UNIVERSITÉ DE GENÈVE

Archive ouverte UNIGE
https://archive-ouverte.unige.ch

Thèse 2009

Open Access

This version of the publication is provided by the author(s) and made available in accordance with the copyright holder(s).

A multi-dimensional compositional approach for business process semantic engineering

Chen, Ang

How to cite

CHEN, Ang. A multi-dimensional compositional approach for business process semantic engineering. Doctoral Thesis, 2009. doi: 10.13097/archive-ouverte/unige:12252

This publication URL: https://archive-ouverte.unige.ch/unige:12252
Publication DOI: 10.13097/archive-ouverte/unige:12252

© This document is protected by copyright. Please refer to copyright holder(s) for terms of use.

UNIVERSITÉ DE GENÈVE
Département d'Informatique

FACULTÉ DES SCIENCES
Professeur D. Buchs

A Multi-Dimensional Compositional Approach for Business Process Semantic Engineering

THÈSE

présentée à la Faculté des sciences de l'Université de Genève
pour obtenir le grade de Docteur ès sciences, mention informatique

par

Ang Chen
de
Henan (Chine)

Thèse No 4192

Genève
Atelier d'impression ReproMail
2009

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Doctorat ès sciences
mention informatique

Thèse de *Monsieur Ang CHEN*

intitulée :

" A Multi-Dimensional Compositional Approach for Business Process Engineering "

La Faculté des sciences, sur le préavis de Messieurs D. BUCHS, professeur associé et directeur de thèse (Département d'informatique), N. GUELFI, professeur (Université du Luxembourg – Faculté des Sciences, de la Technologie et de la Communication - Luxembourg-Kirchberg, Luxembourg), et Ph. DUGERDIL, docteur (Département d'informatique), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 12 mars 2010

Thèse - 4192 -

Le Doyen, Jean-Marc TRISCONE

N.B.- La thèse doit porter la déclaration précédente et remplir les conditions énumérées dans les "Informations relatives aux thèses de doctorat à l'Université de Genève".

Table of Contents

Remerciements

D'abord, je profite de cette occasion pour adresser mes sincères remerciements au professeur Didier Buchs, mon Directeur de thèse, pour m'avoir transmis des compétences scientifiques essentielles à la réalisation de ce travail, dont je profiterai dans l'avenir. Je le remercie de sa patience, de son intérêt et de sa disponibilité toujours manifestés pour les éclaircissements sollicités au long de cette grande aventure.

Je tiens également à remercier vivement les collégues du groupe SMV à l'Université de Genève: Steve Hostettler, Levi Lùcio, Luis Pedro, Matteo Risoldi, Alexis Marechal, qui m'ont donné des commentaires constructifs et m'ont aidé à clarifier les idées pendant les séminaires et les discussions quotidiennes. Je voudrais remercier particulièrement Dr. Levi Lùcio d'avoir accepté de reviser le manuscrit pendant la rédaction de ce travail.

Mes remerciements vont aussi à M. Phillipe Dugerdil et M. Nicolas Guelfi, qui m'ont fait l'honneur d'accepter d'être les jurés de ce travail.

Enfin, je remercie de tout mon coeur, Jianghua, qui m'a encouragé et soutennu au cours de ces denières années.

à mes parents et mes grand parents

Abstract

Business processes are descriptions of how information should be organized and exchanged during business activities between the business process participants, e.g. workflow and business protocols. A business process is a complex, multi-dimensional entity which represents the designer's high-level objectives, and it may have several dimensions such as control flow, data model/flow, organizational perspective, and resource perspective.

In many business domains, such as global markets, business processes can (and should) be automatized and deployed into computer systems to facilitate the information exchange between human and interconnected heterogenous computer systems. In these cases, *executable business processes* are long-lasting, parallel computer programs running in concurrent and distributed environments. Moreover, the business processes should be verified and tested thoroughly before being deployed, especially for critical systems.

The development of business processes reflects the cycles of general software development with some specificities. From software engineers' point of view, business process engineering should be itself an iterative process which consists of modeling, prototyping, verifying, and executing of business processes in an incremental, rigorous, systematic, and efficient way. It has the objective of rapidly adopting new business processes (new requirements) and the modification of existing business processes (e.g. change of business protocols). Due to the complexity of models and the frequent change of requirements in business domains, this objective is an open problem thus a challenge in software development.

In this work, we propose a methodological framework to improve the cycle of business process engineering, i.e. a multi-dimensional compositional approach based on formal notation (Petri nets), the principle of Multi-Dimensional Separation of Concerns (MDSoC), and Model-Driven Development (MDD). The core element in this approach is an extensible intermediate language called *ID-net*.

Derived from classical Petri nets, ID-net uses identification tokens which can be values or resource references. Designed by the principe of MDSoC, ID-net captures

the control flow of business process and monitors the resources involved during the execution of processes. The concrete resources are managed by the models from other dimensions, i.e. identification tokens are created and interpreted by the models outside of ID-net called *co-models of ID-net*. Unlike other business process models which are designed without the notion of concurrency, ID-nets have true concurrency semantics and they can be executed in parallel distributed environment.

The incremental development of business processes or concurrent executable programs is done via the composition (synchronization) of ID-net with the co-models which manage the corresponding resources. Syntactically, it is realized by annotating ID-net using domain specific languages (DSL). The composed model, called ID-net Controlled System (IDCS), will respects the constraints implied by the models from all the problem dimensions. Our approach is able to handle the control flow, transactions (short-lasting and long-lasting), concurrency of business processes, and incorporate different data models during the development of executable business process via the composition of dimensions. We give the formal semantics of ID-net and the composition mechanism.

At any stage of the development, each dimension of an IDCS can be verified independently from others. ID-nets can be easily mapped to classical Petri nets in order to profit from the abundant model verification techniques developed for classical Petri nets developed in the past decades.

Resumé

Les processus métiers sont des descriptions de la façons dont l'information devraient être organisées et échangées au cours des activités de métier entre les participants, par exemple le flux de travail (workflow) et les protocoles commerciaux. Un processus métier est une entité complexe et multi-dimensionnelle qui représente les objectives du concepteur de haut-niveaux, et il peut avoir plusieurs dimensions telles que le flux de contrôle, le flux de données, la perspective organisationnelle et la perspective des ressources.

Dans nombreux domaines métiers comme les marchés mondiaux, les processus métiers peuvent (et doivent) être automatisés ensuite déployés dans les systémes informatiques afin de faciliter l'échange d'information entre l'homme et les systémes informatiques hétérogènes interconnectés. Dans ces cas, les processus métiers exécutables sont des programmes parallèles de longe durée fonctionnant dans des environnements concurrents et distribués. En autre, les processus doivent être validés et testé avant d'être déployés, notamment pour les systémes critiques.

Le développement de processus métiers reflète le cycle de développement des logiciels généraux avec quelques spécificités. Du point de vu génie logiciel, l'ingénierie de processus métiers devrait être en soi un processus itérative qui consistes à la modélisation, le prototypage, la vérification et l'exécution de processus d'une façon progressive, rigoureuse, systématique et efficace. Elle a pour objective d'adopter rapidement de nouveaux processus métiers (nouveaux besoins) et les modifications des processus métiers existants (par ex. changement de protocoles commerciaux). En raison de la complexité des modèles et les fréquents changements de besoins dans les domaines métiers, cet objectif est une problème ouvert ainsi un défi pour le développement de logiciels.

Dans ce travail, nous proposons un cadre méthodologique pour améliorer le cycle de l'ingénierie des processus métiers, c'est à dire une approche multi-dimensionnelle de composition basée sur la notation formelle (Réseaux de Petri), le principe de séparation de problème en dimensions et le développement piloté par les modèles. L'élément de base dans de cette approche est un langage intermédiaire extensible appelé ID-net.

Dérivé de Réseaux de Petri classiques, ID-net utilise des jetons d'identification qui peuvent être des valeurs ou des références de ressources. Conçu par le principe de séparation de problème en dimensions, ID-net capte le flux de contrôle des processus métiers et contrôle des ressources nécessaires pendant l'exécution des processus. Les ressources concrètes sont grées par les modèles d'autres dimensions, à savoir l'identification des jetons sont crées et interprétées par les modèles à l'extérieur de l'ID-net appelé co-modèles de ID-net. Contrairement aux autres modèles de processus métiers qui sont conçu sans la notion de concurrence, la sémantique de ID-net est basé sur la "vraie concurrence" et les modèles peuvent être exécutés en environnement parallèle et distribué.

Le développement progressif de processus métiers ou de programmes concurrents se fait via la composition (la synchronisation) de ID-net avec les co-modèles qui gèrent les ressources correspondants. Syntaxiquement, il est réalisé par annoter ID-net en utilisant des langages spécifiques de domaines. Le modèle composé, appelé un système contrôlé par ID-net, sera respecte les contraints impliquées par les modèles de toutes les dimensions de problème. Notre approche est capable de gérer le flux de contrôle, les transactions (de courte durée et de longe durée), la concurrence entre les processus, et d'intégrer différents modèles de données pendant le développement de processus métier, via la composition de dimensions. Nous donnons la sémantique formelle de ID-net ainsi que celle du mécanisme de composition.

À n'importe quelle stade du développement, chaque dimension du système peut être vérifiée de façon indépendamment des autres. ID-nets peuvent être facilement transformés en Réseaux de Petri classiques afin de profiter les nombreux techniques de vérification élaboré pendant ces dernières années.

Chapter 1

Introduction

1.1 Business Process Reengineering

Business process reengineering (BPR) is, in computer science and management, an approach aiming at improvements by means of elevating efficiency and effectiveness of the business process that exist within and across organizations. Also known as Business Process Change Management, BPR is a fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in cost, quality, speed, and service. A key stimulus for reengineering has been the continuing development and deployment of sophisticated interconnected information systems [1].

Business process modeling in systems engineering and software engineering is the activity of representing processes of an enterprise to get better manageability, so that the current processes may be analyzed and improved in future. Business process modeling is typically performed by business analysts and managers who are seeking to improve process efficiency and quality. The process improvements may or may not require the involvement of information technology, although that is a common choice. In many business domains, business processes can (and should) be automatized and deployed into computer systems to facilitate the information exchange between human and interconnected heterogenous computer systems.

In software engineering, business process engineering or business process management consists of discovering and modeling real-world business processes by means of some *business process modeling languages (BPMLs)*, in order to execute, control, and monitor business processes using computer systems. A business process specified by these languages is called *business process model*, while concretely each executing business process is called an *instance* or *case* of the business process model. A

BPML is a domain-specific language (DSL) for business process.

Business Process Models Based on BPML, business process models are artifacts which describe how information should be organized, processed, and exchanged during business activities between the business process participants. Driven by objectives, business process models usually represent the sequences of expected actions or events in existing information systems, i.e. the plan to achieve the objectives. They also represent the relationships such as dependency and causality between the actions or events. The state of a process instance evolves when expected events occur, e.g. user inputs, task finished; the states of running processes are monitored and controlled explicitly by the business process management system (BPMS).

As depicted in figure 1.1, a business process model has some related elements during its life cycle, e.g.:

- Designer, the domain expert who defines the process.
- Participants, who have capabilities of working on a particular task, e.g. translate texts, fill order forms, approve requests.
- Owner, who is responsible for the execution of the process model.
- Developers, who implement the business processes and make them available for execution.
- Infrastructure, the computer systems which ensure the execution of process.
- Metrics, with which we can measure the performance of working with the process from different aspects.

Multidimensionality of Business Process Model A business process model is an entity with several aspects or dimensions, where each dimension represents a particular concern about the process, e.g.: how the activities should be organized; what are the data dependencies between the activities; which user or role is allowed to perform a specific task; what are the inputs/outputs of an activity etc. A particular model does not need to have all the dimensions but dimensions can be added to the model at any moment, and each dimension may change independently from others. For example if the organizational structure of an enterprise changes, the business processes do not need to change but their relation should be adjusted. Concrete executable business process models are difficult to manage due to the complex relation between the dimensions and frequent change of the business process, e.g. during the business process reengineering.

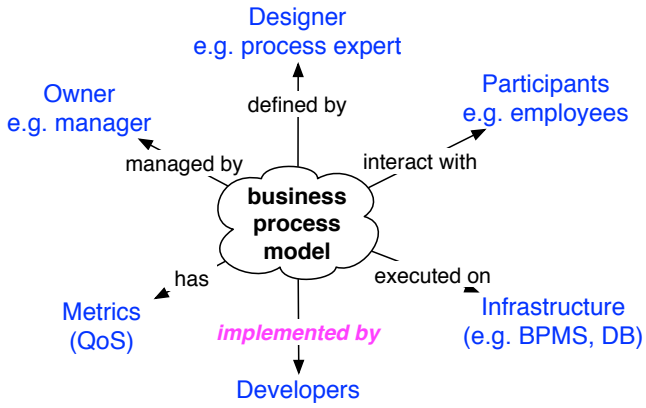


Figure 1.1: Elements Related to a Business Process Model

Figure 1.2 shows a business process for an online shop using a Statechart aliked notation. It has three main activities: *order*, *payment*, and *delivery*; each activity interacts with some participants which can be human, local or remote computer system. Each activity can be also a subprocess which specifies how the interactions with the participants should be organized. In this figure, only the abstract control flow, i.e. the dependency between the activities, is given. We can enrich this model by adding dimensions such as the data model for the process. Moreover, in different environments different data models may be used for the same business process.

Business Process Management System (BPMS) BPMS is the infrastructure which supports the execution and management of business processes. A BPMS can ensure the following requirements of business processes, regarding to their executions:

- Deploy and execute of business process models, e.g. creation of process instances, pass control from one activity to another activity.
- Authenticate of users and roles before performing the tasks.
- Manage and ensure the interactions between human and machines, e.g. generate graphical user interfaces from a business process model.
- Execute and manage local and distributed database transactions.
- Monitor the business process instances.

In general, a BPMS also provides tools to support the BPML for the modeling of business process. The tools will allow to create, manage, and verify the business process models at the design phase.

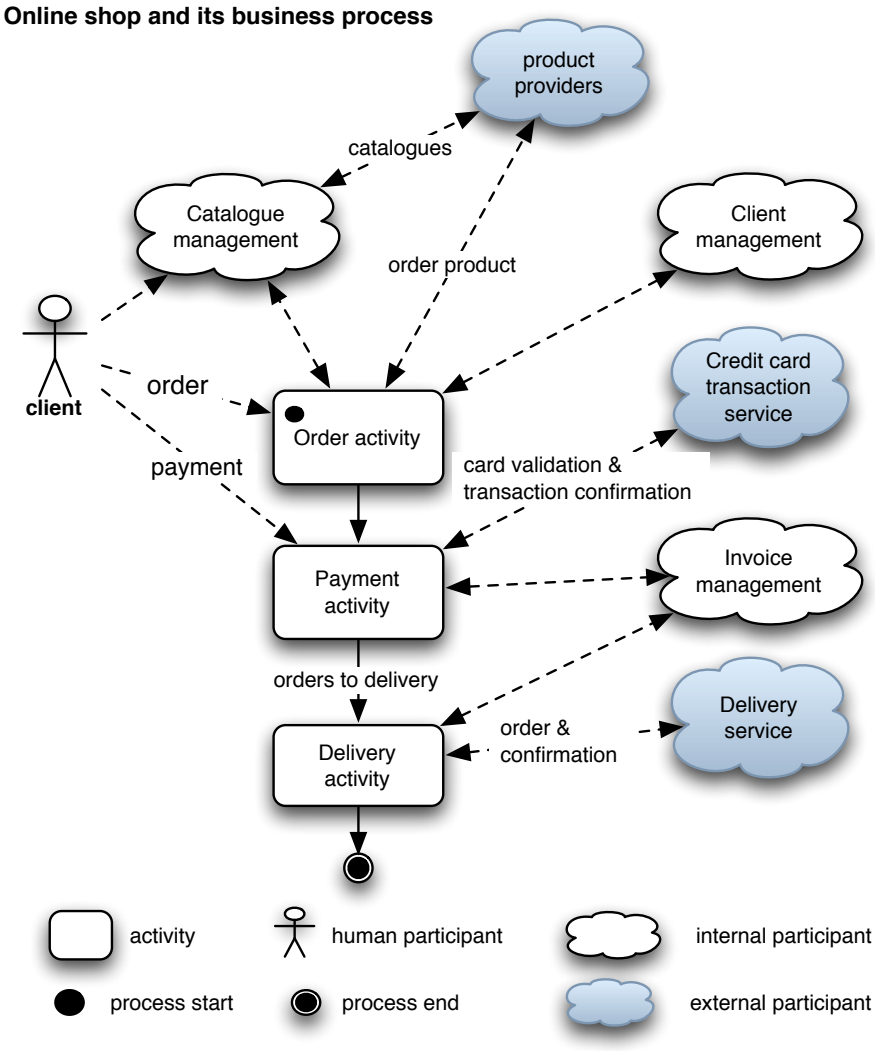


Figure 1.2: An Example of Business Process

In today's computing environment, business processes are long-lasting, parallel computer programs running in concurrent and distributed environments. Moreover, the business processes should be verified and tested thoroughly before being deployed, especially for critical systems. We are interested in the modeling, development, and management of business processes with modern software development methods and technologies, in order to continuously and rapidly satisfy the requirements from the domain of management.

1.2 Business Process Modeling and Development

The development of business processes reflects the cycles of general software development with some specificities. From software engineers' point of view, business process engineering should be itself an iterative process which consists of modeling, prototyping, verifying, and executing of business processes in an incremental, rigorous, systematic, and efficient way. It has the objective of rapidly adopting new business processes (new requirements) and the modification of existing business processes (e.g. change of business protocols). Due to the *complexity of models* and the *frequent change of requirements* in business domains, this objective is an open problem thus a challenge in software development.

1.2.1 Model-Driven Engineering

A software development process or life cycle is a structure imposed on the development of a software product. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

Computer scientists have been trying to find repeatable, predictable processes and methodologies that improve productivity and quality of software for decades. Some researchers try to systematize or formalize the tasks of developing software, others apply project management techniques to the development process. Software engineering processes are composed of many activities, notably the following:

- **Requirement analysis**. It consists of studying the domain knowledge and client's needs, then produce documents describing functional and nonfunctional requirements. It requires skill and experience in software engineering to recognize incomplete, ambiguous or contradictory requirements.
- **Specification**. Specification is the task of precisely describing the software to be written in a mathematically rigorous way, i.e. obtaining artifacts called

specifications. Requirements are considered as a kind of the artifacts. Specifications are most important for external interfaces that must remain stable.

- **Validation**. In practice, most specifications are written and used to understand and analyze the applications under development. However, safety-critical systems should be carefully specified and verified prior to application development. Validation is the task of checking the properties (e.g. correctness) of specifications before implementing them.
- **System design (architecture)**. The architecture of a software system refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements (of product or clients), as well as ensuring that future requirements can be addressed.
- **Implementation**. It is the step of developing the system in a concrete environment, i.e. realizing its functionalities according to the specification for a specific platform.
- **Testing**. Tests are used to validate the architecture and implementation of a system.
- **Maintenance and Improvement**. Maintaining and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. Maintenance and improvement may imply one or several development iterations and they represent an important part of software's life cycle.

Model-Driven Engineering (MDE) is a software development methodology which focuses on creating *models*, or abstractions, more close to some particular *domain concepts* rather than computing (or algorithmic) concepts. It means to increase productivity by maximizing compatibility between systems, simplifying the process of design, and promoting communication between individuals and teams working on the system. One objective of MDE is to apply model transformation and composition automatically via supporting tools. In this context, "Models" are rigorously defined specifications. Some activities or steps of development process can be automated or semi-automated.

For example, instead of manually writing code, tools can generate executable code from the specifications; on the other hand, automatically tests can be generated from specification and verify the codes written by programmers; validation of properties (model-checking) can be also applied automatically on the models. Figure 1.3 shows the activities and artifacts involved in MDE, activities such as test generation, testing, properties validation, and implementation are supposed to be fully automatized or automatized with human assistance.

