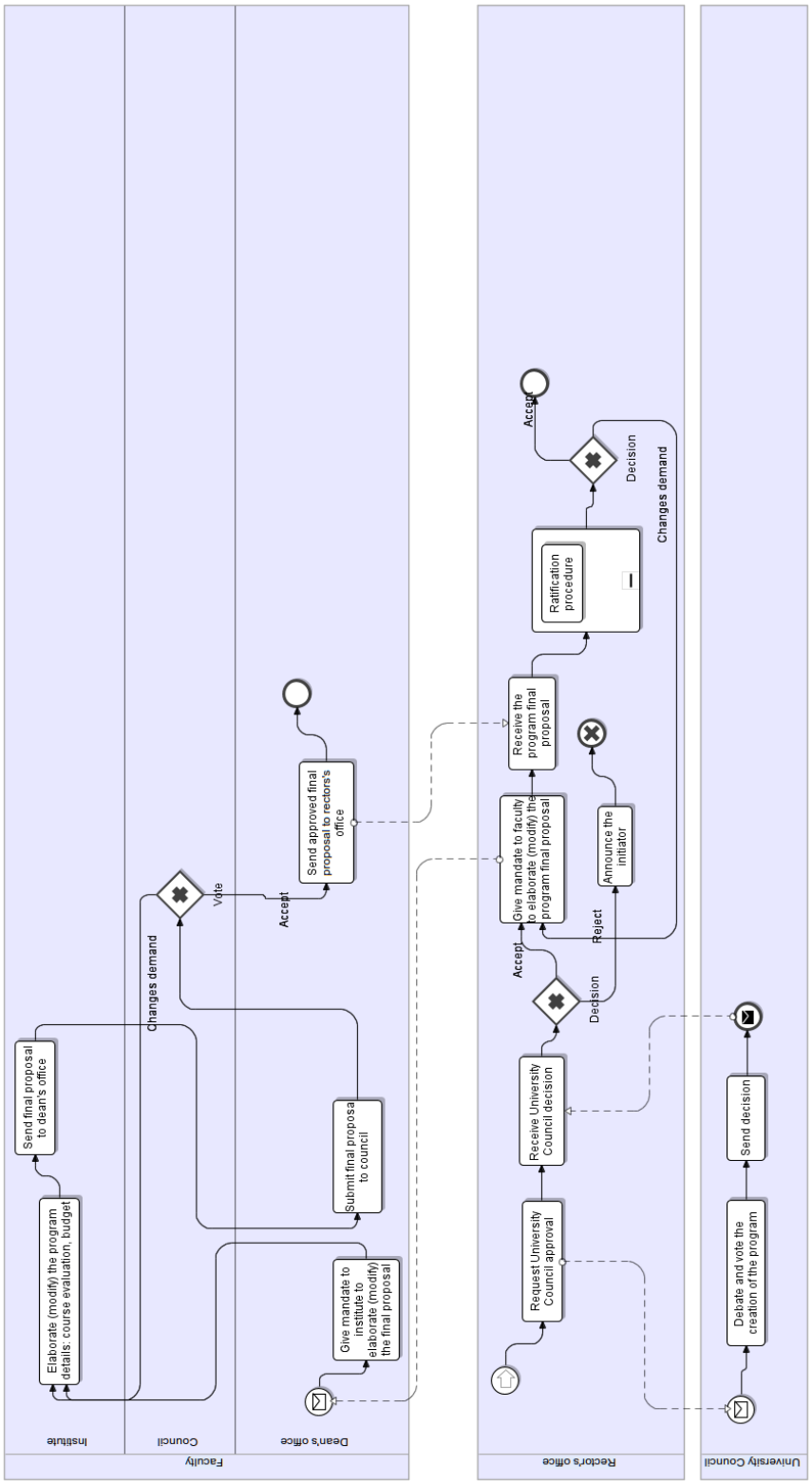


Figure 5.16 – Implementation of a new educational programme by the Faculty.

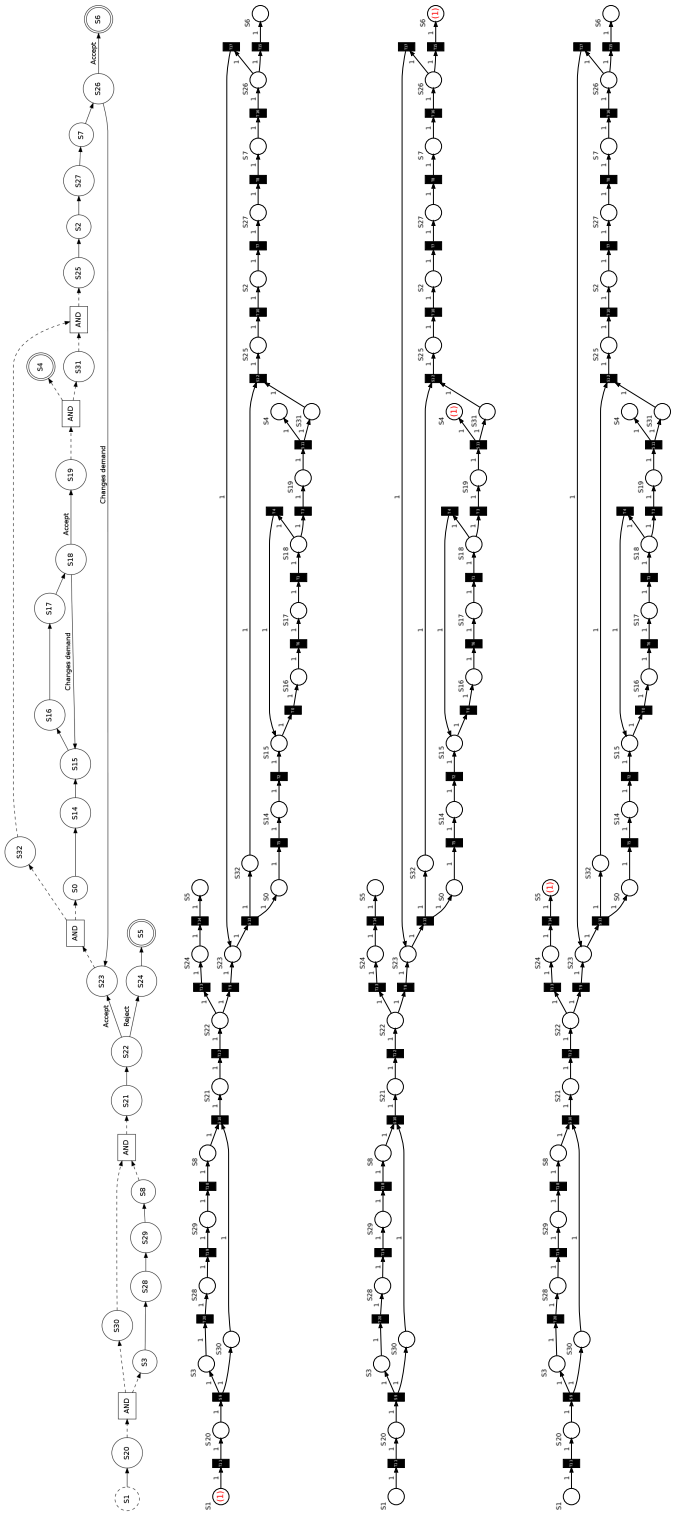


Figure 5.17 – The internal representation of the BPMN in Figure 5.16 and the corresponding Petri net with its two final states: accept or reject. Note the two tokens in distinct final places in case of acceptance (second to last).

5.2.4 Update of an existing educational programme

The last example is an important illustration of the power of mapping in detecting mistakes, that would otherwise be lost at the graphical level. It is the mapping of the Faculty's update of an existing educational programme.

The process was modelled as in Figure 5.18 and validated by several people. It involves quite a few more entities and roles, but is still fairly straightforward, and considered valid.

Now if we look at the corresponding model and Petri nets in Figure 5.19 (page 86), we immediately see that something is wrong with this model. There is a lonely start state (respectively place in the Petri net) without any outgoing transition, and the token in it never moves.

If we look back at the BPMN diagram, we notice that there are five possible events that start a request for update: either the student, the professor, the head of the programme, the quality control team, or the Dean's Office can file such a request. But in our case, one of them is ignored: the professor's one. Why is that?

If we look closely at the BPMN, the problem is clear. The edge between the professor's request and the recommendation was modelled, not as an arrow, but as an undirected link. This is shown in Figure 5.20 (page 87), at the bottom of the figure.