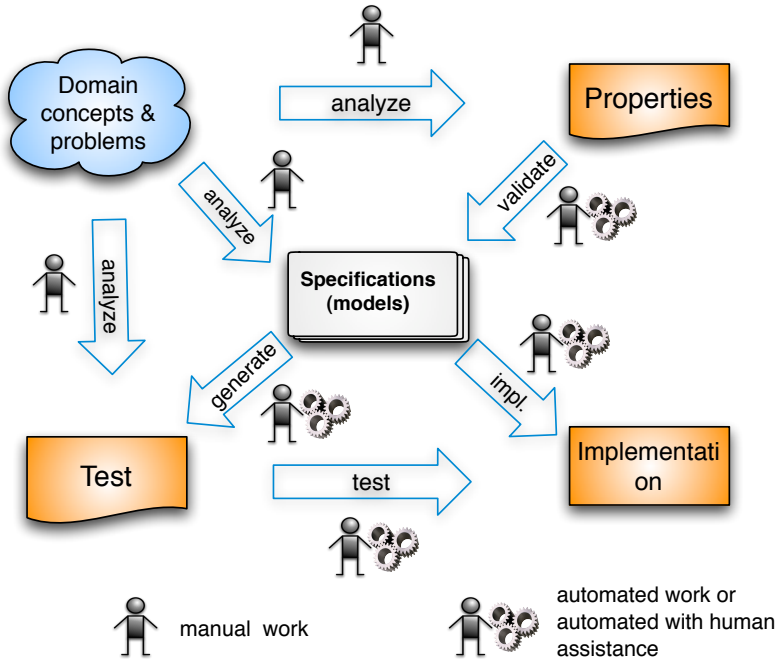


Figure 1.3: Activities and Artifacts in Model-Driven Engineering

1.2.2 Business Process and Service-Oriented Architecture (SOA)

The business process showed in figure 1.2 is an abstract and conceptual diagram, i.e. it does not give any details on how to realize these activities for a particular system. This abstraction is necessary because business process designers do not have knowledge about the environment the process will be executed in. In contrast, a software developer has to add more details and implement the business process for a particular system, i.e. *concretize the semantics* of the interactions and activities. At this phase, the semantical coherence has to be checked between the high-level model used by the designer and the implementation model created by the software developer.

Business Process as Service Composition The Service Oriented Architecture (SOA) is a distributed software model well-suited for business process. SOA comprises loosely coupled, highly interoperable application services. These services interoperate based on a formal definition independent of the underlying platform and programming language (e.g., WSDL [2]). The interface definition encapsulates (hides) the vendor and language-specific implementation. A SOA is independent

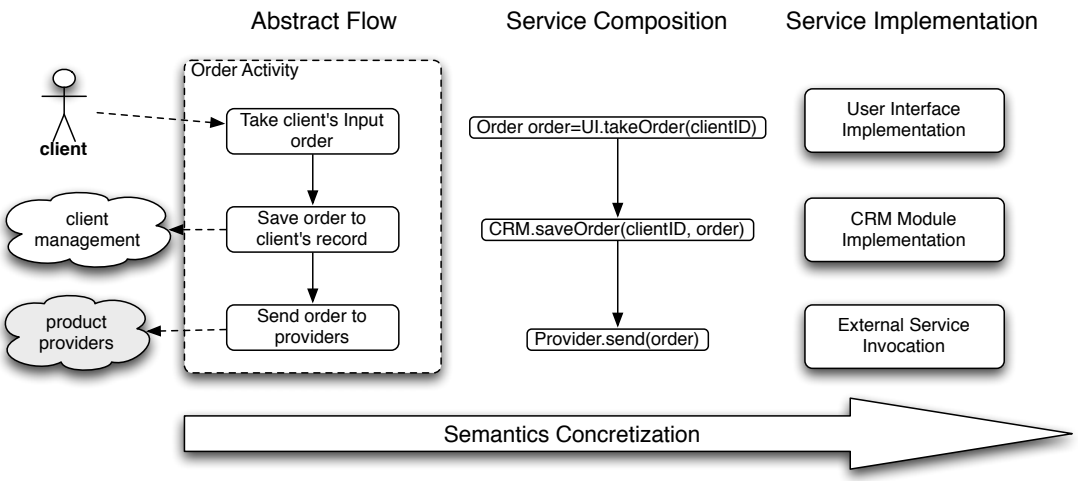


Figure 1.4: Example: Semantics Concretization of Business Process Models in SOA

of development technology and the software components become very reusable because the interface is defined in a standards-compliant manner. In SOA, business processes are often considered as service orchestration (intra-organizational) and service choreography (inter-organizational).

The SOA implies the separation of activities (services) of business processes and their implementation. The service composition introduces a layer based on the service layer to model business process. Based on the concept of SOA, we identify three levels of abstraction during the development of business process as shown in figure 1.4 for the example of online shop:

- Abstract and conceptual flow, e.g. a working plan of realization for an objective.
- Service composition. Decompose the process into semantically aggregated services (service composition), e.g. detailed plan specifying the tasks for each activity. Each service invocation corresponds to an activity.
- Services (related to data model) implementation, e.g. implementation of the services and their compositions using a domain-specific language (DSL), for example with XML and SQL. Data model can be also implemented at this level.

The services can be also implemented using a general programming language such as Java or C#. Most of the time this level is optional because DSL can be executable and supported by the environment. The corresponding architecture for business process as service composition is also given in figure 1.5.

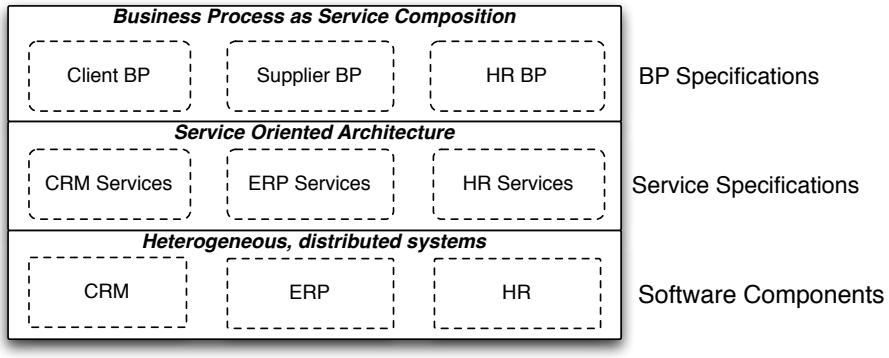


Figure 1.5: The 3-Level Architecture of Business Process as Service Composition

In this architecture, the specifications from different levels have different semantics and details about the business processes. From the software engineering's point of view, the goal is to obtain the traceability between the specifications of different levels and the coherence between the semantics of the specifications. Thus we need a methodological framework, i.e. the application of MDE for business process, to ensure the creation, transformation, and composition of models related to business process.

SOA is a standard way of understanding large information systems. Because services are implemented by software components or models, the composition of services is in fact the composition of models. The MDE for business process has to work with the underlying models and consider the semantics composition of the models.

1.3 Semantics of Business Process

1.3.1 Dimension of Business Process Models

A common convention of process modeling is of viewing business process from several dimensions or perspectives: control, data, operational, and resources. Each dimension describes a specific kind of events in workflow systems: control flow manages the states of process instances; data flow represents information retrieval and moves between the activities; operational flow represents interactions between the processes and other systems; resources usually represent available human or non-human resources for the processing of tasks. Sometimes resources also mean the objects used by the business processes during their execution. Among these perspectives, the *control-flow* is the most relevant one with which we distinguish business process

and database management; in most cases, the *data model* is indispensable to create executable business processes.

Control Flow and Data Model An executable business process has two essential aspects or dimensions: control flow and data model. Control flow can be abstract (e.g. based on activities) or concrete (e.g. based on actions or operations), it is the main concern which specifies what tasks should be done for a business process, the dependences between them, and the possible alternatives; data model is in general more concrete, it accumulates the results of control flow execution and influences the control flow, i.e. the data-dependent decision points in control flow.

The analysis of relation between the control flow and data model (flow) is a classical problem for the design of concurrent and parallel programs. Many work have been done in the domain of parallel computing, in order to minimize the data dependence insider the programs and maximize their parallel executability. However, the design of concurrent programs is still a challenge in parallel and distributed computing, regarding to the issues such as race conditions, deadlocks, and resource allocation problems.

Executable business processes are concurrent programs running in parallel and distributed environments. For the above mentioned reasons, many business process modeling approaches only consider the control flow and the data model is not explicitly managed, i.e. the impacts of data on the control flow are simply neglected during the modeling phase and handled in an ad-hoc manner during the implementation phase.

Transactions and Concurrency Transactions play an important role in the business world. Most of the time, the word *transaction* refers to transactions in database systems. Traditionally, transactions have a flat structure and follow the ACID properties. The DBMS should guarantee the atomicity (A), consistency (C), isolation (I), and durability (D) for each transaction through transaction management protocols and mechanisms, such as multi-phase commit protocols and concurrency control mechanisms. In general ACID transactions are supposed to be *short-lasting* *.

A business process may have simultaneously many instances, each instance shares resources with other instances. For example, in an online reservation system has limited number flight ticks and hotel rooms, the same flight ticket (or room)

*In fact, the notion of *short-lasting* is relative. For example, as far as the duration of resource locking is not an issue for the performance of the system, the ACID transactions implemented with locks can be considered as *short-lasting*.

should not be sold to more than one client. Similarly, several online transactions of the same client should be processed sequentially to avoid concurrent problems.

There are three characteristics for the transactions in business process: *duration*, *locality*, and *concurrency*:

- duration: short-lasting, long-lasting.
- locality: local transaction, distributed transaction.
- concurrency: without concurrency, with concurrency.

All combinations of these characteristics exist in business process modeling. In the context of business process management, transactions are executed in distributed, heterogeneous, multi-database environment. *Transactional business processes* are dynamic, interactive, and based on long-lasting transactions in contrast with classical ACID transactions. For example making flight and hotel reservation for a business trip is a typical transactional process that can last a long time (e.g. several weeks or months) and sometimes needs more than one session to be completed. It is therefore not possible to do the whole reservation procedure within one classical transaction. Long duration locking, which will degrade the performance of the system, should be avoided in this case.

For long-lasting transactions, the states of process instances are persisted in distributed systems and the ways of doing compensation are often depending on concrete situation and business conventions. Compensations or rollbacks in distributed systems are complex to realize and may be also long-lasting processes.

1.3.2 Semantic Engineering of Domain-Specific Models

A domain-specific language is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. Many DSLs have been used in software industry, such as: PostScript, SQL, Cascading Style Sheets (CSS), HTML etc. Domain-specific models are models defined using domain-specific languages.

During the modeling and development of business processes, we create different domain-specific models in different phases, such as: conceptual and graphical diagrams, control flow models, data models, executable models etc. Moreover, we perform the following activities related to the syntax and semantic of the models:

- *enrichment*. We add details into abstract models to develop concrete models. This activity occurs at the beginning of each iteration in iterative development.

- *composition*. Composition of models allows to develop complex models by using simple ones.
- *projection*. The reverse operation of composition, which allows to isolate the dimensions of complex problem as simpler models.
- *transformation*. Transform one kind of models into another kind of models and keep the semantic equivalence between the models. For example, generate executable code from UML models.

The *semantics engineering* of models aims to analyze the impacts of these activities on the semantics of the models, independently from the concrete syntax of the languages. For example:

- how the behavior of an abstract model can be enriched via the model composition,
- how model composition influences the state space of the models,
- what are the properties of each sub-components and the whole system, and their relations,
- how to prove the behaviors of two models are equivalent (e.g. before and after model transformation).

1.4 Motivation and Proposition

Our remarks on business process modeling and development are as follows:

- Ad-hoc, semi-formal approaches are difficult to be integrated into model-driven development, and they are less efficient and flexible for BP development. Using notations with formal semantics is necessary for model-driven engineering.
- The expressive power of a domain-specific modeling language should be enough to represent the domain's problems. For example, we consider *control flow, data model, and transactions are essential dimensions in business process modeling*. Adequate DSL should be used for modeling each dimension of business process.
- Many BP modeling approaches have difficulties to integrate different dimensions or perspectives of business process, e.g. control flow and data flow, during the development process without dramatical increasing the complexity of models.

- The nature of business process is about distributed computing, involving short and long-lasting transactional processes across heterogeneous systems. Concurrency should be explicitly managed by the business process models.

Executable business processes Based on the above remarks, we want to apply the principe of Separation of Concerns (SoC) [3] using formal notations, and apply semantical model composition for the modeling and development of business process. Moreover, we believe that the design of business process should have the same criteria as the design of concurrent, distributed programs. Based on this assumption, we propose an iterative, compositional approach for the development of concurrent, distributed, and transactional systems. *ID-net* is a formal notation we proposed as the semantics basis for model composition. It has the following characteristics:

- it has precise semantics of control flow and concurrency, it is easy to execute ID-net based model or transform it into executable codes for distributed environments.
- it provides a set of synchronization operators, allowing to specify the synchronization between ID-net and other kinds of models (i.e. *co-models of ID-net*). The synchronizations imply the composition of models from different problem dimensions, and they can be implemented by transactions.
- it uses identification as tokens, similar to object references and Uniform Resource Locator (URL), in order to minimize the coupling between different models.
- the system developed with ID-net, i.e. ID-net Controlled System (IDCS), is multi-dimensional. It is possible to extract and replace one dimension without changing others.

Methodologically, with ID-net our approach will allow the Separation of Concerns in business process modeling, facilitate iterative and incremental development, enable model transformation and code generation in development, and facilitate the model validation and verification of complex process-oriented systems.

Roadmap for the development of business process As shown in figure 1.6, our approach to develop business process is *incremental* and *iterative*: starting from conceptual workflow diagrams (which will be mapped to control flow represented by ID-nets), developers will enrich the control flow by synchronizing actions/events of control flow with actions/events from models of other dimensions, e.g. data models, organizational models, security models etc., resulting a multi-dimensional model.

According to our principle of semantical model composition, the multi-dimensional model should respect the constraints implied by each original model during its execution. Each dimension of the multi-dimensional model can be developed, verified, and tested separately, and it is possible to change one dimension without changing the whole system.

Overall, our approach of semantics engineering tries to answer the challenges we mentioned earlier in this section: facilitate business process modeling and engineering; facilitate the management of the complexity of business process models; rapidly adopt new requirements.

This thesis is presented as follows:

- Chapter 2 gives the state of the art of business process modeling and development in industry as well as in academic research, and explains the contribution and position of this work.
- Chapter 3 represents the first stage of this thesis, which can be read independently from others. We present our work for business process modeling and prototyping using Concurrent Object-Oriented Petri Net (CO-OPN) [4,5] with algebraic Abstract Data Types. We will show that the formal semantics and rich expressive power of CO-OPN allow to easily model workflow patterns and we can generate process prototypes from CO-OPN to simulate business processes and verify the properties of CO-OPN models. We present this work first because most concepts of CO-OPN such as the synchronizations, service-orientation, and encapsulation, are assembled and reused later in other chapters. Moreover, we show our principles of business process modeling during this work, which is important to understand our proposition.
- From Chapter 4 on, we start to present our proposition for the multi-dimensional compositional development of real-world business process. Chapter 3 presents a formal foundation for the semantical model composition using synchronizations. This formal foundation allows to specify the composition of heterogeneous models, i.e. different domain-specific models (DSMs).
- Chapter 5 gives the syntax and semantics of ID-net, which can represent the control flow of concurrent programs and business processes. We can synchronize ID-net with external models and obtain the ID-net controlled system.
- In Chapter 6, we present a framework to verify the behaviors of system controlled by ID-net. Because ID-net is a variant of Petri nets, the main idea is to reuse the model verification techniques of Petri net to facilitate the verification of complex system controlled by ID-net.

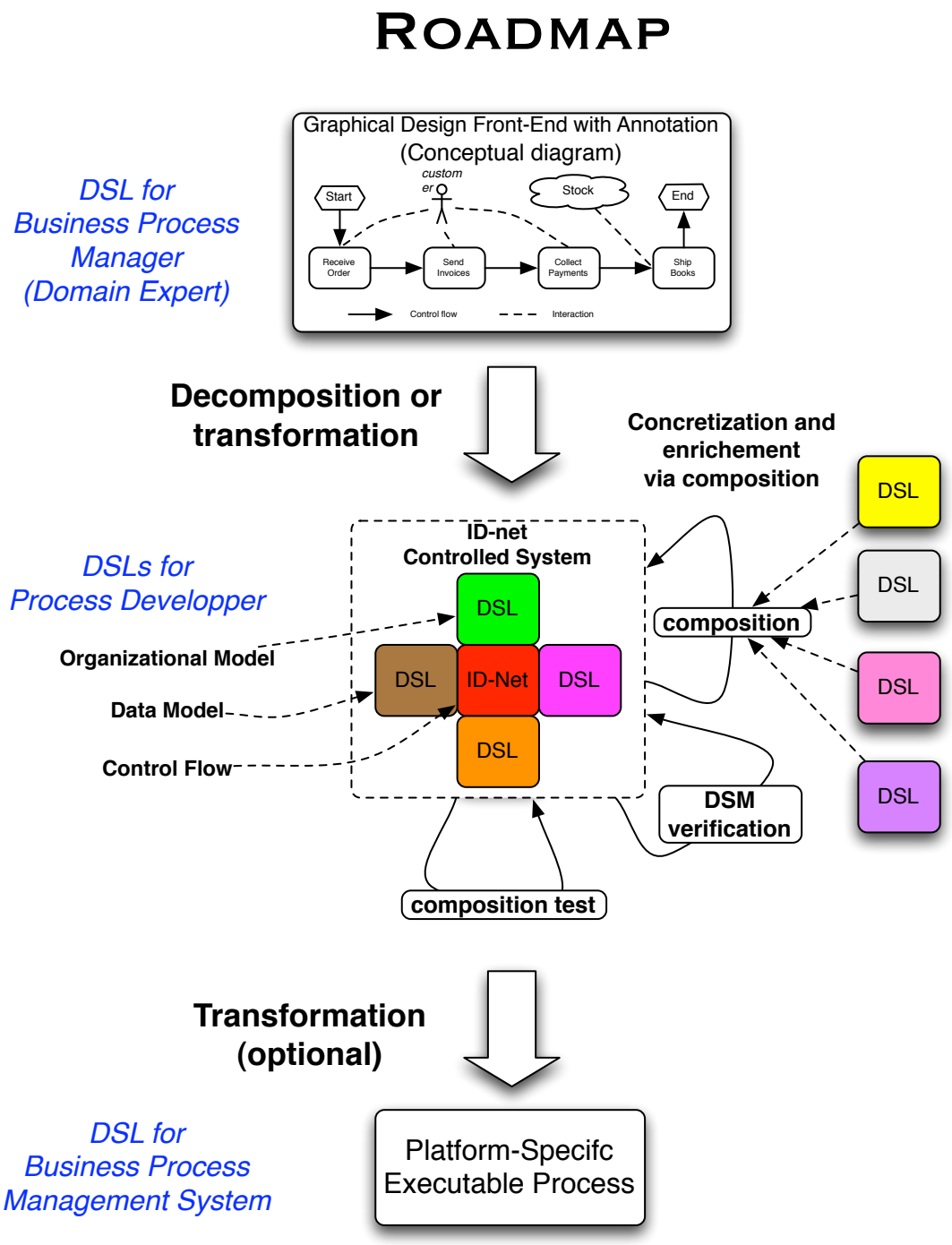


Figure 1.6: Roadmap of Our Approach for the Semantics Engineering of Business Processes

- In Chapter 7, we show the applications of using ID-net to develop concurrent programs and business processes. In fact, for different kinds of problems, ID-nets can be annotated by different languages which use different kinds of tokens. The annotated ID-net is a composition of models and can be transformed into executable code without difficulties. In addition, ID-net models and specific data models can be composed and encapsulated together as *Service Component*, which can be used as the basic building blocks to model and develop domain specific models.

- Chapter 8 presents the applications and case studies of our approach in two different granularities of business process modeling: using BPMN and modeling Web flow in Web applications. The second case is based on our work during the project TestIndus. In collaboration with CLIO, a software consulting company in Geneva, we worked on an online insurance subscription application and develop methods and tools for the software quality assurance. Based on our models, we automatically generate tests cases (Selenium scripts) to test Web-based business processes implemented by software developers.

- Chapter 9 concludes this work and discusses some possible directions in the future.

Chapter 2

State of the Art

2.1 Modeling Languages and Notations

Business process modeling language or *workflow language* is a domain specific language which describes business processes in a human and/or machine understandable way. It is essentially used for the purpose of communication between business process designers and software developers, in order to discuss and define the behaviors of business processes precisely. Many business process modeling languages have been proposed in recent years with different paradigms and concepts, for examples: database-centric languages for the manipulation of database tables, graphical-based languages to facilitate the design of business processes, and service-oriented languages for the easy integration in service-oriented architectures.

Currently, the most used business process modeling languages are the follows (which are also conceptually significant):

- Flowcharts
- Event-Driven Process Chains (EPCs)
- UML activity diagrams and Statecharts
- XML Process Definition Language (XPDL)
- Web Service Business Process Execution Language (WS-BPEL or BPEL)
- Business Process Modeling Notation (BPMN)

Flowcharts is a well-known graphical notation to express basic flows and concepts such as activities (actions and operations), input/output, branchings, and

conditions. They are used more for communication purpose than as rigorous descriptions. There are many variances and extensions of Flowcharts which propose different building blocks and semantics, e.g. program diagrams, control flow diagrams and data flow diagrams.

Event-Driven Process Chains (EPC) are a widely used technique for modeling business processes. Although it is graphical-based and informal by origin, peoples are trying to formalize [6] and define precise semantics for it [7, 8]. Standardization efforts within the EPC research community have led to the definition of an XML-based interchange format for EPCs called EPC Markup Language (EPML) [9]. EPML aims to serve as a tool-neutral interchange and intermediary format for EPC business process models.