It is worthy to note that this mistake was never actually detected by the people involved in the modelling or validation! The correction is trivial and is shown at the bottom of Figure 5.20. The corresponding models and Petri nets, in Figure 5.21 (page 88), are more accurate.

The interpretation of this process is much more difficult, as several tokens travel along the Petri net. Indeed, the mapping assumes that all starting states involve one token in an initial place in the Petri net. That would mean that potentially, all parties could request a change, which is of course possible, but unlikely. Synchronisation with semantics is not possible at this stage, so we would have to either adapt the mapping to include such semantics, or find a better interpretation. These are all future leads mentioned in the future work section of the conclusion (Chapter 6).

In short. The last example shows that mapping provides insights that would otherwise go unnoticed. Furthermore, when the problem is actually fixed, either at this level or by executing the BPMN diagram itself, it shows the limitations of the approach for reinterpreting the semantics a posteriori.

This last example is a good illustration of the differences between the concerns of the two worlds which representations we try to bring together here. The managerial world is concerned with the semantics of the processes,

so the diagrams have to include this in some form or other, in this case by having a proliferation of elements, both graphically and formally. On the other hand, the purely formal validation that the Petri nets allows are not concerned with semantics, only with structure.

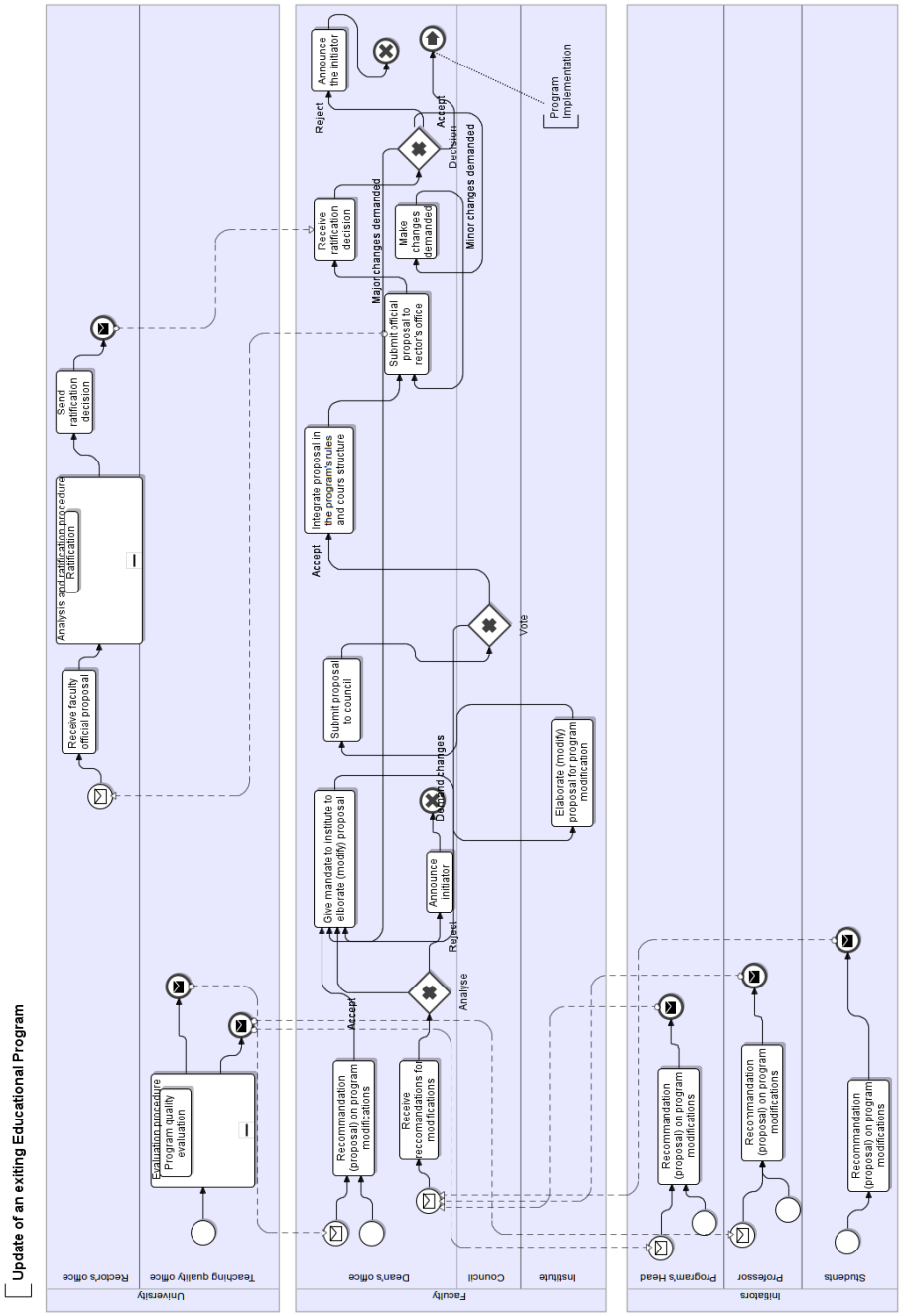


Figure 5.18 – The Faculty's update of an existing educational programme.

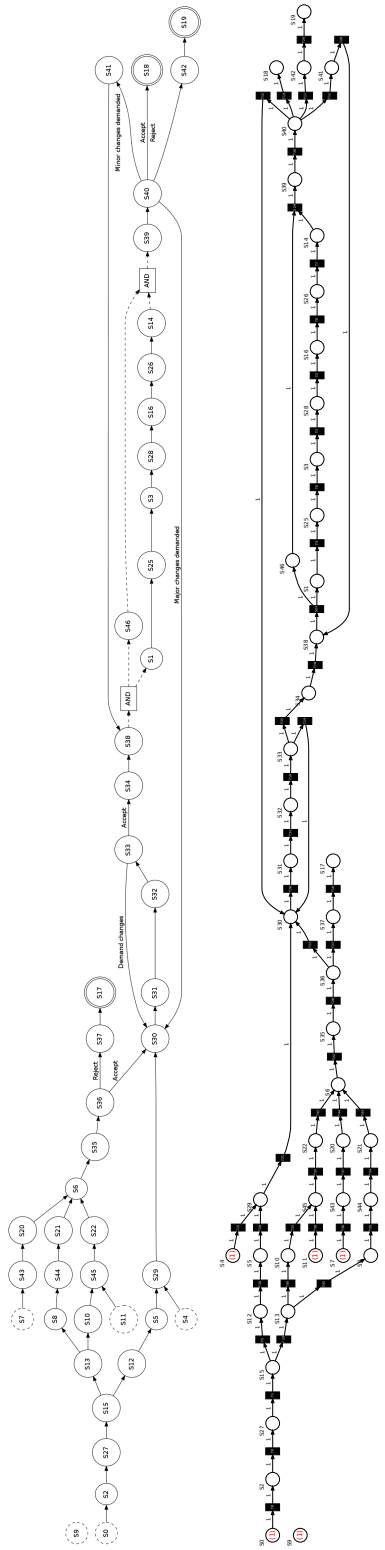


Figure 5.19 – The Faculty's update of an existing educational programme as it was proposed, mapped to the model and the corresponding Petri net. Note the lonely state S9.

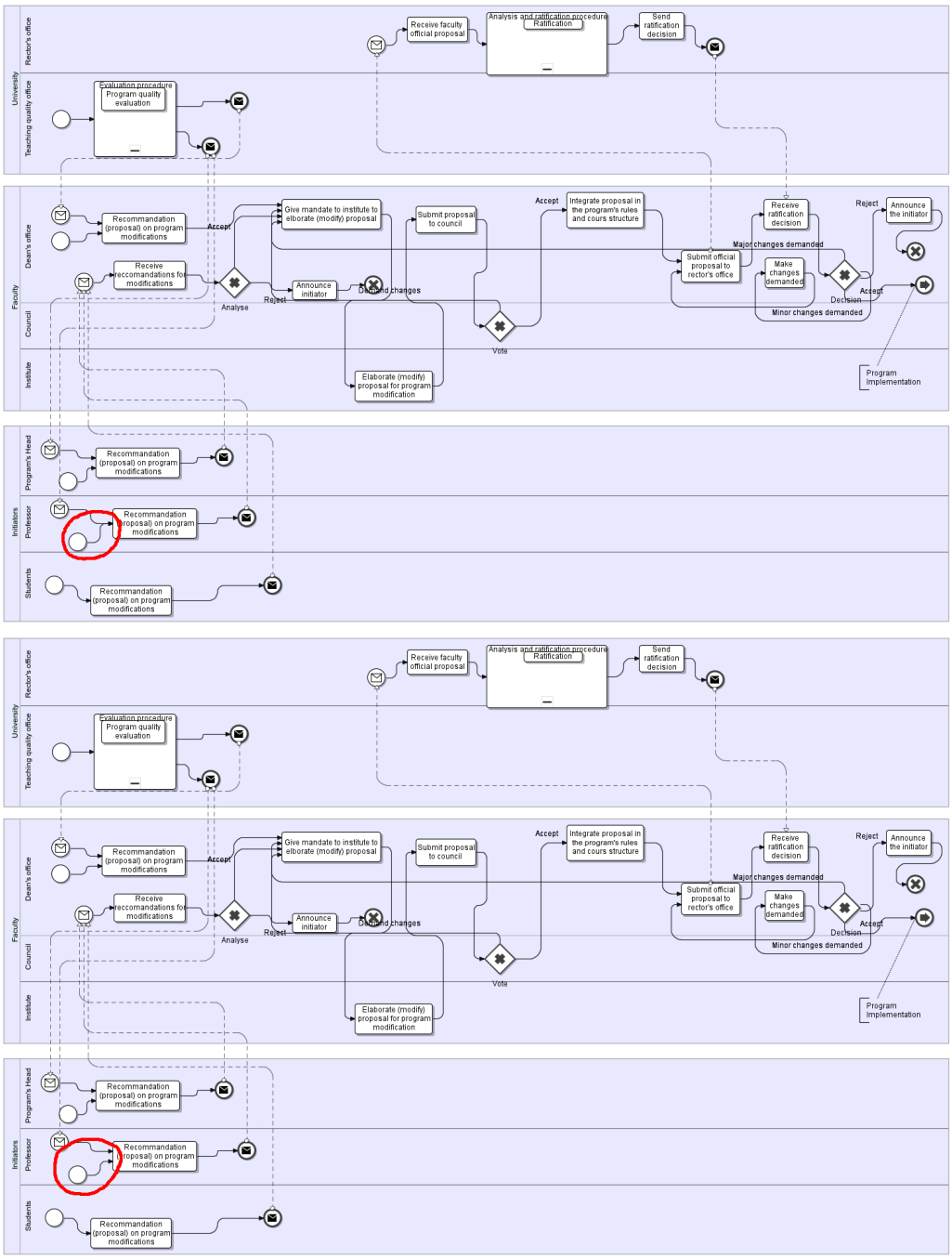


Figure 5.20 – The mistake, at the top, which was never identified by the process reviewers. The correct version, at the bottom.

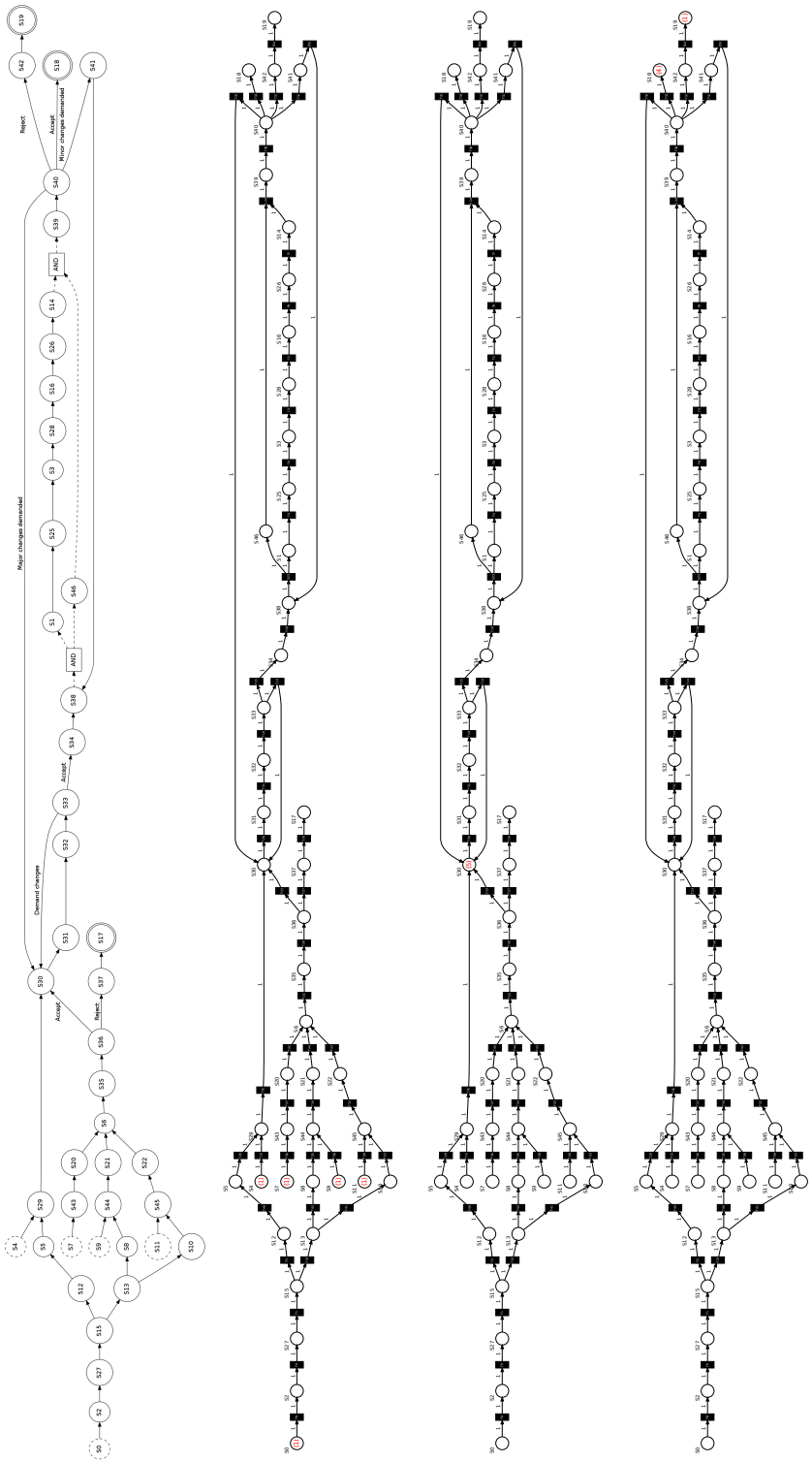


Figure 5.21 – The Faculty's update of an existing educational programme with the graphical correction corresponding to the diagram at the bottom of Figure 5.20, mapped to the model and the corresponding Petri net, with possible intermediate states and final outcomes. Note the potential to have five tokens travel at the same time (second Petri net) and therefore the possibility to have some of them in some states, as in the last diagram. All tokens could as well be in one single state in this particular case.

5.3 Summary

This chapter shows some of the capabilities of the model, the mapping of BPMN to it, and the limitations of the approach as far as semantics are concerned. The last example in particular, in Section 5.2.4, is hard to interpret, although it is entirely valid at the formal level, at least once the mistake has been fixed.

The advantage over the mere execution of BPMN 2.0 is not obvious at this stage, as we limited our research to the execution of Petri nets, not their actual analysis using algorithms for cycle detection (depth-first traversal), reachability of some states or deadlocks. By transforming the diagrams into a purely formal representation, with no semantics, all the toolkit of graph theory and Petri net techniques can be applied directly, including performance analysis techniques such as critical paths.

Chapter 6

Conclusion and Future Work

The contribution of this thesis is in three parts:

1. A model that is both simple enough to represent, yet a formal enough system that it doesn't lose the important properties that would prevent its mapping to a sound mathematical representation.

2. A mapping of the model to such a well-established formal representation, Petri nets, opens the door of automatic analysis and validation of system properties using the numerous techniques described extensively in the literature about Petri nets.

3. A (partial) mapping of BPMN to the model, opens the door to the pragmatic applications for the managerial world.

This effectively bridges the gap between the two worlds of representation and formalisation in the context of processes as it was described in the introduction (figure 1.1 (page 4), as shown in Figure 6.1), where the three contributions are related in a diagram with two axes: the level of formalisation or analysability, and the level of simplicity. BPMN is considered less formal, but simpler, while Petri nets are at the opposite end of the spectrum, i.e. very formal, but complicated. Finite state automata (FSA) are arguably much simpler than Petri nets, but lack the desirable properties, so the gap in between remains. Our model, simple, yet formal enough that it can be mapped in both directions, goes one step further towards bridging that gap.