UML provides activity, state, object and class diagrams to capture important business processes and artifacts. More detailed BPM models can easily be built using UML Profiles. Actually, there is no official UML profile for business process modeling. [10] presents a work of using UML activity diagram as a workflow specification language.

Workflow Management Coalition (WfMC) [11] is trying to define standards to uniform the terminologies in the domain of business process and workflow models. Their propositions principally include XML Process Definition Language (XPDL) and reference model for workflow management systems (WfMS). XPDL is more concentrating on tools interoperability (at the syntax and graphical level) than on the modeling power of the language.

Build on IBM's WSFL (Web Service Flow Language) and Microsoft's XLANG (Web Services for Business Process Design), WS-BPEL [12] (Web Service Business Process Execution Language) provides a means to formally specify business processes and interaction protocols. It combines the features of block structured process language (XLANG) with those of a graph-based process language (WSFL). By now, BPEL has become the de facto standard in the Web services composition arena.

BPMN [13] is a graphical notation for modeling business processes proposed by BPMI (Business Process Management Initiative). Adopted as OMG standard in 2006, it is supposed to be a standard in complement with BPDM (Business Process Definition Metamodel) *, which is another OMG standard related to business process. However, BPMN is not a UML specification. BPMN 2.0 is in the phase of Request for Proposal which solicits submissions that reconcile the Business Process Modeling Notation (BPMN) and the Business Process Definition Metamodel (BPDM) to specify a single language with metamodel, graphical notation and in-

*OMG merged with BPMI in June of 2005.

terchange format. BPMN is often used for frond-end graphical language for process design and it can be transformed into BPEL for execution [14].

2.1.1 Workflow Patterns

The workflow patterns initiative [15] aims to establish a structured approach to the issues of the specification of control flow dependencies in workflow languages by identifying their common patterns.

The objective was extended to *data perspective* and *resources perspective* in [16] and [17]. These patterns have been used as a benchmark for comparing process definition languages and evaluating their relative expressive power. The evaluation has been applied to UML [18], EPCs [19], Pi-calculus [20], BPEL [21], XPDL [22], YAWL (Yet Another Workflow Language) [23], Renew [24], and BPMN [25].

Nevertheless, workflow patterns are described informally and the set of identified patterns grows as the research and development on BPM progress. For example, initially over 20 control flow patterns have been identified, afterwards they have been extended to 43 [26]. It is easy to identify a particular pattern, but systematically and continuously taking into account new workflow patterns by using a single formal formalism seems difficult. For example classical Petri net and colored Petri nets can express most control flow patterns, but it can not express complex patterns such as XOR-JOIN and structured patterns, due to its local semantics.

YAWL is a workflow language oriented to workflow patterns: its building blocks are based on the identified control flow patterns and the authors defined formal semantics for all the building blocks, i.e. via a top-down approach. Similar to BPMN, YAWL has high-level, easy-to-understand building blocks which ease the modeling work. Nevertheless, YAWL's semantic is self-contained and its integration with other aspects such as data model is unclear.

2.1.2 Semantics of Business Process Modeling Languages

Many business process modeling languages were developed with the purpose of conceptual analysis and communication, and they lack formal foundations (e.g. for the control flow of business process) allowing transformation and verification of models. A typical example could be the semantics of OR-JOIN is unclear or ambiguous in EPC [7, 8, 27], UML activity diagram [18], and XPDL [22]. In addition, we should mention that only BPEL and BPMN support transactions in their syntax, and none of the languages consider the concurrency problem of business processes.

In order to give the modeling languages formal and precise semantics to support

model verification and model-driven development, researchers try to transform them into formal notations. For example, researchers try to transform BPEL into into formal notations such as finite state machines (FSM) [28–30], process algebra [31,32], Pi-calculus [20], abstract state machines [33], or Petri nets [34, 35] to verify the properties of business process such as cyclic dependencies, deadlocks, and soundness. Many business process execution engines are also based on these formal notations.

Workflow patterns and semantics of business process modeling language Workflow patterns represent the most common use-cases i.e. the requirements (the expressive power) of business process modeling modeling languages. Thus, a business process modeling language should have rich expressive power and a formal foundation for its semantics. The balances between the expressive power, formal semantics, and user-friendly (e.g. graphical representation) of the modeling language are not always easy to maintain.

Summary In table 2.1, we compare the above mentioned business process modeling languages regarding to the criteria we have identified in Section 1.3: *control flow modeling, data modeling, the supports of transaction, formal semantics, and their design objectives*. We expect to have a modeling language which has good supports of these aspects and can facilitate the model-driven engineering of process-oriented systems.

2.2 Related Work for the Semantics of Business Process

2.2.1 Formal Notations

In the development of large software systems often the problem of managing the complexity of the task arises. Formal methods have been suggested as a means of facilitating the design of complex systems and ensuring their correctness. The formalism differ in the underlying paradigms, in the abstraction level and their view of a system. As a result, for every specification methods a specific theory has been developed.

Process-centric vs. Concurrent system In business process modeling, there has been debates for the modeling of business process between the communities of process algebra and Petri nets [36–38]. From our point of view, process algebra has

Table 2.1: Comparison of Business Process Modeling Languages

BPML	**Control flow**	**Data model**	**Transaction**	**Semantics**	**Objective**	**Executable**	**Remark**
Flowchart	simple	no	no	informal	concept representation	no	widely used in many domains
UML Activity Diagram	simple	possible	no	state-machine based	behaviors modeling in UML	no, possible with some extensions	part of UML standard
XPDL	simple	possible	no	unclear	process definition interchange	no	supported by WfMC
EPC	simple	no	no	some formalization work exist	graphical modeling language	no, possible with some extensions	has vendor supports
BPMN	has complex gateways, but not all of them are formalized	use messages	syntax support with exception handling	partially formalized	graphical modeling language	no, possible with some extensions	OMG standard, widely adopted
BPEL	complex: links and structured flow	use messages	via WS-transaction	clear, operational	execution, web service composition	yes	standard in SOA
YAWL	most workflow patterns	no	yes	formal, operational	modeling, verification, execution of business process models	yes	formal semantics is limited to control flow
Expected Language	*has no limit to express the patterns, extensible*	*yes, integrate any kind of data models*	*yes*	*formal, operational*	*modeling, validation, execution of business process models*	*yes*	*MDE approach*

the advantages to express a particular business process's behaviors and compare them with others, but the problem of concurrence between the processes is not addressed; Concurrency is an essential concept in Petri nets, and it is easy to express a *system with many business processes and their interactions*, the behaviors of each business process may be non-deterministic due to the concurrence (i.e. on the shared resources between the business processes).

Petri nets is the most important business process modeling formalism because of its graphical representation, formal semantics, and abundance of analysis techniques [39]. In addition, its has explicit state representation and the analysis techniques are application-independent. Classical Petri nets are not modular and limited by its indistinguishable tokens, thus most Petri nets based approaches use high-level Petri nets with modularity support to model business process. Renew [24] uses nested nets as tokens and passes net references between places to realize the mobility of objects. YAWL [23] uses P/T nets as its theoretical base for process verification. There are also several propositions based on Colored Petri Nets (CPN) [40–43].

Model-Checking and Verification With the increased power of computation, in the formal method community, people are able to check and verify the properties of complex computer system in a reasonable time. For instance, model checking techniques have been widely used in the development of microprocessors and mission-critical embedded systems. Some common verification techniques and tools are: SMV [44], SPIN [45], SAT [46], Binary Decision Diagrams (BDD) [47], Data Decision Diagrams (DDD) [48,49] etc. In this case, a modeling language with formal foundation presents important advantages for the management of complex systems.

The state spaces of software components (models) are in general more complex and dynamic than hardware and embedded systems, principally due to the volume of data and the relation between the data. Moreover, depending on the properties we want to verify, different strategies have to be adopted for the model-checking of software models. For example, we may want to check the following properties of business processes in a complex system:

- eventually a business process instance will terminate, i.e. no deadlock and live lock.
- a business process instance will terminate in a reasonable steps and times, i.e. the performance of the business process.
- a business process has exclusive usage of some resources during its execution. i.e. isolation of resource.

If we apply the principe of the Separation of Concerns during the modeling phase, it is possible to reduce the complexity of the models and perform classical model-checking and verification techniques on each concern of the models. Each concern can be obtained by applying projection on the complete model.

2.2.2 Researches in Petri nets Community

Designed as a variant of Petri nets to facilitate model-driven software development, ID-net can profit from the principles of Petri net modeling and the abundant model analysis techniques for classical Petri nets developed in the last decades. In this subsection, we will discuss some relevant work (we have inspired from or using similar concepts) related to the design principles of ID-net, i.e.: multi-dimensionality, tokens types, modularity, data flow modeling, transaction, and concurrent execution of models.

Multi-dimensional Petri net model Petri nets has become mature modeling and analysis tools for complex systems for decades, especially for the aspects of *non-sequential process* and *concurrency*. Many researches have been investigated and proposed extensions for the classical Petri net. Examples of such extensions include "time", "color", and "probability".

The *multi-dimensional Petri net model* tries to deal with the extensions of Petri nets in a unifying way, it has been studied by [50] and [51]. Multi-dimensional Petri nets can be analyzed using traditional techniques. By analyzing the projected multi-dimensional Petri net, we can deduce the properties of the original multi-dimensional Petri net. Most Petri net extensions can be considered as improving or adding dimensions to classical Petri nets.

The token dimension Many extensions of Petri nets are related to the token dimension: colors, algebraic data types (e.g. bags, sets, nested data), tokens as nets, tokens as (object) references etc.

Colored Petri Nets (CPN) is one of the most used Petri nets for modeling and model checking, for the rich expressive power of color tokens and its tools supports (CPN Tools). CPN allows to use a predefined set of primitive data types and create complex data types based on these primitive types.

Algebraic Petri Nets (APN) uses algebraic data types (ADT), ADT tokens are algebraic terms. The axioms give the rules of how to build the terms representing the values. Similar to the colors in CPN, it is possible to define complex data types.

In CPN and APN, tokens are self-contained and immutable. The only inconvenient of using this kind of tokens is the mix of control flow and data model (and data flow). The state space of models increase rapidly in development, it is impossible to apply the classical model-checking techniques and difficult to reduce the complexity of models.

In Object Petri nets (OPN) [52, 53], tokens can be of elementary type (e.g. colors) and subnets, subnet tokens are lying in places and are moved by transitions. In contrast to ordinary tokens, however, they may change their state (i.e. their marking) when lying in a place or when being moved by a transition. Transitions in subnets can synchronize with the first-level transitions.

Reference Nets [54] allows to use object reference as tokens. Similar to identification tokens used in ID-net, *object references* can point to any kind of resources. *Guards and operations* related to the resources can be annotated on the transitions, where the operations imply *synchronous channels* communicating with the referred resources. Renew [55] is a tool based on Java for the development of executable Reference Nets models.

Tokens as names: the *v-net* [56] uses *names* as tokens and variable inscriptions on the arcs of Petri nets. Similar to *identifications and references*, *names* can refer to any kind of resources. Their approach concentrates on the dynamic generation and duplication of names and the analysis of their impacts on the properties of the system modeled by v-net. In ID-net, the name creation and duplication can be specified by using the conditional *generator transitions* and *pipe transitions*, respectively. Their analysis techniques can be applied with ID-net directly. The notion of *fresh token* in ID-net is similar to the *name creation* in *v-net*.

Data flow modeling in Petri nets One main issue in data flow modeling with Petri nets is the *flexibility*, because different data modeling notations have different application domains. For example: concurrent programs use mostly the primitive data types and structured data types (e.g. records); data processing applications use mainly the sets of structured (and nested) data; database-centric workflow use the relational data model. Data modeling is domain-dependent, and the underlying notations may have limitation in terms of expressive power.

Due to the simplicity and the rich expressiveness of relational model [57] for information systems, several researchers have tried to bridge the relational data model and Petri nets [58, 59] to model data flow. Andreas Oberweis and Peter Sander [59] use the Nested-Relation/Transition Nets (NR/T-nets) to handle the relational data and its transitions are annotated by the conditions and operations on the nested data tokens. Similarly, Jan Hidders et al. [58] proposes a dataflow language DFL based on Petri nets and nested relational calculus (NRC), where

tokens represent complex static data values (nested relational data).

Modularity: encapsulation, hierarchy The encapsulation and hierarchy are ways of (de)composing and organizing the Petri nets modules. They allow to create reusable components. CPN uses the notion of *page* to organize the subnets. The "nets within nets" paradigm, which is applied by Object Petri nets and Reference nets, can be also considered as an encapsulation mechanism.

Transaction and Communication Transaction and communication are ways of synchronizing different modules together to make larger system. In general both of them imply the atomic execution of actions and allow message passing.

Few Petri nets support transaction in their semantics. Zero-safe net [60–62] supports the synchronization of transitions, it uses zero-safe places to synchronize the transitions and create transactions. CO-OPN supports the semantics of transaction via its synchronization operators.

[63] extends Colored Petri Nets with channels for synchronous communication by inscribing the channel operators (?!, !?) on CPN transitions. A step is only enabled when each channel receives the same number of access via ?! and !? and if the arguments values of the accesses match, too. Each transition has at most one channel inscription of each type. The channels allow to synchronize CPN transitions with external models which use the channel (though the proposed analysis technique does not support this).

Reference Nets uses the notion of *synchronous channel* while a transition is synchronized with (or call) the operations of the token objects. It is possible to use conditions (guards) in the synchronization and pass values between Reference Nets and the synchronized objects.

Concurrency semantics, execution and scheduling of Petri nets models The parallel executability of of Petri net models is an important concern for us. Models use interleaving semantics imply arbitrary orders for the execution of simultaneously enabled transitions and they have limitation in distributed environment. Specification languages based on true concurrency [64, 65] are more informative. True concurrency allows to decide the order of execution at the implementation phase or at runtime, i.e. parallel execution will occur if enough resources are available, otherwise sequential execution will occur.

Petri nets provides a powerful formalism to model various classes of discrete event systems. They can be used for qualitative and simulation purposes because they provide a better understanding of the dynamics of the system. [66] has studied

the parallel and distributed simulation of free-choice Petri nets. There are also people using Petri net for the design and validation of circuits [67], where the execution of final models is in highly parallel.

Separation of Concerns Even in some concrete cases they are similar or equivalent, Separation of Concern is more general than modularity. With Separation of Concerns, it is easy to isolate an aspect of a complex system (e.g. in terms of state space), modify and verify the aspect independently from other parts of the system. In the world of general programming languages (GPL), Aspect-Oriented Programming (AOP) is a technique of applying the principe of Separation of Concerns.

In the Petri net community, to our knowledge currently there is no approaches which explicitly applies the principe of Separation of Concerns from the modeling to the verification of models. Even some of them (such as Reference Nets) can syntactically support the principe, their designs have different objectives and the semantical analysis between the concerns is missing. A typical example could be the separation of control flow model and data model in business processes, where each kind of model has its own state space and the whole system is a composition of the models and state spaces. At this point, we think our proposition ID-net and the framework around ID-net have unique value proposition for the modeling and development of complex systems.

Summary In this subsection, we discussed some related work and concepts in the Petri net community, which we identified as relevant for the semantics supports of business process modeling language, i.e.:

- the ability of using different kinds of tokens in Petri net, which means expressive power in data modeling.
- the ability of expressing data flow in Petri net.
- the support of modularity, which allows to build large systems.
- the support of transaction semantics, which is important in business process modeling.
- the extension mechanisms, which allows to integrate with other kind of models (of other problem dimensions) easily.
- the support of Separation of Concerns: if we can obtain different concerns of the system, make modifications, verify and update the models representing the concerns easily.

In table 2.2, we summarize the comparison of main Petri net variants mentioned in this section. We expect to have a formal, Petri net-based notation which supports these requirements as much as possible — it is the main reason why we propose ID-net and the framework around ID-net.

2.3 Conclusion

In this Chapter, we give an introduction about the principal business process modeling languages by comparing their design objective and expressive power. The comparison is by no means exhaustive, and we want to show the limitations of the existing modeling languages : flexible and formal semantics, the support of transaction, and the support of model-driven engineering. In addition, we evaluated some existing Petri net-based notations to see if they can support the semantics requirements we identified for business process modeling. We are expecting to have a flexible modeling language with (relatively) rich expressive power and formal foundation, with which we can build large and reliable complex systems via model-driven engineering.

The *semantics engineering* of models consists of analyzing the *enrichment, composition, projection, and transformation* of models at the semantics level independently from the concrete syntax of the languages. It allows better understanding of the behaviors of complex models and building them correctly. Among these activities, **semantics composition** is the essential mechanism for the realization of these activities, and it is the foundation of our approach.

There are some researches working on the semantics composition and enrichment of process models, such as *Action Refinement* in Process Algebras [68, 69]. The operation of action refinement is meant to support the hierarchical construction of concurrent systems. The idea is to start the construction with an abstract model, through a rigorous action refinement process, we obtain a detailed model with equivalent but enriched semantics. In hierarchical system design, refinement allows to incrementally increase the level of detail in the system description. This approach has limitations to integrate heterogeneous models other than the process algebra models.

The semantics composition of models can be specified by CO-OPN synchronizations. In fact CO-OPN contexts use this mechanism to synchronize CO-OPN modules, and we can deduce and analyze the behaviors of the synchronized models. In Chapter 3, we use CO-OPN to model business processes and synchronize them with other kinds of models. However, the algebraic data types are not practical in real-world business process modeling.

Table 2.2: Comparison of Semantics Supports of Different Petri net Variants

Type	**Tokens**	**Data Flow**	**Modularity**	**Transaction Semantics**	**Extension mechanism**	**Separation of Concerns**
P/T nets	indistinguishable	no	no	no	no	no
Colored PN	colored tokens	explicit	pages	no	via colors	no
Object PN	colored tokens, net objects	implicit or explicit	object	no	via colors	no
APN	defined by algebraic data types (ADT)	explicit	no	no	via ADT	no
Zero-Safe net	indistinguishable	no	no	yes	via synchronization	no
Reference Net	any object references	implicit	object	no	via objects	no
NR/T-nets	nested relational data	explicit	no	no	nested relational data	no
DFL	nested relational data	explicit	no	no	nested relational data	no
CO-OPN	defined by ADT, object references	implicit	object, service	3 operators	synchronization, ADT	no
Expected Formal Notation	*any kinds of tokens*	*implicit or explicit*	*yes*	yes	yes	yes

In addition, there are many work about the syntax composition and integration of concrete domain-specific models or UML models without analyzing the influence on their semantics, i.e. "mush-up" different kinds of models. From software engineering's point of view, it should be avoided for the development of reliable and critical systems.

Semantics Composition in Business Process In the domain of business process modeling and development, we have to consider the problems from different points of and dimensions, for example:

- from a business process' point of view: the objective of this process; what are necessary steps and conditions to achieve the objective, e.g. activities, control flow, information (data) flow.

- from a system's point of view: when many (different) business processes are running in the same system, what are the impacts and interaction between them, e.g. concurrence access of shared resources, priorities of the tasks of different process instances. These aspects will influence the state of business process instances.

- from a user's point of view: a user may interact with one or several activities of a business process during its execution, the user's role and responsibilities are defined by the business process, e.g. filling forms, taking decisions.

Each problem dimension may be modeled by a domain-specific language with a specific design paradigm. During the modeling and development of business process, we have to compose and integrate (or refine) the domain-specific models correctly with respect to their semantics.

Our work on the composition of semantics is based on the *synchronized composition* of Labeled Transition Systems (LTS) in Arnold's book [70]. LTS is a simple, intuitive and flexible tool for defining the semantics of concurrent programs, and as shown by Plotkin [71], they can be used to define an operation semantics for a wide range of programming languages by means of structural rules. Our semantics composition approach can be applied to all the DSLs whose semantics can be given by LTS.

Basically, our approach supposes that the synchronized composition will imply constraints in the initial models, which are independent before the composition. A composition consists of a set of synchronizations between the LTS of the models to be composed; each synchronization will imply collaborations between the models

and reduce their own (free) behaviors; we can compute the semantics of the composed models by using the semantics of the initial models and the synchronization specification. The semantics of the synchronization is given in Chapter 5.

Chapter 3

Service-Oriented Business Process Modeling and Prototyping with CO-OPN

In a system of Service-Oriented Architecture (SOA), e.g. a process execution systems using BPEL4WS [12], activities are carried out by internal or external services. In SOA, business processes are an aggregation of services and some local functionalities, for example the online shop shown in figure 1.2 aggregates some catalogue services of product manufacturers, one or several delivery services and an online credit card transaction service. The whole process uses the information system to persist the transactions and manage data. Furthermore, interaction between business processes and their participants are accomplished via the services.

In this Chapter, we will present a business process modeling and prototyping approach with CO-OPN [4, 5], a formal specification language based on algebraic Petri nets, transaction, and object-orientation. Through this chapter, we will analyze the advantages and weakness of using high-level Petri nets to develop business process, and identify the requirements for the expressiveness of a language: the modeling of control flow, data, concurrency, transactions, and long-lasting transactions.