Future Work

First, the aim of the research is to explore ways in which the processes, like the ones presented in this thesis can be validated automatically, or semi-

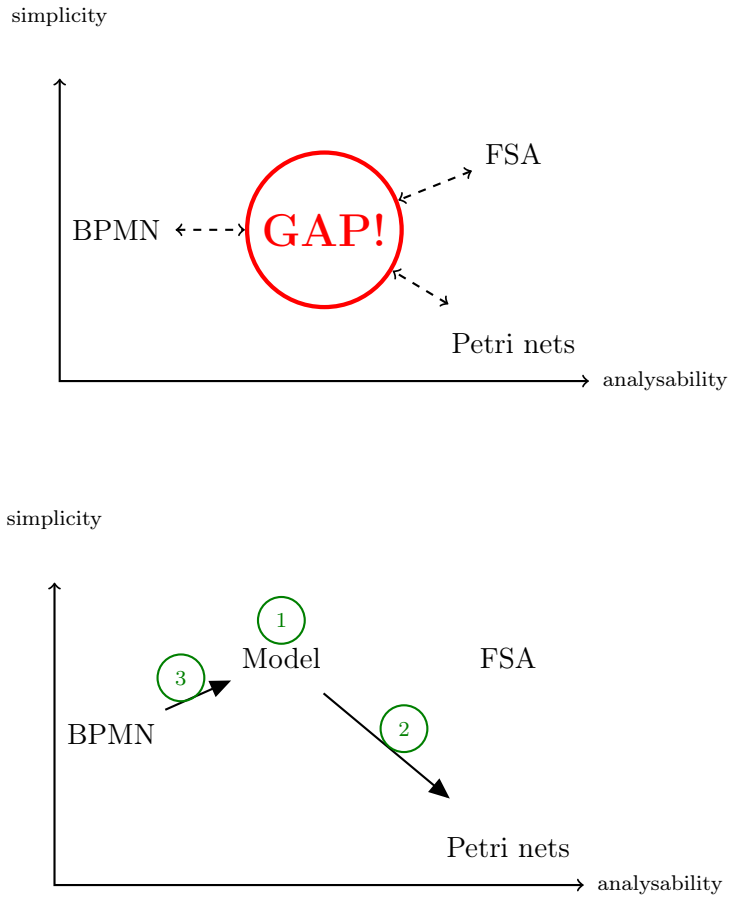


Figure 6.1 – A diagram showing the three contributions of this thesis in relation to the two desirable property characteristics of the two worlds: simplicity, and analysability, and how they contribute to bridging the gap presented in the introduction (page 4). Note that the model is inspired from FSA, but less formal, so the two mappings effectively bridge the gap between the "less analysable" world on the left, and the "more analysable" world on the right.

automatically. For this, the entire arsenal of algorithms applicable to Petri nets can be used, to detect the reachability of states, the cycles, the bottlenecks, the liveliness of the system, or even do performance analysis, among other things. A detailed analysis of these algorithms and their application in this particular context could be an interesting path of research. In particular, the meaning of these properties in the initial model (the processes or work flows), would constitute a good basis to do real validation and allow for a system in the form of an assistant for process design to be built. Also, restrictions have to be taken into account, as many problems in this area exhibit complexities in the general case that are not applicable in practice unless some assumptions are made on the graphs (PSPACE-complete for reachability for example). Some are even undecidable (the equivalence problem for one), so a thorough analysis would represent a huge step further towards validation (or rejection) of the actual applicability of the current proposal to real automatic properties analysis.

It is important to note that with synchronisation mistakes such as the one described in Figure 4.13 (page 59), several tokens get generated and end up accumulating at the final place of the Petri net (in this case an endless amount since the cycle never ends). There is no semantics on the tokens in our models, since we don't use coloured Petri nets, but it could be interesting to investigate a minimal amount of semantics, for example by saying that the token represents some activity in a task management system or ticketing system like Jira. Each time some activity is performed, an entry is placed in the log, and this could be modelled as a token in the corresponding mapped Petri net. Statistics could then be produced on the amount of times an activity is repeated in a cycle, or other metrics of interest.

As already hinted in the introduction and in the argument for our model (Chapter 3, section 3.1), μ-calculus would be another potential candidate to do some model checking in this context. A line of research would then be to try to find a mapping between our model and μ-calculus, and then exploit the representation and its properties.

The assumption in this research considers that BPMN is the frontier of formalism in the management world. This is not exactly true in general, as there exists some algorithms for validating or optimising processes, which goes further in the direction of formal methods. First BPMN 2.0, which appeared during the course of this research, allows the execution of the processes under certain conformance restrictions on the elements, unlike version 1, which was the standard used at the start. It would be worthwhile to see exactly what this execution can bring in terms of validation of the graph properties, compared to the mapping to a sound mathematical nota-

tion such as Petri nets. There could be other possibilities as well: the best known and apparently most favoured algorithmic technique for the validation of management processes is seemingly the critical path method algorithm, developed in the fifties by Morgan R. Walker and James E. Kelley, Jr. Although it is not applicable directly in this case, as it require the execution time to be specified, it would be interesting to define unitary execution times for all activities and investigate potential applications of the algorithm, for example.

Also, this time in the context of web services validation, a language called BPEL, or WS-BPEL (for web-services business process execution language) exists, that could serve the purpose of analysing system properties in the same way we did, by running the Petri nets interactively, or by analysing the reachability graph (state space). The mapping of BPEL to Petri net has been done [33], the authors even proposed a tool to validate BPEL automatically using Petri nets [32], which is exactly what the aim of this research was, but for BPMN. It is unclear if BPEL can be mapped to BPMN though, although some research seems to suggest at least an informal mapping like the one we provide [68, 20], which is a promising direction. A last (unpublished[1]) paper by Ouyang et al. suggests that they overcame the limitations of this type of mapping [31]. This technique will have to be investigated and compared to our approach.

In addition, as already mentioned in the preamble of the chapter on BPMN mapping to Petri nets (Chapter 5), Pi-calculus may be worth exploring, although very controversial [65]. The fact that van der Aalst explained "Why workflow is NOT just a Pi-process" [66] doesn't necessarily mean that the idea cannot be explored. The idea would be to abandon the strict formal soundness of the model, as we did with our extension of finite state automata for that matter, and simply see what can be made of it in practice, regardless of the absence of theoretical or formal validation.

As the first and third example of the mapping from BPMN to the model suggests (Chapter 5 (page 72 and 80 respectively)), the BPMN semantics could also be interesting to interpret in a corresponding Petri net. This would in fact be an essential feature if one were to one day build a kind of assistant, or wizard, to guide in the conception and validation of BPMN using our approach. With semantics, one could go back from the resulting Petri net to the initial model and provide interpretation and guidance as to what is happening. The problem with BPMN is that the mapping will probably always be incomplete, and furthermore, the fact that BPMN allows for the extension of the set of elements renders an exhaustive analysis impossible, unless

[1] as of June 2013

one really restricts oneself to a limited subset of clearly defined elements.

Furthermore, other mappings could be attempted, of virtually any informal, semi-formal or formal representation of work-flow processes. Some proposals have been made that are very similar in nature to our approach, at least in the identification of the gap and the fact that the problem has to be tackled using mappings instead of bringing formalism to informal representation. Most notably, "Bridging the Gap Between Business Models and Workflow Specifications" by van der Aalst [13] identifies many issues in this respect that would be very interesting to investigate. An exhaustive survey of existing or possible mappings would provide statistical validation (or rejection) of the pertinence of our approach as a good trade-off in this particular respect.

Also, experiments could be conducted on users with our approach, as a means of collecting data to evaluate the practical value of this research. The problem with such an experiment would be to define measures and constrain the variables in such a way that valid results are provided. A simple poll would also be interesting as a means of analysing the feasibility and pertinence of a full-scale practical experiment.

Finally, Picard's desirability function could be added to the model. On the BPMN side, a complex gateway can be defined that produces exactly the behaviour of choosing a more likely transition over other ones, and on the side of Petri nets, priorisation or another mechanism can be added to model this. One could imagine a system where parties add new possible sub-processes or call activities to an overall activity, and even put alternative activities in competition with one another to test the possible paths by running the network, to help optimise the tailoring, resulting in a greater adaptability of the process and its modelling.

In conclusion, going back to the problem that initiated this line of research, the gap between the managerial and technical worlds, it is clear that we explored only one single dimension out of many of the differences between the two cultures. Other points of view would certainly be beneficial on this topic, in computer science, but also in the managerial, social, and human domains.

Appendix A

The Prototype

The design of the prototype follows the three successive distinct contributions summarised in the conclusion (Chapter 6). The prototype was built to validate the ideas and provide demo for the funding organisation[1]. One of the requirements was to have a modular infrastructure, where other editors, parsers, or other modules could be plugged in easily and without substantial overhead.