3.1 Concurrent Object-Oriented Petri Net

CO-OPN (Concurrent Object-Oriented Petri Net) is an object-oriented formal specification language based on synchronized algebraic Petri nets. This language allows the definition of reactive concurrent objects and includes facilities for sub-typing, sub-classing, and genericity. Basically, CO-OPN has three types of modules: *ADT*

(algebraic Abstract Data Type), class, and context: ADT represents data, class is an encapsulation of algebraic Petri nets, and context is a higher level of encapsulation which defines the contextual coordination between components. A component can be a context or an object; an object is an instance of a class.

3.1.1 ADT, Class and Context

CO-OPN ADT modules define data types by means of algebraic specifications. Each module describes one or more sorts, along with generators and operations on these sorts. The properties of the operations are given in the section *body* of the modules, by means of positive conditional equational axioms.

CO-OPN classes are described by modular algebraic Petri nets with particular parameterized external transitions which are defined by so-called *behavioral axioms*, similar to the axioms in an ADT. Class instance are objects; they possess methods and gates allowing specifier to model components of a system. A method call is achieved by synchronizing external transitions, according to the fusion of transitions technique. The axioms have the following structure:

Cond => event **With** synchro :: pre -> post

In which the terms have the following meaning:

- *Cond* is a set of equational conditions, similar to a guard;
- *event* is the name of a method or transition with algebraic term parameters;
- *synchro* is the synchronization expression defining the policy of transactional interaction of this event with other events, the dot notation is used to express events of specific objects. Synchronization operators are: sequence **(..)**, simultaneous **(//)**, and non-determinist exclusive choice ($\oplus$);
- *pre* and *post* are usual Petri net flow relations determining what is consumed and what is produced in the object state places.

CO-OPN contexts are units of computation where included entities are coordinated following the same synchronization rules used for the class methods and gates. Contexts can be organized hierarchically and objects can move between them.

The state of an object is represented by tokens in its places; tokens can be ADT terms or objects. Passing objects between two places is implemented by passing their references. Furthermore, the state of a context depends on all state of objects in the context; ADTs are immutable entities without state.

Syntactically, each module has the same overall structure, it includes an *interface* section defining all accessible elements from the outside, and a *body* section including the local aspects private to the module. Moreover, class and context modules have convenient graphical representation, which are used in this work.

The features dealing with object-orientation such as sub-classing, sub-typing and genericity can be found in [4, 5].

3.1.2 Service Components

We use adopt CO-OPN's graphical notation to describe the *service components* and their composition. In CO-OPN class and context, *methods* and *gates* describe *provided services* and *required services* of modules, respectively. Each method or gate has a signature that indicates the types of its parameters, and the value of a parameter can be an algebraic ADT term or an object.

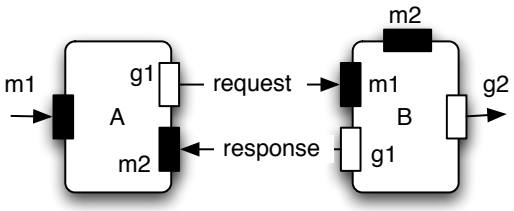


Figure 3.1: Representing Services in CO-OPN

Figure 3.1 shows two CO-OPN components **A** and **B**, where **A** uses a service provided by **B**. Methods are depicted by black boxes and gates are depicted by white boxes on the edges. The *request* and the *response* can be synchronized or not. If they are synchronized, *m2* of **A** can be considered as the "callback" method of the *request*. Depending on the state of objects, its methods can be firable or not; if a method is firable, fire it or not depends on the external environment of the object, e.g. context containing the objects; if a method is not firable, any transaction with tries to synchronize with this method will fail.

Before explaining the expressive power of CO-OPN for service representations, we would like to refer the transmission primitives used by Web Service Description Language (WSDL [2]) to abstractly represent services. Full support of these transmission primitives makes possible the rapid integration of components into a service-oriented environment.

WSDL has four transmission primitives that an endpoint can support [2]. The *endpoints* can be considered as CO-OPN components:

- One-way. The endpoint receives a message.
- Request-response. The endpoint receives a message, and sends a correlated message.
- Solicit-response. The endpoint sends a message, and receives a correlated message.
- Notification. The endpoint sends a message.

As web services, CO-OPN services are stateless, which means there are no sessions between different invocations or the session should be managed explicitly. In Figure 3.1, *A.m1* and *B.m2* represent a 'One-way' transmission, *B.g2* is a 'Notification', the set of *A.g1*, *B.m1*, *B.g1* and *A.m2* is a 'Solicit-response' for *A*, and at the same time, it is a 'Request-response' for *B*.

A service component described by this notation can be a process, a subprocess, an activity, a logical connector, or an abstraction of functional module inside or outside of the system. Business processes are built via the composition of these service components and the business processes themselves interact with partners via services.

3.2 Modeling Business Process

3.2.1 CO-OPN Building Blocks and Patterns for Business Process

The basic element of CO-OPN building blocks for business process is *Activity*. Activities use the set of values of type *Case* to pass control from one to another. This uniform handling of values is not ideal for methodological reasons but we will keep it for simplicity. Generic types can be defined instead and instantiated to compatible types when necessary in order to allow specificity of each building block. The ways of passing control flow between activities are summarized by some *Workflow Patterns* taken from the well-known list of W. van der Aalst [72]. A *Process Definition* consists of activities and rules used to pass control and other flows between them. In CO-OPN, *Case* is modeled by ADT, *Activities* are defined using *Class* and *Process* are defined using *Context*, hence the axioms of *Context* are rules for passing the control flow. Note that a process can be abstracted to an activity, hence processes can be composited hierarchically.

- *Case:* In CO-OPN, *Case* is the super type* of all types used to represent a process instance in activities. We define *Case* as an ADT of pair <*ID, LocalState*>, where *ID* is the global identification of process instance and *LocalState* is the local instance value representation. For example, in an insurance company, the "client claim ID" can be a global identification of the process which handles clients' insurance claims. At each step (activity) of the process, a case may have different *states* such as "approved, not approved" and may be represented by different documents such as "claim form" and "evidence". In this example, the "client claim ID" is the global identification of case in the process; "approved claim", "rejected claim", "claim form" and "evidence" are local states used by specific activities. Note that a global identification in one process does not need to be the global identification of another process. Figure 3.2 shows an example of activity with 4 LocalState values. It is possible to define algebra on these ADT values to obtain "chemical reaction" on tokens (fusion, merge...) by means of specific functions; however, this subject will not be investigated in this work.

- *Activity:* An activity accepts *cases* values and performs actions or treatments related to the cases. It can invoke external services and accept responses from external systems or users. An activity *sends out* the cases when it finishes its actions, note that an outgoing case can be any value of type *Case*, even with a different identification(workflow pattern: multiple instances). The invocation of external services can be considered as data perspective or operational perspective, which will not be discussed here. The destination of outgoing case is determined by the process. *Activity* is the equivalent of *Task* in some workflow languages, however, we distinguish *Activity* from *Task* by considering *Activity* as static definition and *Task* as runtime "workitem" w.r.t. a case instance.

- *Process:* Process specifies collaboration between activities. Defined by means of CO-OPN context, a process instantiates activities from their definitions (classes) then specifies the causalities of events among the instances (objects) using three operators: parallel, sequence and alternative execution. Having a structure of behavior axiom similar to that of classes, contexts can use conditional expressions(on ADTs) to decide the selection of control flow.

Figure 3.3.a and 3.3.c depict the basic CO-OPN constructs describing an activity and a sequence of activities. The corresponding Petri nets representation is also given in Figure 3.3.b and 3.3.d respectively, with less expressive power. The method *in_:case* accepts a token (an ADT Term) and puts it into place $p_{\_}$. Since

*In fact, because "type" is more common in object-oriented languages, we use the term "type" instead of "sort" which is the name of a type in the theory of Algebraic Abstract Data Type.

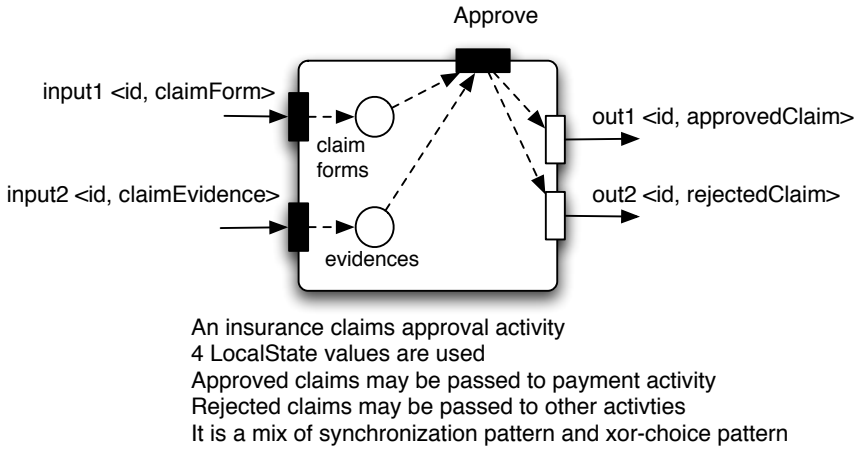


Figure 3.2: Example of Activity and Cases as CO-OPN components

we don't limit the capacity of the place here, this method is always firable thus an activity will always accept input cases. The transition **E** removes tokens from the place, and according to the value of *LocalState*, it generates output to gate *out_:case*. By simply connecting input and output of two activities, we obtain a sequence of activities. In further discussion, we will show that this basic construct of activity can be extended with little effort by adding more inputs and outputs and by specifying their corresponding axioms. Figure 3.3.a and Figure 3.4 illustrate the two modes of activity, *Normal* and *Immediate*:

- Essentially, an *Activity* has one or more *inputs* receiving cases, zero or more *outputs* sending cases out and some axioms specifying the relationship between inputs and outputs. An activity without output signifies an implicit termination of process.
- *Normal Mode Activity:* A normal mode activity has *one place for each input*. A trigger (**E** in Figure 3.3.a) takes input cases and produces output cases according to the axioms. Like in Petri nets, the control moves when a trigger is fired.
- *Immediate Mode Activity:* In an immediate mode activity, inputs and outputs are synchronized, hence no trigger is needed for output, furthermore, places for storing cases are optional.

The difference between normal mode and immediate mode activity is that immediate mode activities produce output as soon as input cases satisfy the output conditions (e.g. in the AND-Join pattern, all input cases arrive). In a normal

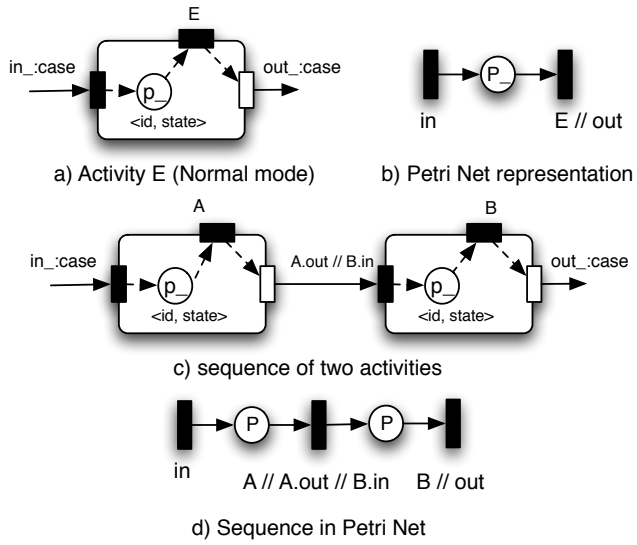


Figure 3.3: Basic CO-OPN Building Blocks for Control-Flow

mode activity, a trigger should be fired explicitly to produce outputs. Because all firable events are synchronized in an immediate mode activity, it is relatively easier to verify a process which contains *only immediate mode activities*. It means that if a legal input is not firable, the process (which is a single transaction) has deadlocks. However, immediate mode activities have restrictions when building certain workflow patterns due to possible stabilization problems of a given system that include cycles. In consequence, our further discussion on workflow patterns will consider normal or mixed mode activities.

We distinguish three kind of methods and gates to model *control*, *data*, *operational* flows:

- Control flow: flow between activities, in general this kind of methods and gates is always firable in normal mode activities, which means an activity does not refuse incoming cases. Moreover, their parameters only contain values of the type *Cases* or its sub-types.

- Data and information flows: additional information provided by the process execution system may influence control flow, they can use any ADTs as parameters.

- Operational flow: invoke external services and accept callbacks. This kind of methods and gates is not always firable, and their parameters can also be any ADTs.

In our approach, CO-OPN workflow models handle only control flow, the data flow and operational flow are managed by the process execution system. However, some methods and gates are used to interact with the process execution system, e.g. tell the process execution system to invoke a service.

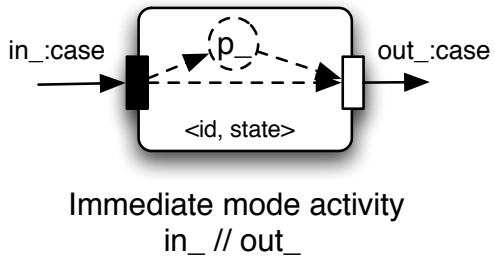


Figure 3.4: Immediate Mode Activity

3.2.2 Workflow Patterns

Workflow patterns will be built as CO-OPN components with local states described by tokens, methods as input events, external events also as methods and gates as output events. Composition will be done by directly linking gates to methods. As synchronization rules will be given within pattern, only simple synchronization will be used among patterns. The workflow pattern typical list is borrowed from the pioneer work of W. van der Aalst.

Let us take an example of *synchronizing merge* to explain how to build workflow patterns with CO-OPN axioms. A synchronizing merge multiples paths converge into one single flow. If more than one path is taken, synchronization of the active flows needs to take place. If a token representing a case instance value is passed to the activity and presents in the place, the path is active for the case instance value. In our example Figure 3.5, if both *input1_* and *input2_* are active at the time we fire **E**, the two flows are merged, then *out_* is activated only once. If only one path is active at the time we fire **E**, *out_* is activated once for this path, and if afterwards another path becomes active, **E** will also be firable and *out_* will be activated another time. Note that the value of *out_* can be any value of type *Case*. For the sake of simplicity, in following figures and lists of axioms, we use **T, T1, T2** or **c, c1, c2** as values of output cases.

The following axioms describe the pattern *synchronizing merge*, an activity with two inputs and one output. T, T1, and T2 are ADT terms of type *Case*, where T=f(c1,c2), T1=f(c1), T2=f(c2). f is the function between inputs and outputs.

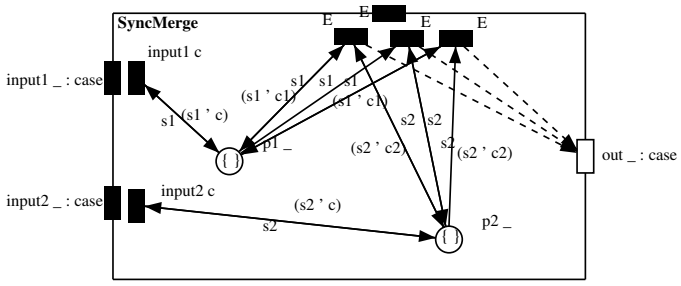


Figure 3.5: Workflow Pattern: Synchronizing Merge

Expression *c1 sameID c2* is a condition that means c1 and c2 identify the same case instance; **c**, **c1** and **c2** are variables of type *Case*; **s**, **s1** and **s2** are predefined **Sets** containing *Case* values; `s'c` is the set of adding element **c** into set **s**; `s1^s2` is the intersection set of **s1** and **s2**; `# s` is the number of elements in set **s**. *c isin s* is true if **c** is in set **s**. Initially, **s1** and **s2** are empty.

```
axiom1: input1 c::p1 s1 -> p1 (s1'c);
axiom2: input2 c::p2 s2 -> p2 (s2'c);
axiom3: c1 sameID c2, s3 = s1^s2, (# s3 > 0) = true,
   (c1 isin s3) = true, (c2 isin s3) = true =>
   E With this.out T:: p1 s1'c1, p2 s2'c2 -> p1 s1, p2 s2;
axiom4: s3 = s1^s2, (# s3 = 0) = true =>
   E With this.out T1:: p1 s1'c1, p2 s2 -> p1 s1;
axiom5: s3 = s1^s2, (# s3 = 0) = true =>
   E With this.out T2:: p1 s1, p2 s2'c2 -> p2 s2;
```

axiom1 and *axiom2* state that *input1_* and *input2_* will put the value *c* into set **s1** of *p1_* and set **s2** of *p2_*, respectively. *axiom3, axiom4, and axiom5* describe the behavior of trigger **E**: when two tokens of the same instance, **c1** is in **s1** and **c2** is in **s2**, firing **E** will remove **c1** and **c2** from the sets and send out T=f(c1,c2); if only one token exists, firing **E** will remove the token and send out a corresponding output; if there is no cases values in **s1** and **s2**, **E** will be not be firable.

Table 3.1 lists some other workflow patterns proposed by [72]; names and descriptions of patterns are kept for reference purpose.

3.2.3 Transactional Workflow Patterns

A transaction in a workflow is one atomic event abstracting a more complex behaviour, and this is directly modeled by using synchronization mechanism on submodels of workflows. In Figure 3.6 we show an example of transactional patterns,

Pattern name	**Axioms**	**Comments**

Parallel Split	`input c:: p s-> p s'c;`
	`E With this.out1 T1 //`
	`this.out2 T2::`
	`p s'c -> p s;`
		A single flow of control is split into multiple flows of control. Firing transition E will trigger gate *out1* and *out2*, sending out T1 and T2. **Synonyms**: *AND Split*

Synchronization	`input1 c1:: p s1-> p1 s1'c1;`
	`input2 c2:: p s2-> p2 s2'c2;`
	`c1 sameID c2 =>`
	`E With this.out T::`
	`p1 s1'c1, p2 s2'c2`
	`-> p s1, p s2;`
		Multiple flows of control converge into one. Transition E is enabled when all inputs have arrived, firing E will send out T. **Synonyms**: *AND Join*

XOR Split	`input c:: p s-> p s'c;`
	`E With this.out1 T1::`
	`p s'c -> p s;`
	`E With this.out2 T2::`
	`p s'c -> p s;`
		Based on a decision or value of case instance, one of several branches is chosen. Firing E will trigger one of gate out1 and out2, sending out T1 or T2. **Synonyms**: *Exclusive Choice*

XOR Join	`input1 c:: p1 s1-> p1 s1'c1;`
	`input2 c:: p2 s2-> p2 s2'c2;`
	`E With this.out T::`
	`p1 s1'c1 -> p1 s1;`
	`E With this.out T::`
	`p2 s2'c2 -> p2 s2;`
		E is enabled for each input. The inputs are not synchronized. **Synonyms**: *Asynchronous Join*

Multiple Merge	`input1 c:: p1 s1-> p1 s1'c;`
	`input2 c:: p2 s2-> p2 s2'c;`
	`E With this.out T1::`
	`p1 s1'c -> p1 s1;`
	`E With this.out T2::`
	`p2 s2'c -> p2 s2;`
		Multiple paths converge into one single flow. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch.

Deferred Choice	`input c:: p s-> p s'c;`
	`E1 With this.out1 T1::`
	`p s'c -> p s;`
	`E2 With this.out2 T2::`
	`p s'c -> p s;`
		Two alternatives are offered to environment: E1 and E2 are all enabled, firing one will disable another. **Synonyms**: *Deferred XOR-split, External choice*

Table 3.1: Workflow Patterns in CO-OPN

where **P** is a sub-model of workflow containing three activities **A**, **B**, and **C**: **A** is an AND-SPLIT activity, **B** is a simple activity, and **C** is an AND-JOIN activity. In particular, input *in2_* of **C** is independent of these three activities. A higher level trigger **TP** is synchronized with the triggers of these activities, specified by an axiom: **TP With TA..(TB//TC)**. When TP is fired, the model tries firing TA at first, then firing TB and TC at the same time. Since one input of AND-JOIN C might not be satisfied at the moment we try firing TC, TC may fail. According to the principle of synchronization, TB, TA and TP will fail, as a consequence, the case instance stays in A, neither B nor C are aware of this attempt.

In this example we observe that the synchronization propagates to an AND-JOIN activity and fails. In fact there are many reasons why a synchronization may fail, especially when dealing with immediate mode activities and operational methods e.g. invoking external synchronous services.

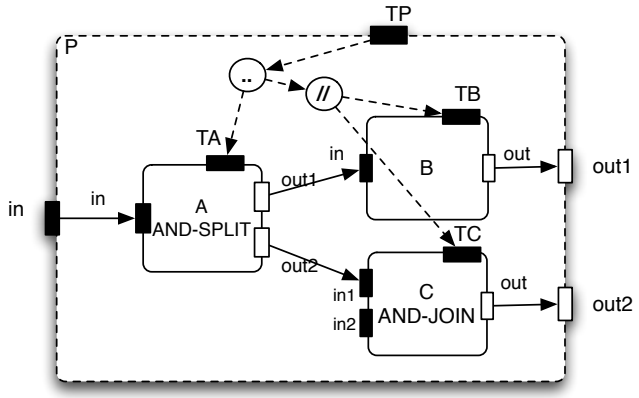


Figure 3.6: Transactional workflow patterns: TP with TA..(TB//TC)

In general, the use of CO-OPN for modeling workflows is based on the principle that hierarchy of context captures the different levels of transaction while linear interconnection of components is used for describing the composition of workflow. This principle is very close to the use of CO-OPN for modeling Coordinated Atomic Actions (CAA) [73] [74]. The hierarchical approach is not fully satisfactory, in particular for shared resources (the payment system, for instance, that can be reused in the context or other online shops). In this case the solution is to not include this resource in the surrounding context. Instead, as it must be accessible to various partners, this resource should be defined as a shared object as it was done in CAA.

3.3 Modeling Long-Lasting Transactional Business Process with CO-OPN

In the context of business process management, transactions are executed in distributed, heterogeneous, multi-database environment, frequently they are long-lasting, interactive, and dynamic. These transactions are called Long-Running Transactions (LRT) or Long-Lasting Transactions (LLT).

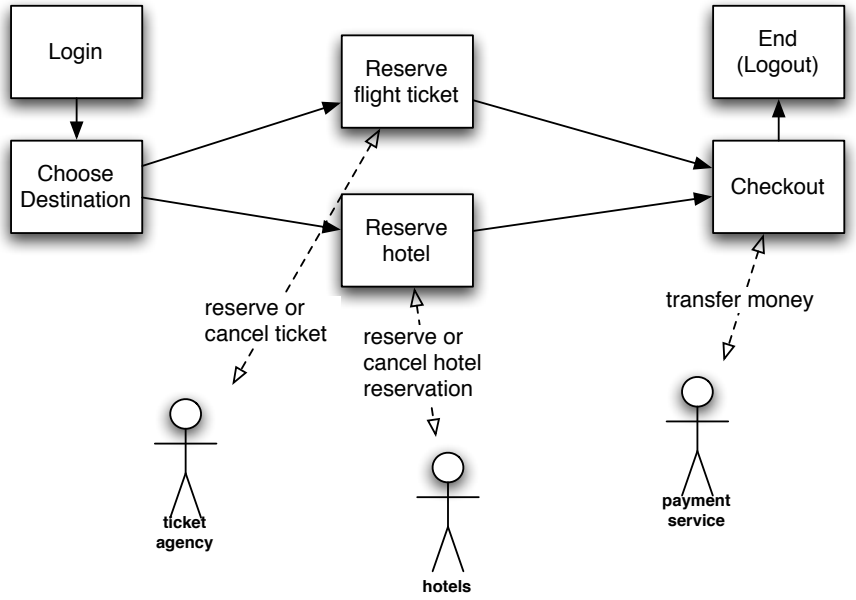


Figure 3.7: Online Trip Reservation Process

Figure 3.7 shows an example of online travel reservation process. Making flight and hotel reservation for a business trip is a typical transactional process that can last a long time and sometimes needs more than one session to be completed. It is therefore not possible to do the whole reservation procedure within one classical transaction. During the execution of the process, the BPMS interacts with different information systems and users to achieve tasks defined by the process. In case of transaction failure or canceled by user, the BPMS should inform all participants involved in the process to perform the rollback.

Most existing workflow models, including Petri nets approaches, are concentrating on inter-activities dependencies and inter-transaction dependencies without considering the properties of transactional processes, thus transactional processes are often handled in an ad-hoc manner. On the other side, people from the database domain try to model long-lasting transactions (LLT) and inter-transaction dependen-

cies by relaxing traditional ACID properties and proposing Advanced Transaction Models (ATM) such as Saga [75], ACTA [76], and ConTract Model [77]. However, their approaches of managing inter-transaction dependencies are not widely accepted by process modeling community because in many cases they are too database-centric or restrictive, e.g. all operations in a transaction should have a corresponding undo or compensation operation.

WS-Transactions Specifications [78] propose a transaction framework to handle transactions in Web service based environment. This framework supports both short duration ACID transactions and long-lasting business transactions. Developers can define and implement specific business agreement protocols within the framework. It can be incorporated with BPEL [12] to realize transactional business processes, this means that the transaction is handled at the Web service level and not in the BPEL process models. Other similar standards are Business Transaction Protocol [79] from OASIS and Web Services Transaction Management Specification [80] from Sun. These three standards are proposed to handle transactions related to Web services composition. The discussion on their differences is beyond the scope of this work. Nevertheless, we would like to emphasize that these approaches are more technical than methodological, because neither process model nor transaction model are described by formalisms with which the properties and the correctness of model can be proofed rigorously.

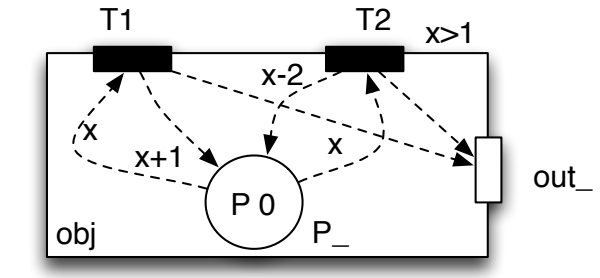
Zero-Safe nets [61] uses invisible *zero places* to connect transitions which participate in a transaction. The zero-safe nets are a refinement of the underlying Petri nets by providing information about the transactions. The real state of Petri nets is called the *stable marking*. The difference between CO-OPN and zero-safe nets is that CO-OPN supports transactions at semantic level via events synchronization operators, while zero-safe nets uses the classical Petri nets semantics to model transactions by adding zero-safe places. Both of them can be mapped or projected into classical Petri nets in order to verify the properties of models using common analysis techniques.

3.3.1 Properties of CO-OPN Transactions

CO-OPN transactions are *synchronous* and of short-duration. A synchronization expression, even it is complex, is one single *transaction*. The result of a CO-OPN transaction can be *failed* or *success* like ACID transactions. A CO-OPN transaction is a CO-OPN transition which synchronizes *with* other transitions; Transitions that do not synchronize with other transitions are *elementary*; Transactions are composed by other transactions and elementary transitions.

CO-OPN transactions are dynamic if the synchronization expression uses the

non-determinism operator $\oplus$. For example the transaction *T1..(T2⊕T3)* can be executed as *T1..T2 or T1..T3*. The sequence is determined at the moment of execution and depends on the firerability of *T2* and *T3*.



Initially, place P_ contains one token "0".
x is a variable.
Axioms for T1 and T2:
T1 with this.out x+1:: P x -> P x+1
T2 with this.out x-2:: (x>1)=true => P x -> P x-2

Figure 3.8: A Simple CO-OPN Object

Figure 3.8 shows a simple object which changes its internal state in response to two transactions *T1* and *T2*. The simple object has a place *P_* initially contains a token *0*, which is a natural number. *T1* takes this token, adds 1 to its value and puts it into the place; *T2* subtracts 2 from the value if the value of the token is greater than 1; both of them are synchronized with gate *out_* which outputs the value of the token after transition. By the semantics of Petri nets, *T1* and *T2* are atomic and they access the same resource (token), thus a concurrent problem exists if *T1* and *T2* are fired simultaneously: transaction *T1 // T2* is not fireable whatever the value of the token is. Moreover, with this initial state:

- Transaction *T1..T2* is not fireable because the token has value "1" after firing *T1*, the condition of *T2* is not satisfied, so *T2* fails and consequently *T1..T2* also fails, the state of object does not change; The transaction *T1..(T1..T2)* is fireable with output "0", *T1..(T1..(T1..T2))* is fireable with output "1", and so on.

- Transaction *T1⊕T2* is fireable and will output value "1". Only *T1* is fireable in this situation; *T1..(T1⊕T2)* outputs value "2", *(T1..T1)..(T1⊕T2)* outputs value "3" or "0", and so on.

It should be noted that in the object the gate *out_* takes part of the transaction *T1* and *T2*, which means if *out_* is not fireable due to outside constraints, *T1* and *T2* will not be fireable either. This characteristic is very useful for synchronizing external

events with CO-OPN transaction, as we will do with the transactional activity and external services.

3.3.2 Modeling LRT Business Processes

Business process transactions extend traditional ACID transactions because they are not just statically defined and neither always short-lasting. Figure 3.9 shows a possible composition of process, activities, and operations: processes are composed of several activities; activities can pass control flow to another process, activities, or operations; operations are stateless services provided by external systems, e.g. Web services.

In this example **xor** and **and** patterns are used. P2 is a sub-process of P1, processes P1 and P2 can be declared as transactional or not. If P1 is transactional, P2 should also be transactional, because if we abort activity A3, P2 will also be aborted. We consider *a transactional business process* as a business process which supports the operators of transactions: *start*, *commit*, and *abort* with respect to the property of atomicity.

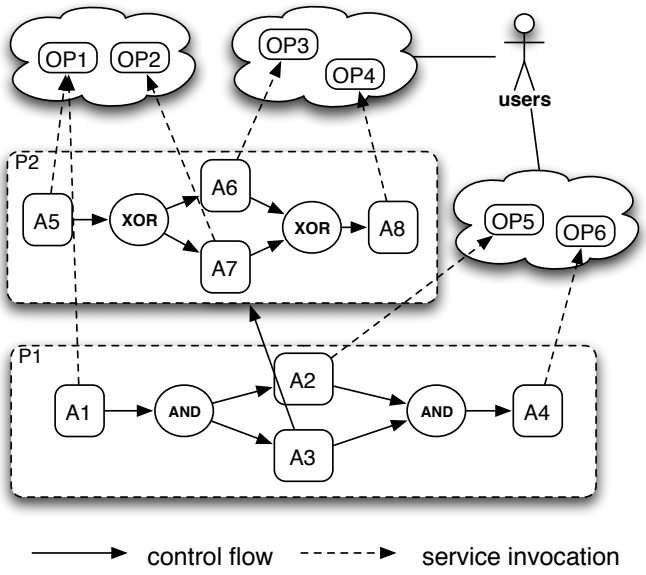


Figure 3.9: Example of Transactional Business Process

The underlying Petri nets model of a transactional business process should be *reversible* and *sound*. *Reversible* means if the transaction is aborted, the state of system moves back to the beginning of the transaction (*home state*); *soundness* is an important property [81] for workflows: for all states reachable from the beginning of the transaction, there exists a firing sequence leading to end of the transaction

(acceptable end states); if the transaction is committed, no hanging tasks and tokens related to the process instance exist in the system; there are no dead transitions in the system.

We consider a *business process transaction* as a sequence of operations which should be executed (a possible run) from the beginning to the end of the process. Obviously, a transactional process can have many possible runs because the sequence of transaction is dynamic and determined during the execution of the process. A sequence *satisfies* the process if it is a possible execution of the transactional process.

The start and the end of a transaction can be specified by some conditions. In our example, a transactional process instance is *started or active* when an instance token is passed into the process through the *in_* method of its first control flow activity; the process is *ended* for the instance if the token leaves its last control flow activity. The participants of the transactional business process are its internal transactional activities and are determined during the process execution. The additional characteristics of business process transactions are:

- *Dynamic.* The sequence of operations are not determined until the execution of process. For instance, the sequence of activities in P1 can be {A1, A2, A4}, {A1, A5, A7, A8, A2, A4} or {A1, A5, A6, A8, A2, A4}, the sequence of operations depends on the sequence of activities.
- *Long-Lasting and Interactive.* The duration of each elementary step of the execution of operations is long and the transaction can not be finished in one session. Some operations consists of getting user inputs or waiting certain events. This prohibits the resource locking techniques used by most short-time concurrency control mechanisms.

Dynamic transaction problem - need of logging In figure 3.9, if we consider P2 as a transactional workflow, its *abort* sequence depends on the OR selection between A6 and A7. Since the process is interactive, the decision might be made by users in a non-deterministic manner (workflow patter: deferred choice), in this case, the only thing we can do is to log the executed operations before the transaction is committed. If we don't log the executed sequence, in activity A8 we do not know which activity between A6 and A7 was chosen, hence it is not possible to undo executed operations.

In this section, we present a three-level architecture to model the behaviors of transaction coordinator, the transactional process activities, and the transaction participants using the Two-Phase Commit (2PC) protocol. The transactional activities are committed before the process finalizes and at each moment we can abort the transactional process. Other transaction protocols can be easily derived from

the same principle, e.g. by using different transactional activity blocks, we can implement the *BusinessAgreementWithParticipantCompletion* Protocol proposed in WS-Transactions [78].

Most modern distributed systems use *transaction coordinators* to manage the enrollment and execution of transactions, as shown in figure 3.10.a. The advantage is the transaction protocol is controlled by the coordinator and the participants do not need to know about it.

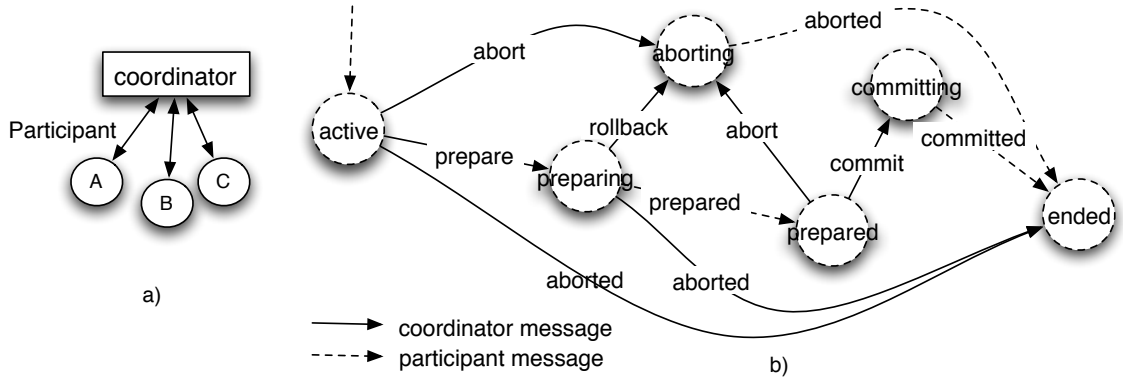


Figure 3.10: State Transition Diagram of Two-Phase Commit Protocol

Figure 3.10.b gives the state transition graph of two-phase commit protocol, where the state of transaction is driven by the messages exchanged between the coordinator and participants. Since it is possible to use 2PC protocol to manage long-lasting transactions as the Business Transaction Protocol [79] does, we are using it to illustrate *the relationship between the coordinator, the transactional activity and the external service (participant).*

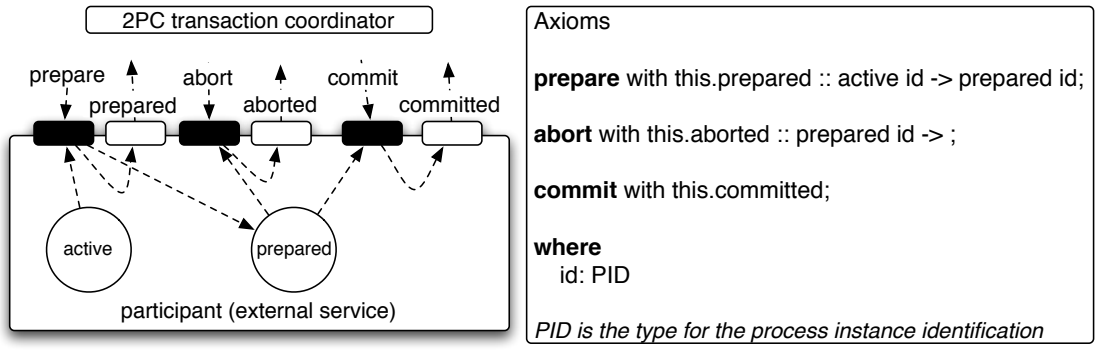


Figure 3.11: A CO-OPN 2PC Transaction Participant

Transaction Participant A transaction participant is a concrete service which realizes the piece of work defined by an activity. In contrast with normal services, a transaction participant is a service which supports rollback or compensation, i.e. it accepts these requests from the transaction coordinator. Figure 3.11 shows a CO-OPN modeled 2PC transaction participant, the axioms specify that it responds immediately to the transaction coordinator for each received message. In the implementation phase, it is replaced by a real Web service with transactional support.

Transactional Activity An activity which supports transactions is called a *transactional activity*. Figure 3.12 depicts a CO-OPN component which stores the state of 2PC transaction. The component can be different for other transaction protocols, e.g. one phase commit or completion protocol. A transactional process is composed by transactional activities. The transactional activity is the middle layer which interacts with the transaction coordinator to realize the transaction, and at the same time, it ensures the invocation of external services, i.e. connect to Web services and handle the responses.

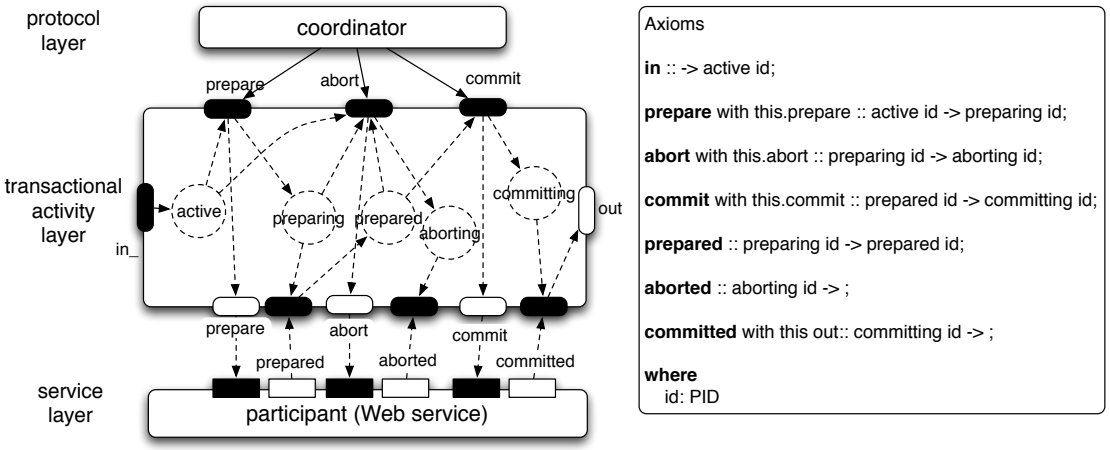


Figure 3.12: A Two-Phase Commit Transactional Activity

As we can see in figure 3.12, the transactional activity delegates calls such as *prepare*, *abort*, and *commit* to the concrete transaction participant, an external service. The action *prepare* of the transactional activity and the same action of the participant are synchronized. This means if anything goes wrong with the participant and its action can not be fired, the transaction activity model refuses to fire because the participant is not ready. Thus the transaction coordinator should find other solutions to proceed the transaction.

Transaction Coordinator A transaction coordinator manages the participant enrollment and the execution of transactions. The coordinator can be another CO-OPN component or an external application which implements transaction protocols. The advantage of including the coordinator in the model is we can verify the properties of the transactional protocol, which is very important and useful. The principe is similar to Coordinated Atomic Actions (CAA) which is modeled with CO-OPN [74].

In figure 3.13, we give an example of 2PC transaction coordinator modeled by CO-OPN. Each time the method *prepare* is fired, it informs the activities to *prepare* the transaction, then move the corresponding token into place *prepared*; its method *commit* becomes firable only if all activities are prepared, firing it will send message to all activities to perform *commit*; at any time before the transaction is committed, we can abort the transaction by firing its method *abort*. If the method *commit* is not firable for a long time, i.e. a timeout, the method *abort* can be triggered automatically. This is a common implementation choice. Note that the coordinator use the process identifications to organize the transactions, several business process instances may participate to the same transaction and a transaction may have its own identification.

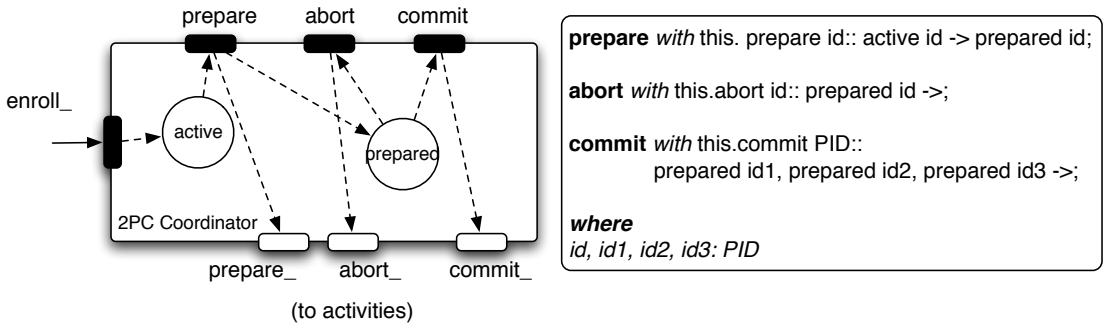


Figure 3.13: A 2PC Transaction Coordinator

Build Transactional Business Processes With the three-layers architecture [†], it is easy to build transactional business processes as shown in figure 3.14 (only the details of about the action *commit* is given for better readability):

- a transactional business process is a CO-OPN context which contains (or acts as) a transaction coordinator and several transactional activities. In figure 3.14, the context acts as the transaction coordinator together with two activities *Start_Transaction Activity* and *End_Transaction Activity*.