Therefore, the prototype consists of several independent parts, most of which are in fact existing software distributed under the terms of the GPL License, and are listed below. The developed parts were written in the form of Python programs. Figure A.1 illustrates the design.

yEd Diagram editor by yWorks, extended with custom artefacts for the model
`http://www.yworks.com/en/products_yed_about.html`

Eclipse IDE, The Eclipse Foundation, with BPMN/SOA Modelling plugins
`http://www.eclipse.org`

GraphViz Open source graph visualisation software and DOT interpreter and converter
`http://http://www.graphviz.org`

The advantage of such tools is that they are doubly free: free as in "free speech", and free as in "free beer", which is essential in an academic setting. So is Python (see `http://www.python.org`).

[1] Hasler Foundation, see Acknowledgements

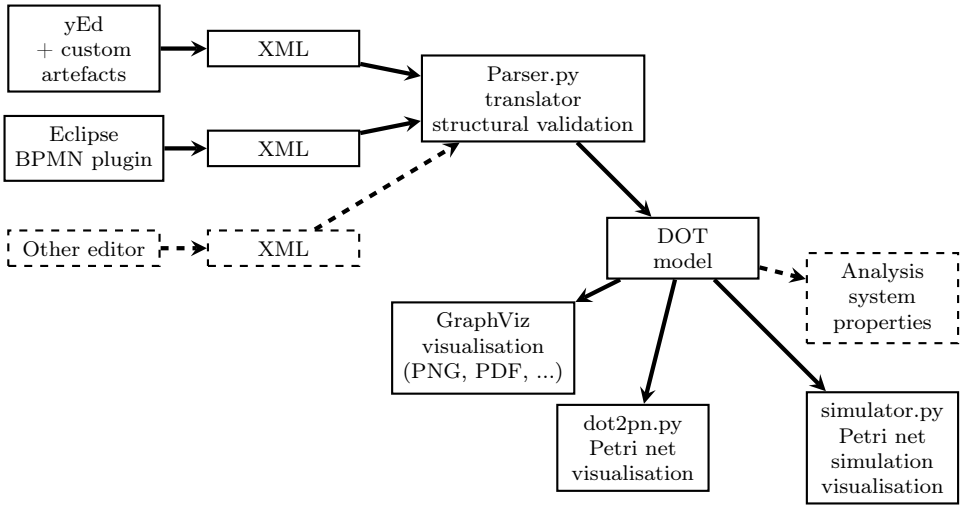


Figure A.1 – The architecture of the prototype. Dashed lines represent future elements that could be added to the framework, according to the possible future work mentioned in the conclusion.

A typical work-flow using the prototype to generate simulations such as the ones shown in Chapter 4, involving the design of a process using the model directly, would look like the following:

1. Edit the process using yEd and the custom artefacts (screenshot in Figure A.2).
2. Transform the XML file into a special DOT file representing the model using Parser.py (the corresponding model is shown in Figure 4.10 (page 55)).
3. Visualise the DOT file with GraphViz, or generate a PNG or a PDF for inclusion in the document.
4. Map the model to a Petri net and execute it using the simulator or simply visualise the resulting Petri net in its initial state as a PNG or PDF using the utility script dot2pn.py (screenshot in Figure A.3)

The other scenario is the edition of BPMN, as in Chapter 5, and it involves the same steps, but using Eclipse and its SOA/BPMN plugin as an editor:

1. Edit the process using Eclipse (screenshot in Figure A.4).
2. Transform the XML file into a special DOT file representing the model using Parser.py.

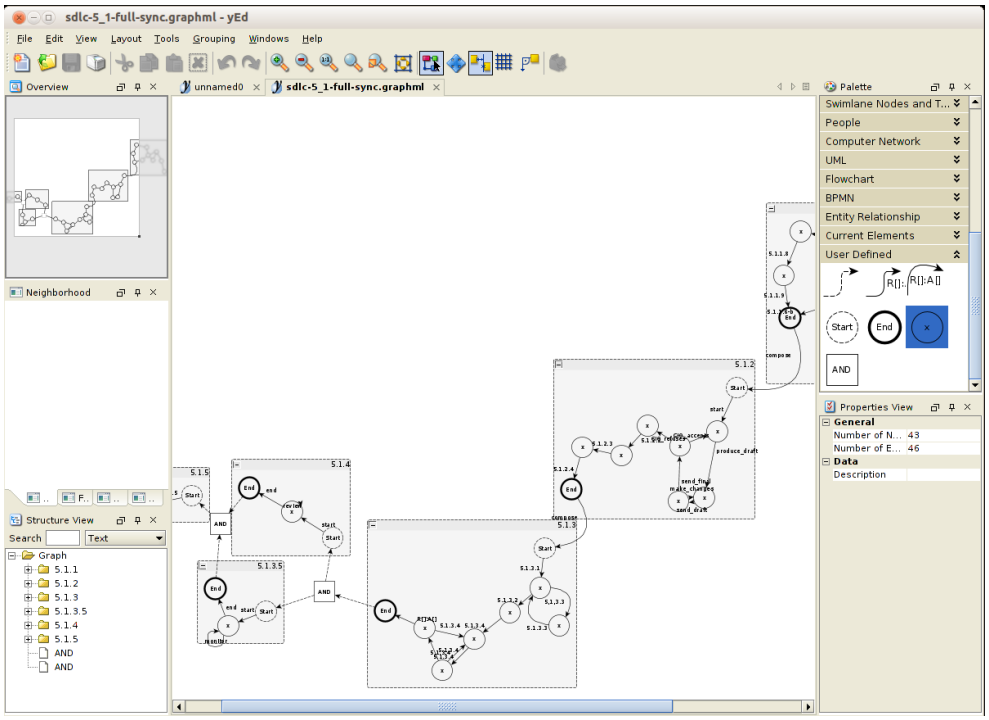


Figure A.2 – A session of the yEd editor with the custom artifacts being used to model a process of the IEC/IEEE 12207 document [22].

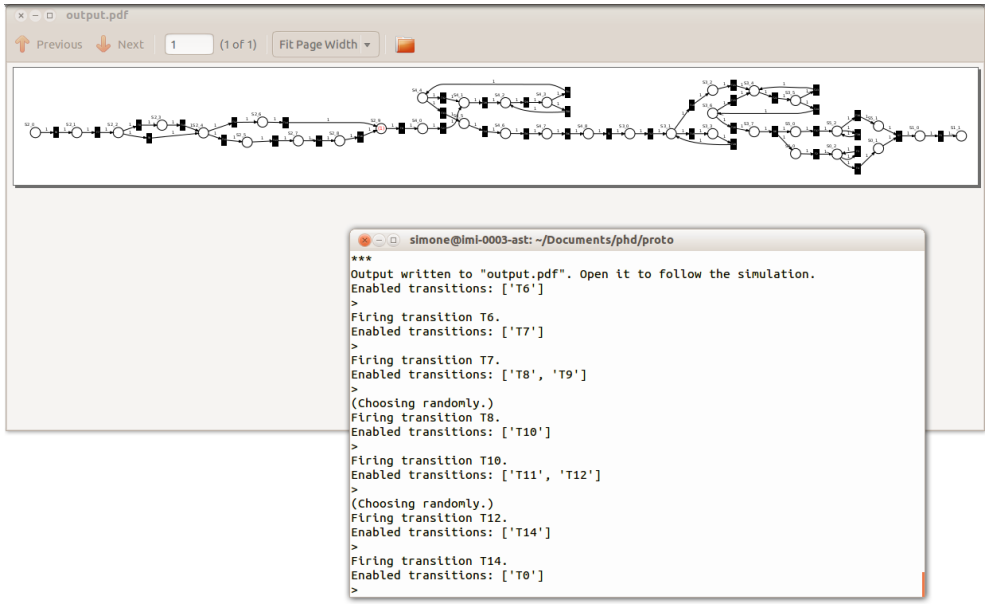


Figure A.3 – A simulator session with the resulting Petri net for the modelling in Figure A.2.

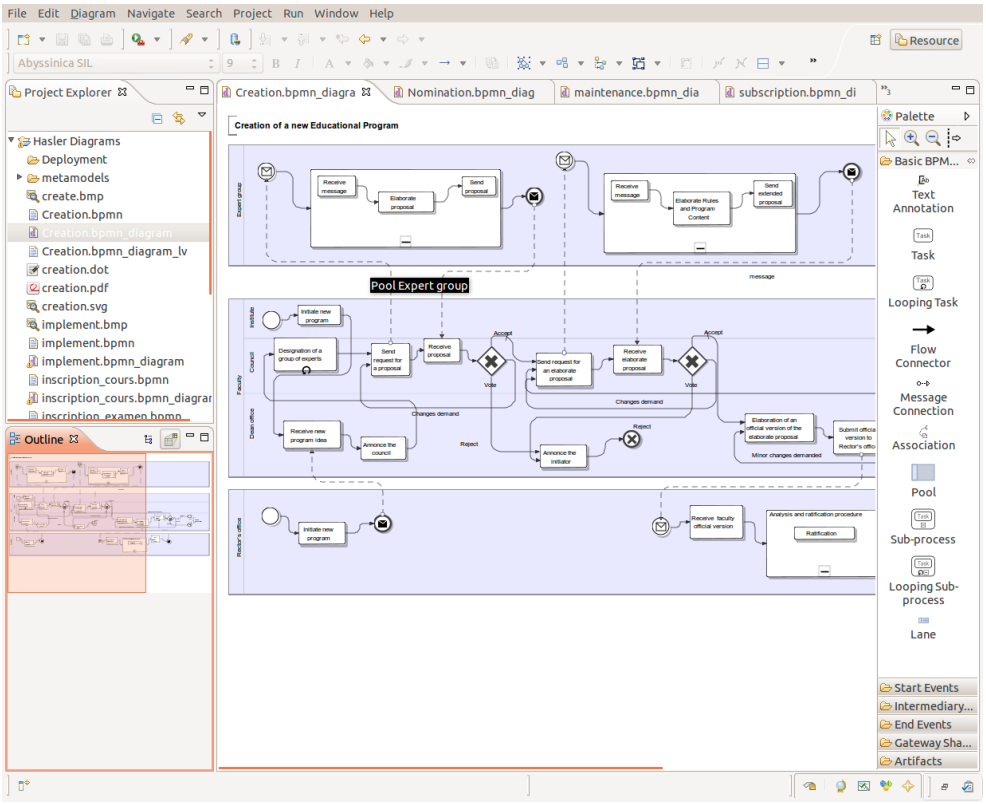


Figure A.4 – A session of Eclipse with the BPMN editor.

3. Visualise the DOT file with GraphViz, or generate a PNG or a PDF for inclusion in the document.

4. Map the model to a Petri net and execute it using the simulator or simply visualise the resulting Petri net in its initial state as a PNG or PDF using the utility script dot2pn.py (screenshot in Figure A.5)

Any other diagram editor could be used, like Microsoft Visio, Sparx Enterprise Architect or any other, provided the format of the files is XML, and the Parser is adapted to understand it.

The model itself was saved in the DOT language[2]. The choice not to use XML was that the resulting file is much easier to understand, and also very easy to parse.

DOT is a pure graph description language aimed at representation and doesn't provide any semantics, so to distinguish start, intermediate, and end nodes, simple labels are used. The special "AND" node and its corresponding

[2] `http://www.graphviz.org/content/dot-language`

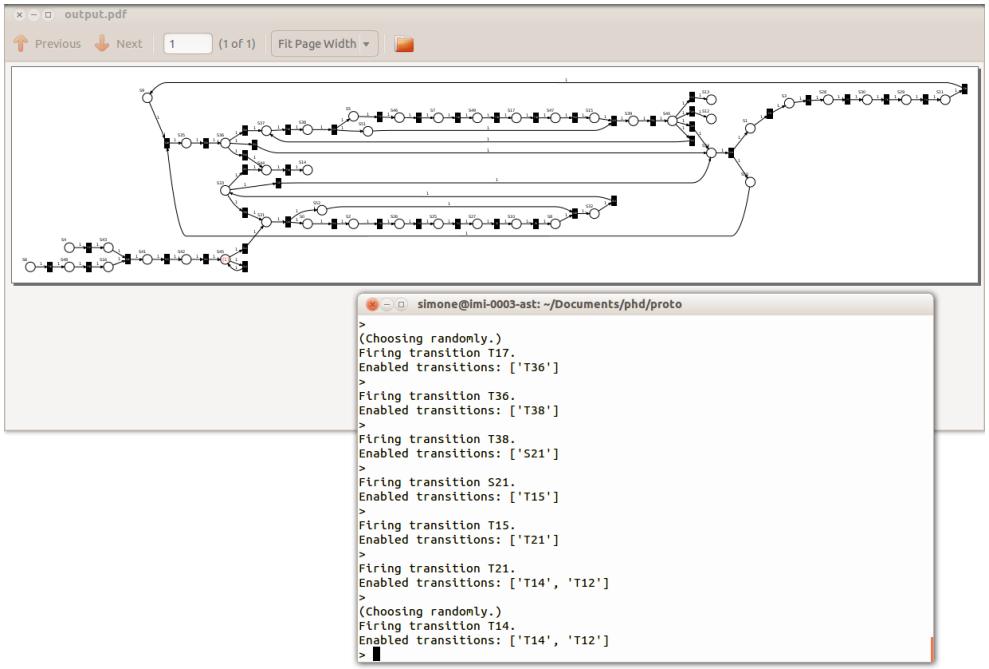


Figure A.5 – A simulator session with the resulting Petri net resulting for the model in Figure A.4.

dashed arcs are also distinguished by a label. The graphical representation is then added accordingly, depending on the external semantics, using DOT standard notations.

A complete specification of the way the DOT language is used to store the model was written by my colleague Paul Cotofrei for the reporting to the Hasler Foundation, which funded the research. It is provided as a reference in Appendix B.

Appendix B

Parser Specifications

LCM-RT Specification[1]

Information Management Institute
University of Neuchâtel

01 January 2009

[1]This work was supported by the Hasler Foundation (grant ManCom 2085).

Chapter 1

Specifications for DOT file

The DOT file is obtained by applying a dedicated parser on a diagram created with a graphical visual editor and saved under a format of type XML. In order to assure a standard input for the LCRMT Module (Petri Nets Validator) independently of the user diagram editor (see Figure 1.1), each dedicated parser must generate as output a DOT file satisfying the following specifications:

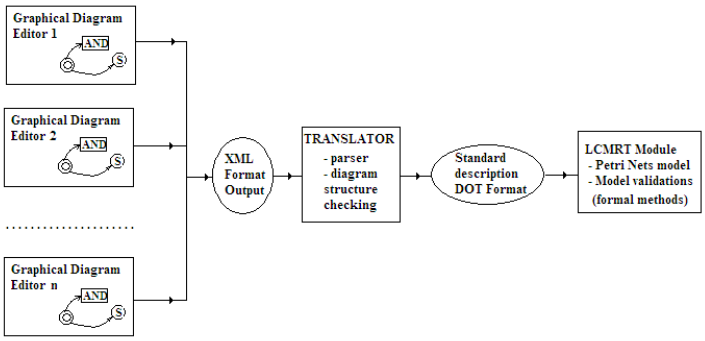


Figure 1.1: From SSP graphical representation to Petri Nets representation using DOT language as standard description

- Node names: each node of the directed graph is named

  `S<nr>`

  where <nr> is a sequence containing digits and the character '_', generated by an indexing procedure, according to the rules:

  1. If the diagram designed by the user doesn't contains any sub-diagrams (or "component state machines", in the terms of scalable state machine), the index of each node is a natural number from the set $\{0,1,...,n-1\}$, where n is the total number of nodes in the diagram (see an example in Figure 1.2. There is no preference in the mechanism of assigning indexes to nodes (e.g., it's no mandatory that the start node is indexed as '0')
  2. If the diagram contains m sub-diagrams, a node from the sub-diagram $j \in \{0,..,m-1\}$ is indexed as 'j_i', where $i \in \{0,1,..,k-1\}$ and k is the total number of nodes from sub-diagram j. The nodes which are not components of any sub-diagram are indexed using the previous rule, by considering n as the total number of nodes in the diagram, not included in any sub-diagram (see an example in Figure 1.3).

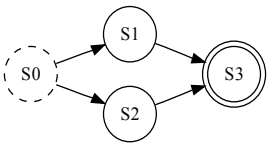


Figure 1.2: Example of diagram without sub-diagrams: index nodes from 0 to 3.

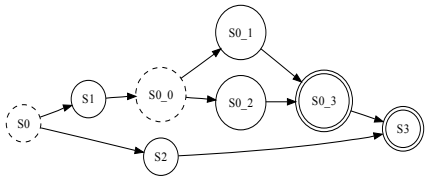


Figure 1.3: Example of diagram with a sub-diagram. Index nodes from the sub-diagram 0: from 0_0 to 0_3.