[†]In programming, this architecture is called the *delegation pattern*.

- the transaction protocol is managed by the transaction coordinator, each activity of transactional layer delegates an internal activity or an external transactional service (participant).

- the transaction coordinator and the transactional layer monitor the state of each transaction, i.e. an activity is activated, external service is prepared, external service is invoked, etc.

- action *T_commit* is firable only when all the activities are ready to commit. At any time, it is possible to abort the long-running transaction.

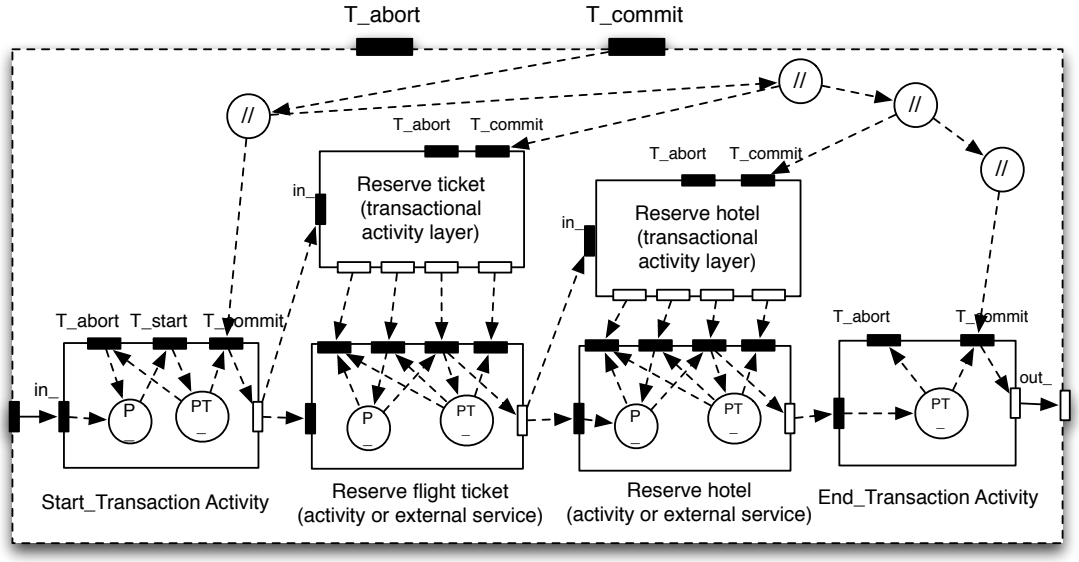


Figure 3.14: Transactional Process for Ticket and Hotel Reservation

Tracing dynamic transactions The sequence of invoked services are are recorded in the transactional activities. Because only invoked activities will enroll with the coordinator, it is easy to find the sequence and compensate the invoked services before the transactional business process is committed. In this way, the dynamic problem of transactional business process shown previously in this section is undertaken and the states of all participated transactional activities are conserved in the model.

3.4 Prototype Generation & Integration

In this section, we will explain how to validate business processes by prototyping. This will mainly be done by generating process controller prototypes from CO-OPN specifications and then interpret or simulate prototypes in the CoopnBuilder environment. Finally we will compose these prototypes with the process execution systems.

Tools Supports and Prototype Generation CoopnBuilder [82] is the latest generation of tools supporting CO-OPN language, this continually developing tool provides an integrated development environment to edit, verify, and visualize CO-OPN modules. Moreover, the included prototype generator produces executable JavaBean-like components from CO-OPN specifications. These components implement the underlying CO-OPN specifications by means of the embedded state-transition system with well-defined management of synchronization and atomicity. Methods of CO-OPN modules are transformed to Java methods with support for transactions. To capture outgoing events, we add *EventListener* on the *gates*. For transactional processes, all the activities which manage the transactions are transformed into the prototype, i.e. the prototype will manage the states of the long-lasting transactions.

Prototype Integration We have shown how to build complex processes by means of workflow patterns, now we will illustrate how the process execution system interacts with process controller prototypes and external process participants. A simple process with three activities is given in Figure 3.15. The *Shopping Process* is a CO-OPN context incorporating *order*, *transaction*, and *delivery* activities. Execution of activities are sequential in the process. The context delegates some methods and gates of internal activities, especially the inputs of case instance and the operational actions. All inputs and outputs of the context are handled by the process execution system. According to output messages of controller prototype, the process execution system communicates with external business process partners, who effectuate the real work of activities; when a process relevant event raises from the partners, e.g. payment by credit card is finished, the process execution system *"tells"* the controller prototype to change the state of related process instance. The controller prototype can be considered as an *"Abstract State Machine"* and the "brain" of the process execution system. A process execution system is a *"service provider"* of all embedded process controllers. The prototype integration consists of connecting the generated process controller prototypes to a process execution system, e.g. similar as using any Java components.

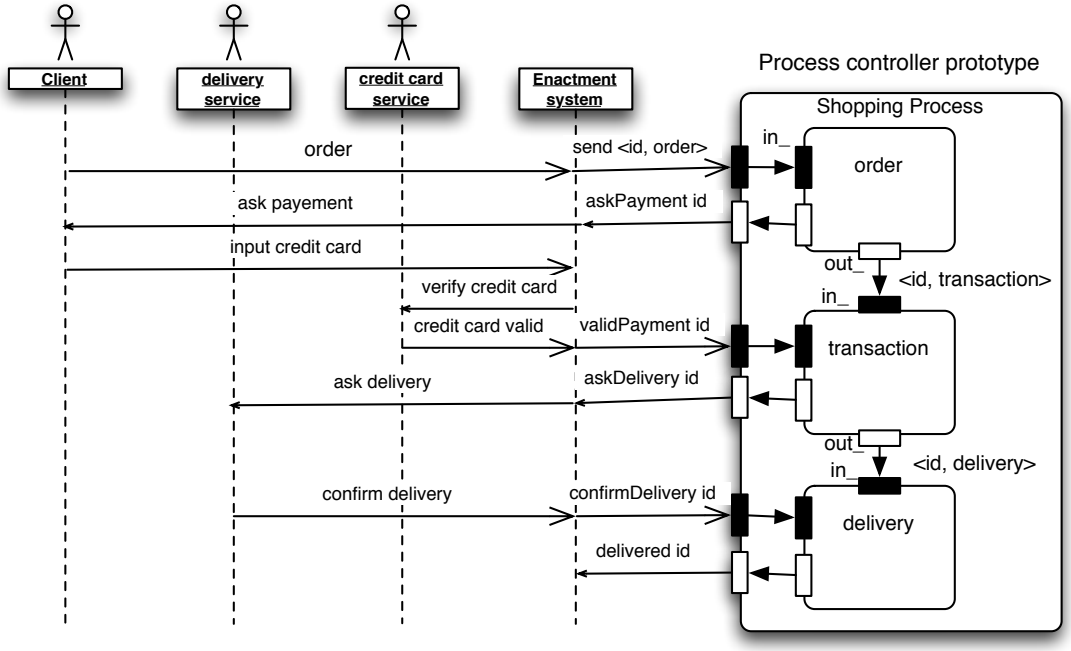


Figure 3.15: Prototype Integration: an Online Shop

In appendix A.1, we give some implementation details about generating Java codes from CO-OPN specifications, especially for CO-OPN transactions. Further details on prototype generation and integration can be found in related work: [83] demonstrates using prototypes as controllers of embedded systems. [84] explains techniques for interfacing generated prototypes with existing Java libraries.

3.5 Summary

In this chapter, we explained the modeling and prototyping of business process with CO-OPN, a formal specification language based on abstract algebraic data type and object-oriented Petri nets.

One important concept we used to model business process is *transaction*: CO-OPN transactions are short-duration ACID transactions as in databases; Long-Lasting Transactions are common in business process which need special care during the modeling phase (e.g. compensation or rollback). We show that CO-OPN has rich expressive power to model the transactional behaviors of business process.

Moreover, we demonstrate the mechanisms of generating Java codes from CO-

OPN specification (in Appendix), i.e. how CO-OPN classes, contexts, and axioms are transformed into Java. The generated codes are the prototypes of the business processes and they can be integrated into a project to simulate and test the process.

However, as other approaches of process modeling in academic research, this approach is well-suited for prototyping and model validation, but it has limitations in real-world business process software development:

- algebraic abstract data type is an abstraction of data type used in real-world applications. It has rich expressive power but it is not practical to use algebraic data types in software development. Currently the most used data model in information system is the relational model.
- data-flow is essential in workflow and information systems. It is possible to do data-flow modeling in CO-OPN with algebraic data types, but the control flow and data flow will mix and the complexity of models will increase dramatically, this will scare general software developers. It is one of the reasons why we need to apply the principle of separation of concerns and incremental development.
- the realization of business process depends of its execution environment, more details have to be added into the process models to make them executable, i.e. adding problem dimensions into the models.

In the following parts of this work, we propose a multi-dimensional, compositional approach to model and develop real-world business process. This proposal can be considered as an attempt of integrating CO-OPN into software development process because many fundamental concepts of our language come from CO-OPN: synchronization, concurrency, service-orientation, and encapsulation. However, our proposition has several improvements with respect to the software development process:

- we *externalize* the algebraic abstract data types (ADTs) as domain-specific data models which synchronize with Petri net models (i.e. ID-net). Thus any kind of model can be used to represent data and resources.
- we consider business processes as multi-dimentional models. We use synchronizations to incrementally compose different kinds of models from different problem dimensions, i.e. applying the Multi-Dimensional Separation of Concerns [3, 85], and the Aspect-Oriented Development (AOD) [86] paradigm.

The semantics of synchronizations should be guaranteed by concrete implementation. By means of the externalization and composition mechanism, we can enrich our models incrementally during the development of business process.

Chapter 4

Semantical Model Composition

4.1 Model Composition

Complex models and systems are built through composition. The *Separation of Concerns principle (SoC)* is one of the key principles in software engineering. In SoC, a given problem involves different kinds of concerns which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability.

In software development, SoC is the process of breaking a software system into distinct features that overlap in functionality as little as possible. All programming paradigms and modeling approaches aid developers in the process of improving SoC, e.g. functional programing, modularity, encapsulation (information hiding), and layered system designs can be considered as mechanisms of applying SoC in software development.

The multi-dimensional separation of concerns (MDSoC) [3, 85] is an abstract approach which believes that software systems can be constructed by assembling/-compositing reusable models (*HyperSlice*) together via the notion of unit from *HyperSpace*. Another important tendency in software development is the aspect-oriented software development (AOSD) [86]. AOSD concerns principally the separation of crosscutting concerns from principal model, to avoid code scatting and tangling, and obtain the modularity of crosscutting concerns (called *aspect*). Our concept of model composition is partially inspired from the MDSoC, and technically we adopt the AOSD.

Composability and Its Requirements. The requirements of the composability of models can be classified into two levels: the metamodel (syntax) level and the

semantics level. Even two metamodels are *composition compatible* for a composition, the composition should be semantically sound, e.g. while composing Web services with BPEL, not only should the signatures of the service operations be verified, but the characteristics of the service such as *request* and *request-response* should also be taken into consideration to avoid the deadlock of process. The design and the expressiveness of the composition language have direct impact on the composition of semantics.

Figure 4.1 shows some examples of model construction viewed as model composition. Some compositions also provide new constructs and operators, e.g. control-flow blocks are used to compose services, and temporal operators are proposed to compose sequential-execution programming languages as in AspectJ [87].

Kind of Models	Elements	Composition techniques
Object-Oriented Model	- class - method - member	* Association, Aggregation * Generalization /Specialization * e.g. class diagram
Data Modeling	- entity - attribute - relation	* Relation * Candidate key, Foreign key * e.g. Relational model, ER diagram
Service	- service - signature - data type	* Flow: parallel, sequence, join, split * Service invocation * Transactions * e.g. WS, BPEL
Programming Language	- data type - flow block - function	* Function composition, e.g. LISP, Haskel * Temporal operators, e.g. PCD in AOP, AspectJ

Figure 4.1: Some Model Construction/Composition Techniques

If the models to be composed are orthogonal, i.e. without overlapping domains, the semantics of resulting models will be the union of the semantics of the original models. However, most of the time this is obviously not the case.

Managing complexity. Although model composition is essential to build systems, it can be achieved technically in many ways. In software domain, a more important task is managing the complexity and it is crucial for safety-critical systems. For instance, design of concurrent system requires rigorous model verification techniques to analyze the behaviors of each process and the whole system. The dinning philosophers problem is a classical example. In such cases, it is easy to put many simple processes (e.g. philosophers and forks) together, but the complexity increases exponentially and their behaviors become unpredictable, e.g. deadlocks in

operation systems. Thus, while building complex systems from smaller systems, an important rule is *know the semantics of the composition at each step.*

4.1.1 Composition Language

We will analyze the effects of model composition at the semantic level, i.e. independently from the syntax used during the modeling phase, in terms of behaviors how a system is built through the composition of smaller systems. In order to do this, we use a simple composition language to synchronize behaviors of state-transition systems. The core concept of this composition language is *transaction* (of actions) or *synchronization* (of events).

We use two operators to define the composition: simultaneous (//) and exclusive choice ($\oplus$):

- simultaneous actions/events. Event **a** in system **A** and event **b** in system **B** must *simultaneously* occur. Denoted as **A.a // B.b**.
- exclusive choices. At least one but not both event will occur: event **a** in system **A** and event **b** in system **b**. Denoted as **A.a** $\oplus$ **B.b**. Exclusive choice is used to represent the branching and non-deterministic behaviors in a system.

Visibility of actions While composing several systems together, some new actions are created in the composed systems, and we would like to hide some original actions. Thus, we introduce the *visibility* modifier$^{\sim *}$. For example, synchronization $\widetilde{A.a}//B.b$ means the action $\widetilde{A.a}$ will be visible in the composed system but $\widetilde{B.b}$ will not.

The composition operators are defined in the followed section.

4.2 Synchronized Composition of Labeled Transition Systems

In theoretical computer science, the semantics of a language or a model, e.g. Petri net, process algebra, and many DSLs, can be given by state-transition systems [70]. We use labeled transition system as the basis of system composition to become independent from a concrete modeling language.

$^{*}Infact, \sim$ can be considered as the inverse operation of "restriction" in process algebra, i.e. actions without $\sim$ are restricted in the composed system.

Definition 4.2.1. ***Labeled Transition System.*** *A labeled transition system is a quadruple* $\mathcal{A} =< S, A, T, s^0 >$ *where:*

- S *is a set of states,*
- A *is a set of actions names (labels), called alphabet*
- $T \subseteq S \times A \times S$*, denoted as* $s \xrightarrow{a} s', a \in A, s, s' \in S$*, is a transition relation.* $\alpha : T \to S$*,* $\beta : T \to S$*,* $\lambda : T \to A$ *are mappings from a transition to its source state, target state, and label, respectively,*
- $s^0 \in S$ *is the initial state.*

Labeled transition systems are directed graphs where nodes represent states, edges model transitions, and labels are annotations on edges. By definition, each transition must only have one label. The product mapping $< \alpha, \lambda, \beta >: T \to S \times A \times S$ is injective, i.e. two different transitions cannot have the same source, target, and label at the same time. However, a LTS can be nondeterministic, i.e. the same action can provoke two different transitions leading to different states.

Each LTS transition is an instance of an action. While describing the behaviors of reactive systems, the set of labels A often includes the set of acceptable actions and the set of observable events of the underlying system.

Counter example. We define a simple counter with 4 states, one action *inc*, initial state 0, and 4 transitions, i.e. $S = \{0, 1, 2, 3\}$, $A = \{inc\}$, $s^0 = 0$, $T = \{t1, t2, t3, t4\}$ where:

- $t1 = 0 \xrightarrow{inc} 1$, $\alpha(t1) = 0$, $\lambda(t1) = inc$, $\beta(t1) = 1$
- $t2 = 1 \xrightarrow{inc} 2$, $\alpha(t2) = 1$, $\lambda(t2) = inc$, $\beta(t2) = 2$
- $t3 = 2 \xrightarrow{inc} 3$, $\alpha(t3) = 2$, $\lambda(t3) = inc$, $\beta(t3) = 3$
- $t4 = 3 \xrightarrow{inc} 0$, $\alpha(t4) = 3$, $\lambda(t4) = inc$, $\beta(t4) = 0$

$t1, t2, t3, t4$ are all labeled by *inc*.

In general, labels are the syntax of a domain-specific language while LTS gives its semantics. For example, in figure 4.2 the semantics of action *inc* is given by the LTS.

An **atomic action** is a logically indivisible action which can result either *success* or *failure*; a *complex* action is a composition of actions, e.g. transactions.

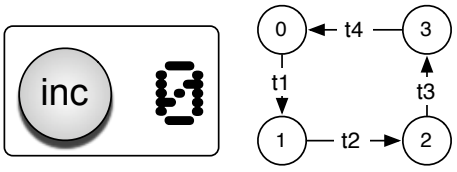


Figure 4.2: An One-button Counter and its LTS

Transactions are complex actions with the properties of atomic actions, i.e. a transaction has no intermediate states (thus logically indivisible) and also results in either *success* or *failure*. In general, a label of LTS represents an atomic action or a transaction.

Definition 4.2.2. ***Synchronization***. *Given* A *a set of atomic actions.* CA_A *is the set of possible synchronizations defined on A:*

- $a \in A \Rightarrow a \in CA_A$, *an atomic action is a synchronization*
- $a_1, a_2 \in CA_A \Rightarrow a_1//a_2 \in CA_A$
- $a_1, a_2 \in CA_A \Rightarrow a_1 \oplus a_2 \in CA_A$

The two synchronizations are commutative, i.e.

$$a//b \Leftrightarrow b//a$$

$$a \oplus b \Leftrightarrow b \oplus a$$

On one hand, synchronization is a convenient way to express *the sequence of actions to be realized with atomicity*. In this case, it is called *transaction*:

- $a_1//a_2$ means a_1 and a_2 should be performed in parallel
- $a_1 \oplus a_2$ means nondeterministically a_1 or a_2 should be performed but not the both

On the other hand, from the point of view of the observer, a transaction provides a way to describe the *events* of complex behaviors of system. For example, with this syntax, it is possible to describe the transactional behavior of an LTS, e.g. using $s \xrightarrow{a_1//a_2} s', a_1, a_2 \in A, s, s' \in S$.

Definition 4.2.3. $\widetilde{A}$. *Given A a set of atomic actions,* $\widetilde{A} = \{\widetilde{a} \mid a \in A\}$.

$\widetilde{}$is the modifier for visibility of the label in the composed system, e.g. $\widetilde{a}$ means action a is visible in the composed system.

Definition 4.2.4. ***Synchronization between two sets of actions.*** *Given A, A' two sets of atomic actions, the set of all synchronizations between A and A' is:*

$$SYNC_{A,A'} = \{< a, op, a' > \mid a \in A \cup \widetilde{A}, a' \in A' \cup \widetilde{A}', op \in \{//, \oplus\}\}$$

Syntactically, $< a, op, a' >$ can be written as in definition 4.2.2, e.g $a//a', \widetilde{a} \oplus a'$.

Definition 4.2.5. A^-_{Sync}**.** *Given A, A' two sets of atomic actions and $Sync_{A,A'}$ is a set of synchronization between A and A':*

$$A^-_{Sync_{A,A'}} = A \setminus \{a \mid < a, op, a' > \in Sync_{A,A'}, a \in A\}$$

$$A'^-_{Sync_{A,A'}} = A' \setminus \{a' \mid < a, op, a' > \in Sync_{A,A'}\} a' \in A'$$

A^-_{Sync} is the subtraction of actions involved in $Sync_{A,A'}$ from A.