- Header:

```
digraph G{
rankdir = LR;
```

- Nodes of type "Start":

```
/* Start nodes */
node[shape = circle, style = dashed, peripheries = 1];
S<nr>[label="S<nr>"];
```

  where <nr> represents the index of the node.

- Nodes of type "Stop":

```
/* End nodes */
node[shape = circle, style = solid, peripheries = 2];
S<nr>[label = "S<nr>"];
```

- Ordinary nodes:

```
/* normal nodes */
node[shape = circle, style = solid, peripheries = 1];
S<nr>[label = "S<nr>"];
```

- Nodes of type "AND":

```
/* AND nodes */
node[shape = box, style = solid, peripheries = 1, label = "AND"];
S<nr>;
```

The only rule imposed to the order in which the different types of nodes are included in the DOT file is that the "AND" nodes are in the last position.

- Normal edges (not implying a node type "AND"):

```
/* normal edges */
S<nr1> -> S<nr2>[label = "<label>"];
```

where <nr1> and <nr2> are the indexes of the two nodes and <label> is the label (possible empty) of the corresponding edge.

- Edges linking "AND" nodes:

```
/* AND edges */
edge[style = dashed];
S<nr1> -> S<nr2>;
```

Examples:

1. The DOT file corresponding to the diagram from Figure 1.2

```
digraph G{
rankdir = LR;
/* Start nodes */
node[shape = circle, style = dashed, peripheries = 1];
S0[label = "S0"];
/* End nodes */
node[shape = circle, style = solid, peripheries = 2];
S3[label = "S3"];
/* normal nodes */
node[shape = circle, style = solid, peripheries = 1];
S1[label = "S1"];
S2[label = "S2"];
/* normal edges */
S0 -> S1;
S0 -> S2;
S1 -> S3;
S2 -> S3;
}
```

2. The DOT file corresponding to the diagram from Figure 1.3

```
digraph G{
rankdir = LR;
/* Start nodes */
node[shape = circle, style = dashed, peripheries = 1];
S0[label = "S0"];
S0_0[label = "S0_0"];
```

```
/* End nodes */
node[shape = circle, style = solid, peripheries = 2];
S3[label = "S3"];
S0_3[label = "S0_3"];
/* normal nodes */
node[shape = circle, style = solid, peripheries = 1];
S1[label = "S1"];
S2[label = "S2"];
S0_1[label = "S0-1"];
S0_2[label = "S0_2"];
/* normal edges */
S0 -> S1[label = ""];
S0 -> S2;
S1 -> S0_0;
S0_0 -> S0_1;
S0_0 -> S0_2;
S0_1 -> S0_3;
S0_2 -> S0_3;
S0_3 -> S3;
S2 -> S3;
}
```

3. The DOT file corresponding to the diagram from Figure 1.4

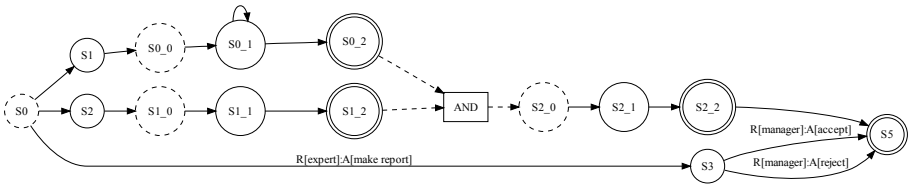


Figure 1.4: Example of a complex diagram with three sub-diagrams and nodes of type "AND".

```
digraph G{
rankdir = LR;
/* Start nodes */
node[shape = circle, style = dashed, peripheries = 1];
S0[label = "S0"];
S0_0[label = "S0_0"];
S1_0[label = "S1_0"];
S2_0[label = "S2_0"];
/* End nodes */
node[shape = circle, style = solid, peripheries = 2];
S5[label = "S5"];
S0_2[label = "S0_2"];
S1_2[label = "S1_2"];
S2_2[label = "S2_2"];
/* normal nodes */
node[shape = circle, style = solid, peripheries = 1];
S1[label = "S1"];
S2[label = "S2"];
```

```
S3[label = "S3"];
S0_1[label = "S0_1"];
S1_1[label = "S1_1"];
S2_1[label = "S2_1"];
/* AND nodes */
node[shape = box, style = solid, peripheries = 1, label = "AND"];
S4;
/* normal edges */
S0 -> S1;
S0 -> S2;
S0 -> S3[label = "R[expert]:A[make report]"];
S1 -> S0_0;
S0_0 -> S0_1;
S0_1 -> S0_1;
S0_1 -> S0_2;
S2 -> S1_0;
S1_0 -> S1_1;
S1_1 -> S1_2;
S2_0 -> S2_1;
S2_1 -> S2_2;
S2_2 -> S5;
S3 -> S5[label = "R[manager]:A[accept]"];
S3 -> S5[label = "R[manager]:A[reject]"];
/* AND edges */
edge[style = dashed];
S0_2 -> S4;
S1_2 -> S4;
S4 -> S2_0;
}
```

Bibliography

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, UK, 1996.

[2] J. R. Abrial and al. *On the Construction of Programs*. Cambridge University Press, UK, 1980.

[3] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of Recursive State Machines. In *ACM Transactions on Programming Languages and Systems*, volume 27, pages 786–818, 2005.

[4] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, MA, 1999. ISBN 0321278658.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile Software Development, 2001 (checked December 2013). http://agilemanifesto.org/.

[6] D. Bjørner and C. B. Jones. The Vienna Development Method: The Meta-Language. In *Lectures Notes in Computer Science*, volume 61, Berlin Heidelberg, Germany, 1978. Springer-Verlag.

[7] B. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11, August 1986.

[8] BPMNcommunity. BPMN 2.0 Tutorials, checked December 2013. http://en.bpmn-community.org/tutorials.

[9] F. P. Brooks. *The Mythical Man Month*. Addison-Wesley, 1975.

[10] F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20:10–19, April 1987.

[11] J. Carroll and D. Long. *Theory of Finite Automata with an Introduction to Formal Languages*. Prentice Hall, Englewood Cliffs, 1989.

[12] P. Cotofrei and K. Stoffel. Business Process Modelling for Academic Virtual Organizations. In L. M. Camarinha-Matos and W. Picard, editors, *Pervasive Collaborative Networks*, volume 283 of *IFIP Advances in Information and Communication Technology*, pages 213–220. Springer, Boston, 2008. ISBN: 978-0-387-84836-5.

[13] J. Dehnert and W. M. P. van der Aalst. Bridging the Gap Between Business Models and Workflow Specifications. *International Journal of Cooperative Information Systems*, 13(03):289–332, 2004.

[14] DSDM Consortium, The. Dynamic Systems Development Method, checked December 2013. http://www.dsdm.org/.

[15] E. A. Emerson. Model checking and the mu-calculus. In *DIMACS Series in Discrete Mathematics*, pages 185–214. American Mathematical Society, 1997.

[16] H. Genrich, K. Lautenbach, and P. Thiagarajan. Elements of general net theory. In W. Brauer, editor, *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science*, pages 21–163. Springer Berlin Heidelberg, 1980.

[17] A. Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.

[18] S. Ginsburg. *An Introduction to Mathematical Machine Theory*. Addison-Wesley, 1962.

[19] J. A. Highsmith. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House, New York, 2000. ISBN 0932633404.

[20] S. Hinz, K. Schmidt, and C. Stahl. Transforming bpel to petri nets. In *Proceedings of the International Conference on Business Process Management (BPM2005), volume 3649 of Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, 2005.

[21] Institute or Electrical and Electronics Engineers (IEEE). IEEE/EIA 12207.0-1996. Software Life Cycle Processes. Industry Implementation of International Standard ISO/IEC 12207: 1995, March 1998. ISBN 0-7381-0428-0, SS94581 (PDF).

[22] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). ISO/IEC 12207. Standard for Information Technology, 1995.

[23] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, Germany, 1992.

[24] E. C. Jr., O. Grumberg, and D. A. Peled. MIT press, Cambridge, Massachusetts, USA, 1999. ISBN: 0-262-03270-8.

[25] A. Martin. Relating Z and First-Order Logic. In *Lectures Notes in Computer Science*, volume 1709, page 715, Berlin Heidelberg, Germany, 2004. Springer.

[26] J. Martin. *Rapid Application Development.* Macmillan Coll Div, May 1991. ISBN 0023767758.

[27] E. W. Mayr. An algorithm for the general petri net reachability problem. *SIAM J. Comput.*, 13(3):441–460, 1984.