Definition 4.2.6. ***Synchronized composition of LTS****. Given two labeled transition systems $\mathcal{T}_a =< S_a, A_a, T_a, s^0_a >$, $\mathcal{T}_b =< S_b, A_b, T_b, s^0_b >$, the synchronized composition of $\mathcal{T}_a$ and $\mathcal{T}_b$ is their* synchronous product *$\mathcal{T}_a \times_{Sync_{A_a,A_b}} \mathcal{T}_b$ with respect to the set of synchronization specifications $Sync_{A_a,A_b}$. $\mathcal{T} = \mathcal{T}_a \times_{Sync_{A_a,A_b}} \mathcal{T}_b =< S, A, T, s^0 >$ is the composed system, where:*

- $S \subseteq S_a \times S_b$
- $s^0 = (s^0_a, s^0_b)$
- $Sync_{A_a,A_b} \in SYNC_{A_a,A_b}$

Sync specifies how actions in the two systems are synchronized. A is determined according to $Sync_{A_a,A_b}$:

$$A = A^-_a \cup A^-_b \cup Sync_{A_a,A_b}$$

The semantics of synchronization specification is defined by constructing T as the least set following these rules:

- *Without synchronization:*

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, Sync_{A_a,A_b} = \phi}{(s_a, s_b) \xrightarrow{x} (s'_a, s_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, Sync_{A_a,A_b} = \phi}{(s_a, s_b) \xrightarrow{y} (s_a, s'_b) \in T}$$

- *Parallel synchronization:*

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < x, //, y > \in Sync_{A_a,A_b}}{(s_a, s_b) \xrightarrow{x//y} (s'_a, s'_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, //, y > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A_a}}{(s_a, s_b) \xrightarrow{x//y} (s'_a, s'_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, //, y > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A_a}}{(s_a, s_b) \xrightarrow{x} (s'_a, s_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, //, \widetilde{y} > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A_a}, \widetilde{y} \in \widetilde{A_b}}{(s_a, s_b) \xrightarrow{x//y} (s'_a, s_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, //, \widetilde{y} > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A_a}, \widetilde{y} \in \widetilde{A_b}}{(s_a, s_b) \xrightarrow{x} (s'_a, s_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, //, \widetilde{y} > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A_a}, \widetilde{y} \in \widetilde{A_b}}{(s_a, s_b) \xrightarrow{y} (s'_a, s_b) \in T},$$

- *Non-determinist choice synchronization I:*

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < x, \oplus, y > \in Sync_{A_a,A_b}}{(s_a, s_b) \xrightarrow{x \oplus y} (s'_a, s_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, \oplus, \widetilde{y} > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A}_a, \widetilde{y} \in \widetilde{A}_b}{(s_a, s_b) \xrightarrow{x \oplus y} (s'_a, s_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, \oplus, y > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A}_a}{(s_a, s_b) \xrightarrow{x \oplus y} (s'_a, s_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, \oplus, y > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A}_a}{(s_a, s_b) \xrightarrow{x} (s'_a, s_b) \in T},$$

- *Non-determinist choice synchronization II:*

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < x, \oplus, y > \in Sync_{A_a,A_b}}{(s_a, s_b) \xrightarrow{x \oplus y} (s_a, s'_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, \oplus, \widetilde{y} > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A}_a, \widetilde{y} \in \widetilde{A}_b}{(s_a, s_b) \xrightarrow{x \oplus y} (s'_a, s_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, \oplus, y > \in Sync, \widetilde{x} \in \widetilde{A}_a}{(s_a, s_b) \xrightarrow{x \oplus y} (s_a, s'_b) \in T},$$

$$\frac{s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, \oplus, y > \in Sync, \widetilde{x} \in \widetilde{A}_a}{(s_a, s_b) \xrightarrow{x} (s'_a, s_b) \in T},$$

The composed system respects the constraints of both initial systems.

According to rules to determine A, i.e. the visibility modifiers, some labels are removed during construction of the new system, for example:

- $s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, //, y > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A_a} \Rightarrow (s_a, s_b) \xrightarrow{y} (s_a, s'_b) \notin T$
- $s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < \widetilde{x}, \oplus, y > \in Sync_{A_a,A_b}, \widetilde{x} \in \widetilde{A_a} \Rightarrow (s_a, s_b) \xrightarrow{y} (s_a, s'_b) \notin T$
- $s_a \xrightarrow{x}_a s'_a \in T_a, s_b \xrightarrow{y}_b s'_b \in T_b, x \in A_a, y \in A_b, < x, \oplus, \widetilde{y} > \in Sync, \widetilde{y} \in \widetilde{A_b} \Rightarrow (s_a, s_b) \xrightarrow{y} (s_a, s'_b) \notin T$

In general, we suppose the systems to be composed are independent and adopt a bottom-up approach for composition.

Definition 4.2.7. ***Free Product****. A free product between two LTS $\mathcal{A}$ and $\mathcal{B}$ is synchronized composition of $\mathcal{A}$ and $\mathcal{B}$ with $Sync = \phi$, i.e. $\mathcal{A} \times_\phi \mathcal{B}$.*

Free product is used to compose parallel systems using the *interleaving executions*, i.e. transitions occur one after another. Nevertheless, our composition is valid with models using *true concurrency* because the *causality* between the actions/events are expressed by the synchronizations in our approach. This means that by chance simultaneous execution of unrelated actions has no impact on the semantics of concurrent systems, the causal relation between the actions/events will be expressed by the synchronizations.

4.3 Example of Synchronized Composition of LTS

In the following part of this section, we use several examples to show how the semantics of LTS are composed for different purposes.

4.3.1 Mutual Exclusion

Mutual exclusion (often abbreviated to mutex) is used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections (CS). A mutex can be implemented as a binary semaphore, i.e. semaphore initialized by 1. A binary semaphore, as shown in figure 4.3, has two states: 0 means it is *free*, 1 means it is *occupied* by a process; and two operations P and V.

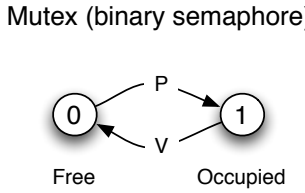


Figure 4.3: Labeled Transition System of a Mutex

A process has to acquire the semaphore in order to access a critical section. Typically, a long-running process works as follows:

```
while(true){
  sem.P();
  CS;
  sem.V();
  Non-CS;
}
```

The process can be abstracted as a labeled transition system shown in figure 4.4, it has two principal states: inside the critical section (*CS*) and outside the critical section (*Non-CS*). Two actions *InCS* and *OutCS* allow switching between the two states. In a concurrent system, there may exist many instances of this process.

In this example, we use two instances of this process A, B, and a mutex M to illustrate the behavior of concurrent processes, i.e. processes A and B should acquire mutex M in order to enter into the critical section.

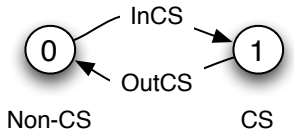


Figure 4.4: Transition System of a Process

First of all, without using a mutex, process A and B are free to change their states, this is illustrated by figure 4.5, the free product of A and B.

Notice that we simply concatenate the states of A and B to express the state of new system C, i.e. *01* means A is in state *0* and B is in state *1* etc. Prefix A and B are used to distinguish origins of the actions. Thus $AinCS$ means action $InCS$ of system A, $BOutCS$ means action $OutCS$ of system B etc.

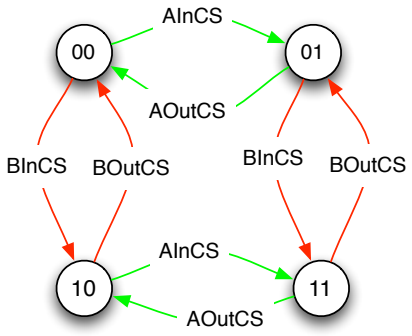


Figure 4.5: Free Product of Process A and B: $C = A \times_\phi B$

Secondly, because there are no direct interactions between process A and B, we keep system C and make a free product of C and mutex M, as shown in figure 4.6. The same state encoding and action naming rules are used. MP, MV means action P, V of M, respectively.

The set of states is shown in figure 4.6. The set of actions in system $C \times_\phi M$ is:

$$A_{C\times_\phi M} = \{AInCS, AOutCS, BInCS, BOutCS, MP, MV\}$$

The set of synchronizations we want to apply on this system is:

$$Sync_{C,M} = \{AInCS//MP, AOutCS//MV, BInCS//MP, BOutCS//MV\}$$

Thirdly, we apply synchronization between the actions from systems C and M. In fact, both A and B interact with mutex M, the synchronizations are:

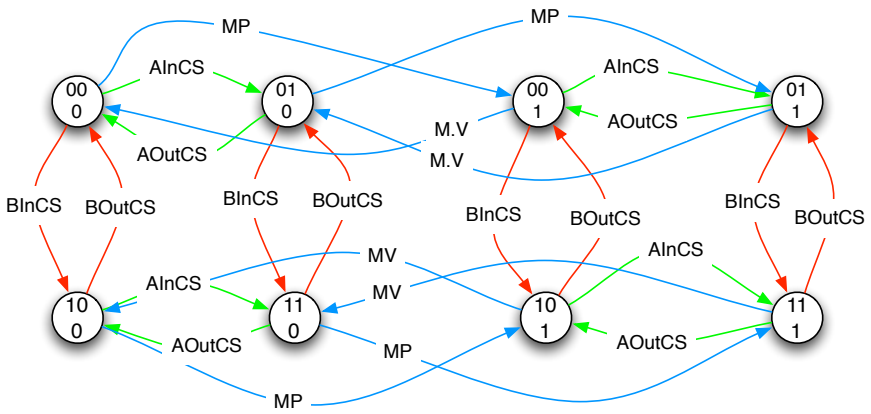


Figure 4.6: Free Product of Process $C \times_\phi M$

- For process A: $AInCS//MP, AOutCS//MV$
- For process B: $BInCS//MP, BOutCS//MV$

In order to obtain the composed system, we apply the rules in definition 4.2.6, concretely in the following steps †:

1. On system $C \times_\phi M$ shown in figure 4.6, remove transitions involved in synchronizations $Sync_{C,M}$, i.e. transitions labeled by $AInCS$, $AOutCS$, $BInCS$, $BOutCS$, MP, MV. In this example, all transitions are removed.

2. Starting with initial state, i.e. *000*, computing the firability and result state of applying actions: $AInCS//MP$, $AOutCS//MV$, $BInCS//MP$,$BOutCS//MV$. In this example, we discover the following transitions/states:

   - $000 \xrightarrow{AInCS//MP} 011$
   - $011 \xrightarrow{AOutCS//MV} 000$
   - $000 \xrightarrow{BInCS//MP} 101$
   - $101 \xrightarrow{BOutCS//MV} 000$

3. Repeat step 2 until no new states are discovered.

4. Remove all isolated states, i.e. states which are not connected with initial state. In this example, all states except $000, 011, 101$ are removed.

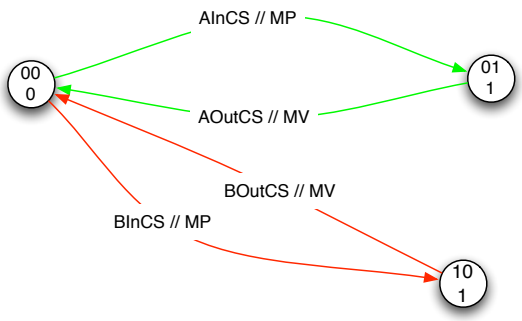


Figure 4.7: $C \times_{Sync_{C,M}} M$

The final synchronized composition of C and M, with

$Sync_{C,M} = \{AInCS//MP, AOutCS//MV, BInCS//MP, BOutCS//MV\}$

is shown in figure 4.7. The final system has 3 states and 4 actions, all other states are illegal states.

4.3.2 Synchronized Composition of Counters

In this example, we will synchronize two counters with different objectives using different synchronization expressions.

Case 1: No synchronization. As depicted on the left side of figure 4.8, system A and B are two counters to be composed.

We use SA and SB to refer to the state of counter A and B, respectively. The state of the composed system is expressed as the concatenation of SA and SB, e.g. 00 means SA is 0 and SB is 0, and so forth. If no synchronizations are specified, i.e. just put the two counters together, the composition of the two systems will have

- 4*4=16 states $\{00, 01, 02, 03, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33\}$, initial state 00.
- two actions (labels): $\{Ainc, Binc\}$
- 32 transitions: $\{4*At1, 4*At2, 4*At3, 4*At4, 4*Bt1, 4*Bt2, 4*Bt3, 4*Bt4\}$

as depicted in figure 4.8.

[†]Notice that different strategies may be used to deduce the result of composition. It is not always necessary to compute the free product before applying synchronizations. Here we use a simple approach by construction.

We use prefix A and B to distinguish the original system of actions and transitions, and labels on the arcs are just for illustration in order to show the differences before and after composition. $At1 = 00 \xrightarrow{Ainc} 10$, $At2 = 10 \xrightarrow{Ainc} 20$, $Bt1 = 00 \xrightarrow{Binc} 01$ ...

$Ainc$ and $Binc$ are actions can be performed in the new system, i.e. free product of system A and B shown on the right side of figure 4.8: all vertical arcs are labeled by $Ainc$ and all horizotal arcs are labeled by $Binc$.

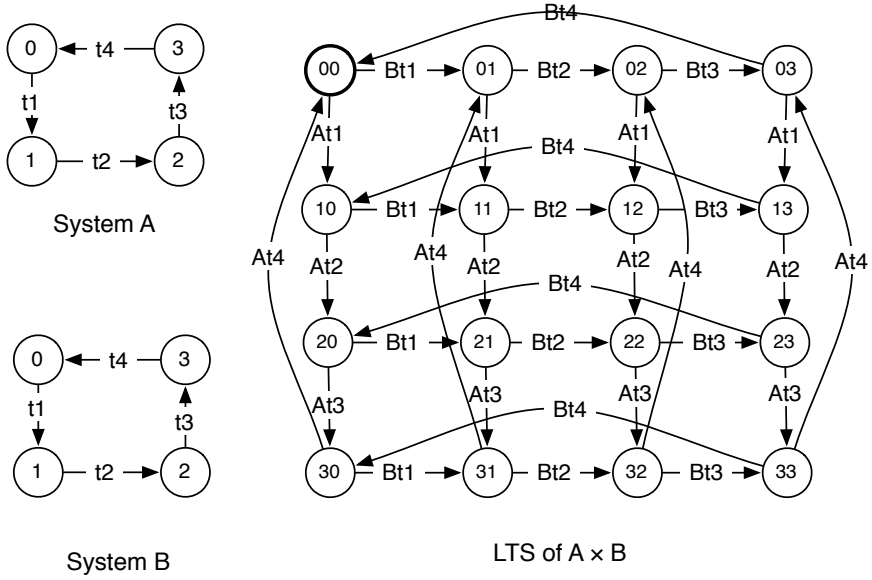


Figure 4.8: Free Product of A and B:$A \times_\phi B$

Now we want to synchronize two counters to get a bigger system. Depending on our intentions, there are several ways to define synchronization.

Case 2: Synchronize LTS transitions. We want the two counters to be synchronized at 0. In this case, when system A or B arrives at state 3, it has to wait another system to be able to finish this synchronization. This is similar to a blocking communication through networks.

Conditions of synchronizations In order to specify this synchronization, one way to identify a state from a set of states is to use *conditions*. We use the syntax $CondList :: Action$ to specify conditional actions. For example $SA = 3$ means system A is at state 3, thus $SA = 3 :: Ainc$ refers to $At4$. Our intention can be expressed by $SA = 3, SB = 3 :: Ainc//Binc$, where the comma ',' means the conjunction of conditions.

Using conditions is a way to dynamically classify (label) LTS transitions in complex systems. However, the syntax of the conditions depends the syntax of the concrete modeling language and we will not discuss details here.

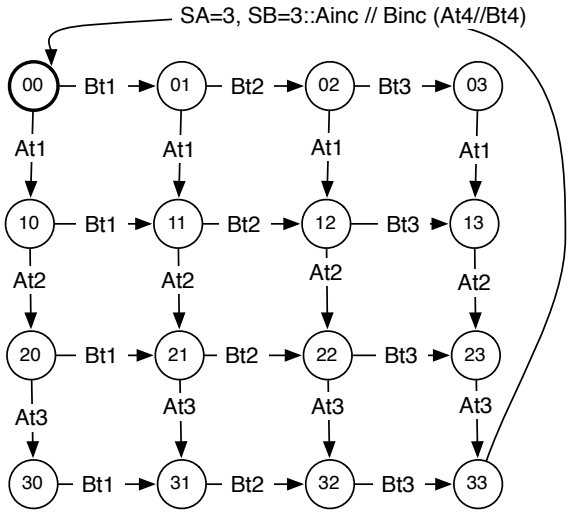


Figure 4.9: Synchronization with $SA = 3, SB = 3 :: Ainc//Binc$

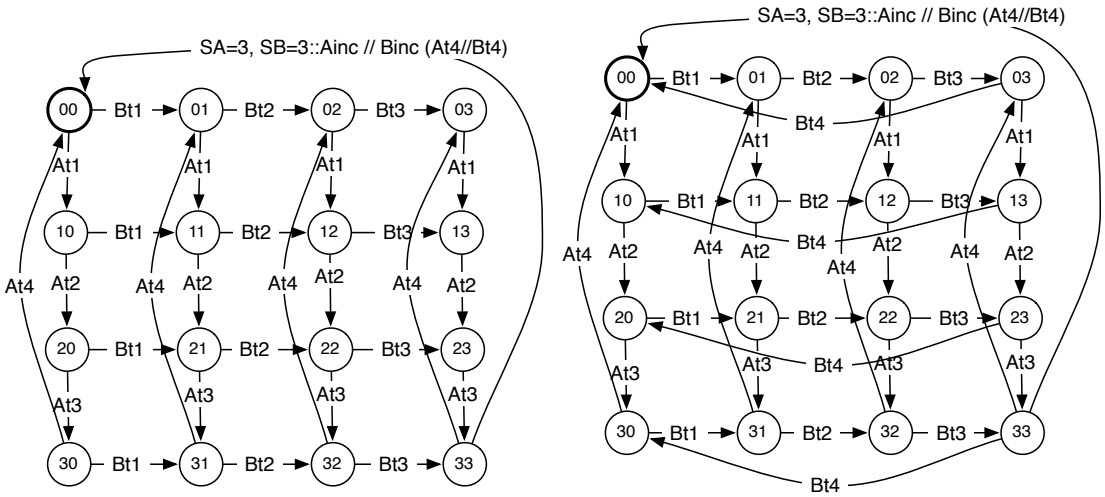
With synchronization expression $SA = 3, SB = 3 :: Ainc//Binc$, the composed system has:

- 16 states: $\{00, 01, 02, 03, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33\}$, initial state 00.
- 3 actions (labels): $\{Ainc, Binc, SA = 3, SB = 3 :: Ainc//Binc\}$
- 25 transitions $\{4 * At1, 4 * At2, 4 * At3, At4//Bt4, 4 * Bt1, 4 * Bt2, 4 * Bt3\}$

as shown in figure 4.9. The synchronized transition is labeled by $SA = 3, SB = 3 :: Ainc//Binc$, which means $At4$ and $Bt4$ occur simultaneously. Other possible synchronizations are (as shown in figure 4.10):

- $SA = 3, SB = 3 :: \widetilde{Ainc}//Binc$: we want counter A to be reset without considering counter B, but B has to synchronize with A while doing its reset.
- $SA = 3, SB = 3 :: \widetilde{Ainc}//\widetilde{Binc}$: we want two counters to run with and without synchronization

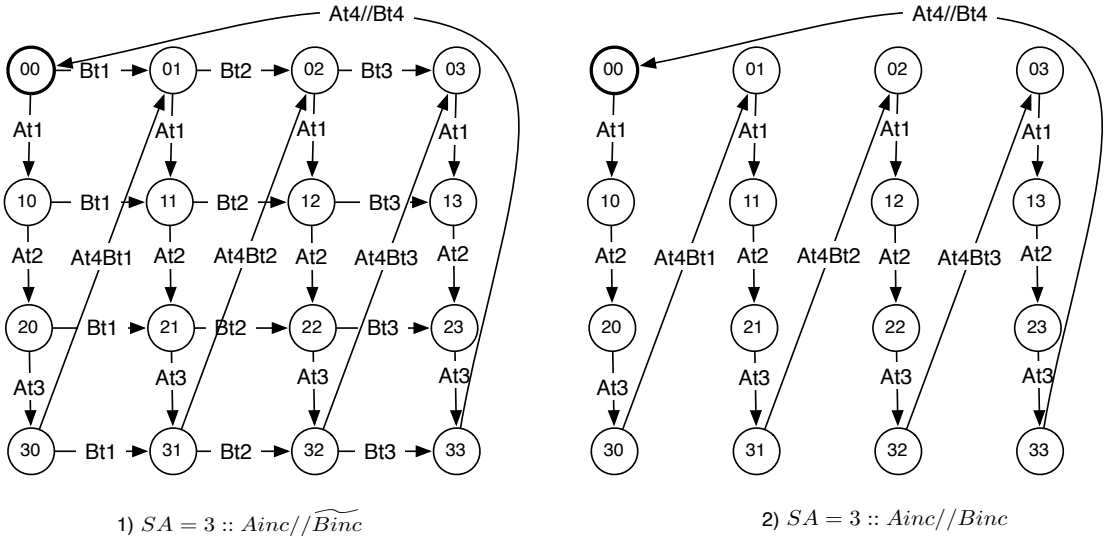
Case 3. We want to connect the two counters in a way that B increments when A finishes a cycle, that means $At4//\widetilde{Binc}, At4 \Rightarrow SA = 3 :: Ainc$. In this



1) $SA = 3, SB = 3 :: \widetilde{Ainc}//Binc$

2) $SA = 3, SB = 3 :: \widetilde{Ainc}//\widetilde{Binc}$

Figure 4.10: $SA = 3, SB = 3 :: \widetilde{Ainc}//Binc$ and $SA = 3, SB = 3 :: \widetilde{Ainc}//\widetilde{Binc}$



1) $SA = 3 :: Ainc//\widetilde{Binc}$

2) $SA = 3 :: Ainc//Binc$

Figure 4.11: Synchronization with $SA = 3 :: Ainc//\widetilde{Binc}$ and $SA = 3 :: Ainc//Binc$

case, transition $t4$ of system A is synchronized with all the transitions in system B labeled by *inc*. The composed system has:

- 16 states: $\{00, 01, 02, 03, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33\}$, initial state 00.
- 3 actions (labels): $\{Ainc, Binc, SA = 3 :: Ainc//Binc\}$
- 28 transitions $\{4*At1, 4*At2, 4*At3, At4//Bt1, At4//Bt2, At4//Bt3, At4//Bt4, 4 * Bt1, 4 * Bt2, 4 * Bt3\}$

as shown in figure 4.11.1. A and B can run separately and are synchronized each time when A moves from state 3 to state 0. A variance of this synchronization is $SA = 3 :: Ainc//Binc$ is shown in figure 4.11.2, where counter B is totally driven by counter A.