[28] H. Mills. Top-down programming in large systems. *Debugging Techniques in Large Systems*, pages 41–55, 1971.

[29] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.

[30] Y. U. D. of Computer Science and R. Lipton. *The reachability problem requires exponential space.* Research report (Yale University. Dept. of Computer Science). Department of Computer Science, Yale University, 1976.

[31] C. Ouyang, M. Dumas, W. M. P. v. d. Aalst, A. H. M. T. Hofstede, and J. Mendling. From business process models to process-oriented software systems. *ACM Trans. Softw. Eng. Methodol.*, 19(1):2:1–2:37, Aug. 2009.

[32] C. Ouyang, E. Verbeek, S. Breutel, M. Dumas, and A. H. M. T. Hofstede. Wofbpel: A tool for automated analysis of bpel processes. In *ICSOC 2005 proceedings, December, Lecture Notes in Computer Science*, pages 484–489. Springer, 2005.

[33] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. T. Hofstede. Formal semantics and analysis of control flow in ws-bpel, 2007.

[34] S. R. Palmer and J. M. Felsing. *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002. ISBN 0-130-67615-2.

[35] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Schriften IIM, Institut für Instrumentelle Mathematik, University of Bonn, Germany, 1962.

[36] W. Picard. Computer support for adaptive human collaboration with negotiable social protocols. In W. Abramowicz and H. C. Mayr, editors, *BIS*, volume 85 of *LNI*, pages 90–101. GI, 2006.

[37] W. Picard. An algebraic algorithm for structural validation of social protocols. In W. Abramowicz, editor, *BIS*, volume 4439 of *Lecture Notes in Computer Science*, pages 570–583. Springer, 2007.

[38] W. Picard. Modelling multithreaded social protocols with coloured petri nets. In L. M. Camarinha-Matos and W. Picard, editors, *Virtual Enterprises and Collaborative Networks*, volume 283 of *IFIP*, pages 343–350. Springer, 2008.

[39] W. Picard. Computer support for agile human-to-human interactions with social protocols. In W. Abramowicz, editor, *BIS*, volume 21 of *Lecture Notes in Business Information Processing*, pages 121–132. Springer, 2009.

[40] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley Professional, 1st edition, May 2003. ISBN 0321150783.

[41] O. O. Prisecaru. Petri-nets Based Approaches for the Modelling and Verification of Workflow Processes, January 2009.

[42] F. Puhlmann. Why do we actually need the pi-calculus for business process management. In *9th International Conference on Business Information Systems (BIS 2006), volume P-85 of LNI, Bonn, Gesellschaft für Informatik (2006) 77–89*, 2006.

[43] F. Puhlmann. On the Suitability of the Pi-Calculus for Business Process Management. In W. Abramowicz and H. C. Mayr, editors, *Technologies for Business Information Systems*, pages 51–62. Springer-Verlag, Netherlands, May 2007. ISBN: 978-1-4020-5633-8.

[44] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3, 1959.

[45] L. B. S. Raccoon. The chaos model and the chaos life cycle. *ACM Software Engineering Notes*, 20(1):55–66, January 1995.

[46] Raise Method Group. *The Raise Specification Language*. Prentice-Hall, US, 1992.

[47] Raise Method Group. *The Raise Method Manual*. Prentice-Hall, US, 1995.

[48] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, Berlin Heidelberg, Germany, 1985.

[49] W. Reisig. *Elements of Distributed Algorithms. Modeling and Analysis with Petri Nets*. Springer-Verlag, 1998.

[50] W. W. Royce. Managing the development of large software systems. *IEEE WESCON*, pages 1–9, August 1970.

[51] E. Simon, C. Künzi, and K. Stoffel. Scalable Social Protocols to Formalize Systems Development Life Cycles. In P. Kommers, editor, *Proceedings of IADIS International Conference e-Society*, pages 177–184, Lisbon, July 2007.

[52] E. Simon and K. Stoffel. State Machines and Petri nets as a Formal Representation for Systems Life Cycle Management. In M. B. Nunes, P. Isaías, and P. Powell, editors, *Proceedings of IADIS International Conference Information Systems 2009*, pages 275–282, Barcelona, February 2009. IADIS Press.

[53] E. Simon and K. Stoffel. Towards Bridging the Gap Between Intuitive and Formal Representations of Systems Life Cycle Processes. *IADIS International Journal on Computer Science and Information Systems*, 4(3):39–50, November 2009. ISSN: 1646-3692.

[54] H. Smith and P. Fingar. Workflow is just a pi process, 2003 (checked December 2013). http://bpmg.orgwww.bptrends.com/publicationfiles/01-04Workflow%20is%20just%20a%20Pi%20Process%20Smith-Fingar.pdf.

[55] J. F. Sowa. Conceptual Graphs for a Data Base Interface. *IBM Journal of Research and Development*, 20(4):336–357, 1976.

[56] H. Takeuchi and I. Nonaka. The New New Product Development Game. *Harvard Business Review*, 64(l):137–146, January-February 1986.

[57] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[58] The Object Management Group (OMG). BPMN 2.0, November 2011. http://www.omg.org/.

[59] The US Department of Justice. Systems Development Life Cycle Guidance Document, January 2003 (checked December 2013). http://www.usdoj.gov/jmd/irm/lifecycle/table.htm.

[60] Unité de pilotage informatique de la Confédération (UPIC). HERMES 5, checked December 2013. http://www.hermes.admin.ch/.

[61] Unité de pilotage informatique de la Confédération (UPIC). HERMES 2003/7, superseded by Hermes 5 (Mai 2013). http://www.hermes.admin.ch/.

[62] W. M. P. van der Aalst. *A Class of Petri Nets for Modeling and Analyzing Business Processes.* Computing science reports. Eindhoven University of Technology, Department of Mathematics and Computing Science, 1995.

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

[64] W. M. P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques, 2000.

[65] W. M. P. van der Aalst. Pi calculus versus petri nets: Let us eat "humble pie" rather than further inflate the "pi hype", 2003.

[66] W. M. P. van der Aalst. Why workflow is NOT just a Pi process, 2004 (checked December 2013). http://bptrends.com/publicationfiles/02-04%20ART%20WhyworkflowisNOTjustaPi%20-%20Aalst1.pdf.

[67] W. M. P. van der Aalst, K. V. Hee, P. D. Kees, M. Hee, R. R. D. Vries, J. Rigter, E. Verbeek, and M. Voorhoeve. Workflow management: Models, methods, and systems, 2002.

[68] S. A. White. Using BPMN to Model a BPEL Process, checked December 2013. http://www.omg.org/bpmn/Documents/Mapping_BPMN_to_BPEL_Example.pdf.

Publications by the Author

[1] Eric Simon and Kilian Stoffel. Towards Bridging the Gap Between Intuitive and Formal Representations of Systems Life Cycle Processes. *IADIS International Journal on Computer Science and Information Systems*, 4(3):39–50, November 2009. ISSN: 1646-3692.

[2] Eric Simon and Kilian Stoffel. State Machines and Petri nets as a Formal Representation for Systems Life Cycle Management. In Miguel Baptista Nunes, Pedro Isaías, and Philip Powell, editors, *Proceedings of IADIS International Conference Information Systems 2009*, pages 275–282, Barcelona, February 2009. IADIS Press.

[3] Eric Simon, Christophe Künzi, and Kilian Stoffel. Scalable Social Protocols to Formalize Systems Development Life Cycles. In Piet Kommers, editor, *Proceedings of IADIS International Conference e-Society*, pages 177–184, Lisbon, July 2007.

[4] Eric Simon, Iulian Ciorăscu, and Kilian Stoffel. An Ontological Document Management System. In Witold Abramowicz and Heinrich C. Mayr, editors, *Technologies for Business Information Systems*, pages 313–325. Springer-Verlag, Netherlands, May 2007. ISBN: 978-1-4020-5633-8.

[5] Eric Simon, Iulian Ciorăscu, and Kilian Stoffel. Concept and Implementation of an Ontological Document Management System. In *Proceedings of the International Conference on Business Information Systems (BIS)*, volume P-85 of *Lecture Notes in Informatics*, pages 403–414. Gesellschaft für Informatik, Klagenfurt, Austria, June 2006.

[6] Eric Simon. Forecasting Foreign Exchange Rates with Neural Networks. Diploma (M.Sc.) thesis, Laboratoire d'intelligence artificielle LIA, Ecole polytechnique fédérale de Lausanne EPFL, Lausanne, Switzerland, February 2002. http://liawww.epfl.ch/uploads/project_reports/report_282.pdf (checked December 2013).

[2]Outstanding Paper Award