In this case, synchronizations are more dynamic than in case 2, i.e. they are interpreted during execution, e.g. $SA = 3 :: Ainc//Binc \Rightarrow \{At4//Bt1, At4//Bt2, At4//Bt3, At4//Bt4\}$. This is a *dynamic binding of synchronization with transitions*.

Case 4. We want to synchronize action *inc* of system A with action *inc* of system B. $Ainc//Binc$. In this case, all transitions labeled by *inc* in system A are synchronized with all the transitions labeled by *inc* in system B. With synchronization $Ainc//\widetilde{Binc}$, the composed system has:

- 16 states: $\{00, 01, 02, 03, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33\}$, initial state 00.
- 2 actions (labels): $\{Ainc//Binc, Binc\}$
- 28 transitions $\{At1//Bt1, At1//Bt2, At1//Bt3, At1//Bt4, At2//Bt1, At2//Bt2, At2//Bt3, At2//Bt4, At3//Bt1, At3//Bt2, At3//Bt3, At3//Bt4, At4//Bt1, At4//Bt2, At4//Bt3, At4//Bt4, 4 * Bt1, 4 * Bt2, 4 * Bt3\}$.

In this case, transitions labeled by $Ainc$ are removed and transitions labeled by $Binc$ are kept. The binding of synchronization is dynamic for both systems, e.g. $Ainc//Binc$ is bound with 16 transitions. Figure 4.12 depicts the results of synchronization with $Ainc//\widetilde{Binc}$, $\widetilde{Ainc}//\widetilde{Binc}$, and $Ainc//Binc$, respectively.

Impacts of synchronization on the properties of system. Synchronizations don't change the properties of the subsystems. However, based on the conjunction of its subsystems' states, the composed system has new properties, e.g.

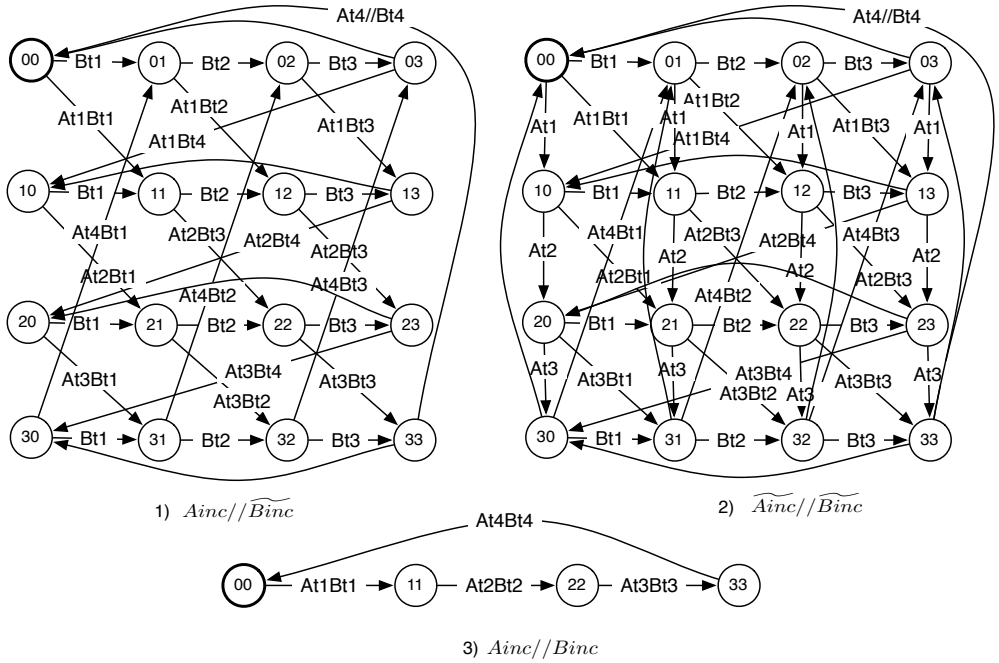


Figure 4.12: Synchronization with $Ainc//\widetilde{Binc}$, $\widetilde{Ainc}//\widetilde{Binc}$, and $Ainc//Binc$

in figure 4.12.3 a property could be: SA==SB. The properties of composed system rely on two kinds of elements: the *initial state of each subsystem* and *the synchronizations*.

While specifying a system, in general we know the initial states of subsystems. While implementing the system, synchronizations should be realized or implemented correctly, especially in parallel, distributed environments. For instance, in figure 4.13 a process in system C tries to send a message to process in system D, synchronization $C.send//D.receive$ should be implemented and ensured by the underlying communication system which may have different levels of reliability, security, and transactional supports. Algorithms such as 2-phase commit and 3-phase commit protocols are often used to guarantee the synchronization of multiple processes in distributed environment.

For instance, in figure 4.14 the transaction coordinator wants to synchronize the counters through a network. The implementation is a refinement of the synchronization expression, e.g. $Ainc//Binc//Xinc$. Based on existing infrastructures and requirements, 2-phase commit or 3-phase commit protocols can be used to implement the synchronization in a centralized way.

The separation of specification (or modeling) and implementation is a fundamental concept of system design. We divides a problem into *specification (of*

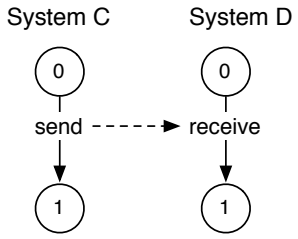


Figure 4.13: Two Distributed Communicating Systems

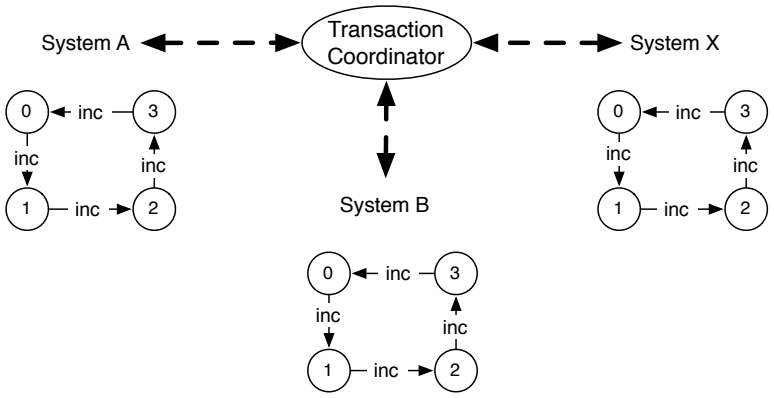


Figure 4.14: Use Transaction Coordinator for Synchronization

synchronization) and *implementation*. How to implement the synchronizations in complex environments is the next step of our work, which consists of automatic code generation from synchronization specifications and environment profiles. We classify the implementations of synchronization into 3 categories:

- communication between processes, e.g. locks, blocking send/receive, with ACK ...
- synchronization using a centralized transaction manager, e.g. 2-phase commit and 3-phase commit protocol
- decentralized peer-to-peer synchronization, e.g. broadcasting, p2p protocols.

Each kind of implementation is adequate for certain environments, and maybe not for others.

4.4 Petri Nets and LTS

Mapping from Petri net to LTS. The mapping from a PN to its LTS is straightforward by applying an injective function $L : T_{PN} \rightarrow A_{LTS}$, i.e. *for a given PN, the relation between the PN transitions and the labels of its LTS is one-to-one.* However, LTS of different models may be isomorphic, i.e. having the same graph structure but different labels. For example, the LTS of the Place/Transition net and the Algebraic Petri Net (APN) models in figure 4.15 are isomorphic to the LTS of the example counter in figure 4.2. Isomorphic LTS means that a system can be built using different syntax (labels). Since our synchronized composition of system is defined on LTS, it can be applied to any DSL whose semantics is given by LTS.

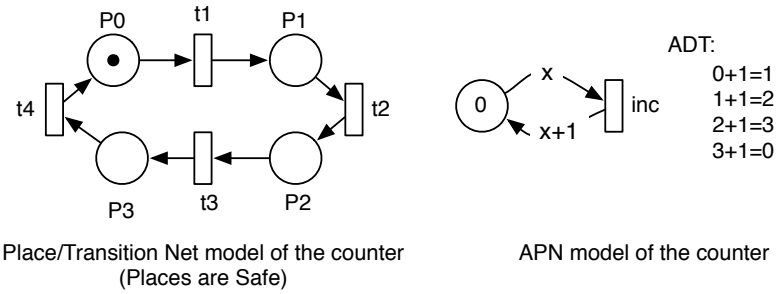


Figure 4.15: The Simple Counter Modeled by Place/Transition Net and APN

Moreover, in figure 4.15 the Place/Transition nets can be seen as the unfolded APN model. This is an example that the same semantic model is built through different languages.

Example of counter Figure 4.16 illustrates the compositions of the APN counter using different synchronization specifications. We can observe that transitions without $\widetilde{\ }$ are removed from the new system. Note that they are just for illustration purpose because we did not give the formal syntax composition rules with algebraic Petri nets. However, the syntactical composition of models has the objective of implementing the semantics of composition we presented in this Chapter [‡].

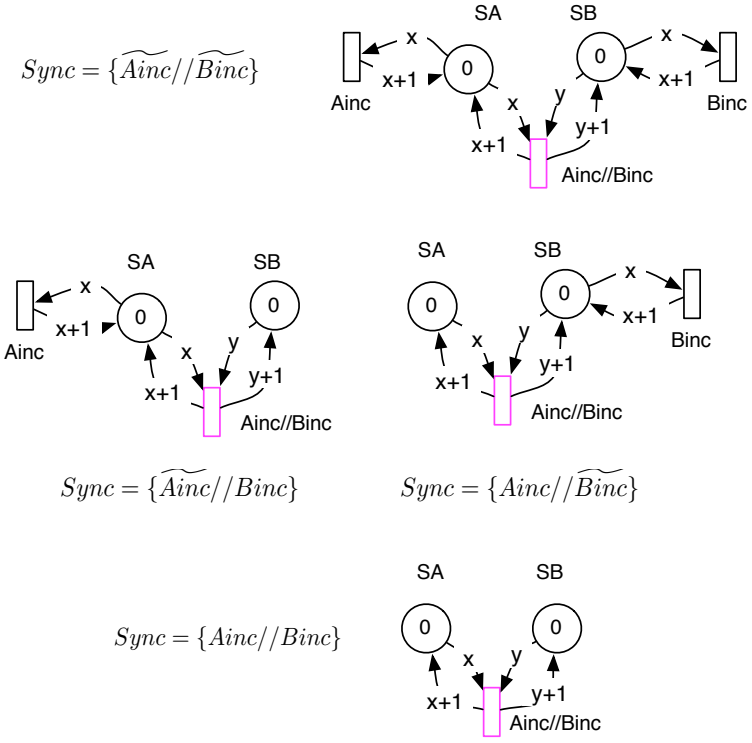


Figure 4.16: Different Compositions of two APN Counters A, B

Example of dining philosophers In dining philosophers problem, a certain number of philosophers spend their lives alternating between thinking and eating. They are seated around a circular table. There is a fork placed between each pair of neighboring philosophers. Each philosopher has access to the forks at her left and right. In order to eat, a philosopher must be in possession of both forks. Each philosopher attempts to pick up the left fork first and then the right fork. When finishes eating, a philosopher puts both forks back down on the table and begins thinking. Since the philosophers are sharing forks, it is not possible for all of them to be eating at the same time.

[‡]CO-OPN transactions can be considered as a syntactical composition of models, because they have the same semantics as our synchronizations.

The interactions between philosophers and forks can be seen as synchronizations between different systems. In our model, each philosopher has two states: *Eating* and *Thinking*, and two actions *GoEat* and *GoThink*. Each fork also has two states: *Free* and *Taken*, and two actions *Take* and *Put*. Figure 4.17 shows the PN model of two philosophers and two forks. Instead of assuming *a philosopher may only pick up one fork at a time*, we suppose *a philosopher can realize transactions*, i.e. a philosopher can take two forks at the same time.

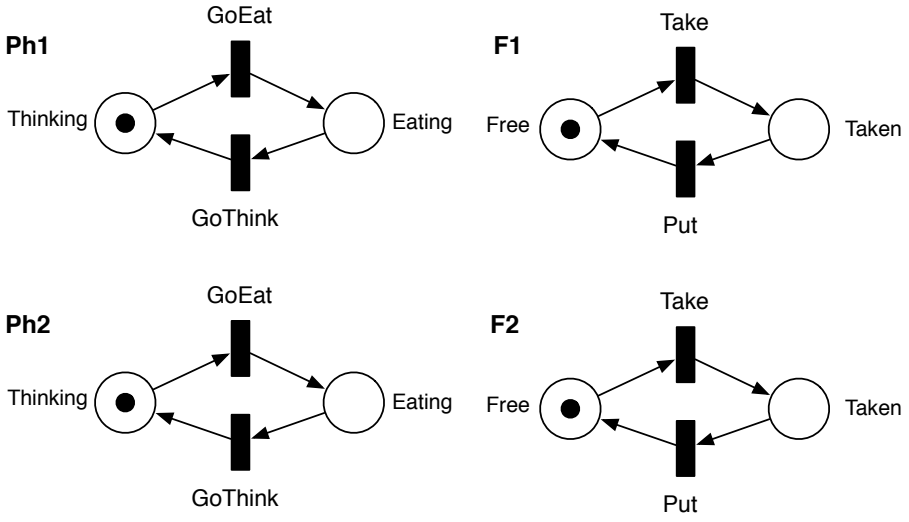


Figure 4.17: PN model of Two Philosophers and Two Forks

In order to compose the 4 systems using synchronizations, we start by identify labels (actions): Ph1.GoEat, Ph1.GoThink, Ph2.GoEat, Ph2.GoThink, F1.Take, F1.Put, F2.Take, F2.Put.

Then, we specify synchronizations (constraints) between the systems, in this example:

- Ph1.GoEat//F1.Take//F2.Take: Philosopher 1 will take fork 1 and fork 2 simultaneously when he or she performs GoEat.
- Ph1.GoThink//F1.Put//F2.Put: Philosopher 1 will put fork 1 and fork 2 simultaneously when he or she performs GoThink.
- Ph2.GoEat//F2.Take//F1.Take: Philosopher 2 will take fork 2 and fork 1 simultaneously when he or she performs GoEat.
- Ph2.GoThink//F2.Put//F1.Put: Philosopher 2 will put fork 2 and fork 1 simultaneously when he or she performs GoThink.

Figure 4.18 shows the composed system from above synchronizations. In fact, transitions synchronized with // are merged into a single transition in the composed system because modifier~is not used in the synchronizations. These synchronizations also impose that forks cannot change their states alone without being synchronized with philosophers.

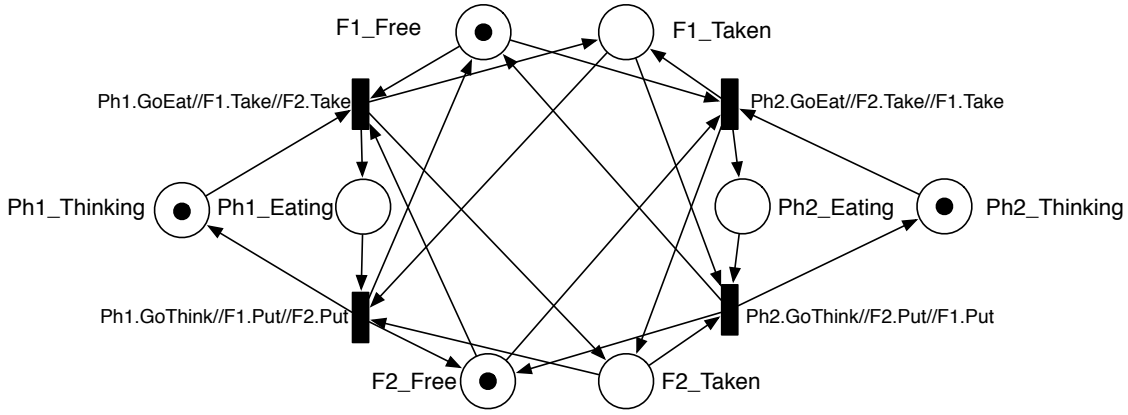


Figure 4.18: Synchronized Philosophers and Forks

4.5 Timing Semantics of the Synchronizations and the Sequence (..) Operator

In the formal definition of synchronized composition of LTS, synchronizations are parallel (//) or choice ($\oplus$). The composition of systems relies on their transactional semantics, i.e.

- for parallel synchronization, *all* synchronized actions/events should occur or *nothing* should occur.
- for choice synchronization, *only one* action/event or *nothing* will occur among all the synchronized actions/events.

However, in parallel synchronization the *timing (order)* of actions/events are unspecified, e.g. by $A//(B//C)$ we mean A, B, C should all occur during the synchronization (transaction), but no constraints are given regarding to the order of actions, such as:

- should actions A, B, C start simultaneously, or

- should actions A, B, C end simultaneously, or
- should they occur one after another with a more specific order, e.g. B after A, C after B etc.

Because **order** is an important factor in real world systems, *we re-introduce the sequence (..) operator as a refined parallel synchronization operator*. In fact, the sequence synchronization has the same semantics regarding to the composition of systems, but it implies ordering on the occurrence of events, e.g. for synchronization $A..B$, its semantics implies:

- the same semantics as parallel synchronization //: A and B should occur both, or none of them occurs (apparently), and
- more constraints on the order: B should be performed after A

A typical example of using the sequence is when several actions are synchronizing with the same actions, e.g., for two synchronizations $A//B$ and $A//C$, we can have several interpretations:

- $A//B, A//C \Rightarrow A//(B//C)$, without implying order on B and C
- $A//B, A//C \Rightarrow A//(B..C)$ or $A//B, A//C \Rightarrow A//(C..B)$, when we want to imply order on B and C for the synchronization

In our modeling work, the sequence synchronization .. can replace // when specific order is necessary for the parallel synchronization (or transaction) specified with //.

4.6 Summary

In this chapter, we propose a semantical composition language for the composition of labeled transition systems. Based on CO-OPN transactions, this language allows to express different intentions of composition, such as parallel and alternative synchronizations, and decide either a label/action of the initial system will appear in the composed system (i.e. visibility of labels) or not. The composition relies on the the transactional semantics of the operators: several things should *all* happen in different systems, or *nothing* should happen.

We have shown that, the composition of LTS is about constraining the free product of initial systems, thus reducing states from their free product. With a

well-designed strategy, it is possible to deduce the state space of composed system from the state space of initial systems, which will facilitates model-checking and verification of complex systems, thus managing the complexity.

In the next Chapter, we will present the formal notation, ID-net, which implicitly implies model composition via its annotation mechanisms.

Chapter 5

ID-Net

5.1 Motivation

Processes and *Resources* are two essential elements of computer systems. The design of computer system has the objectives of *optimizing resource utilization* and *maximizing process execution performance* while keeping the reliability of systems.

Processes are programs which will execute in parallel and compete for resources in concurrent environments. Created and deployed by software developers, the execution of processes is dynamic and depending on the execution environment, i.e current available resources in a system.

Resources can be CPU (cycles), memories, network communications, data objects, and users (interfaces). Resources can have fixed and limited numbers, or can be created and disposed dynamically.

The analysis of resource allocation in concurrent parallel environment is a classical problem and it is crucial for today's software systems. Properties such as liveness, deadlock-free, absence of starvation of processes, and fairness between concurrent processes are very important for the design of systems.

The development of concurrent, distributed system requires rigorously defined languages and models as well as automation tools which are capable to manage, verify the models, and transform them into executable codes. Formal methods and models such as process algebra [88–90], finite state machine, and Petri nets [91–93] are widely used in design and verification of distributed systems. Among them, Petri nets is the most often used model for concurrent and reactive systems in practical applications.

In Petri nets, *processes* are represented by the *net structure* and *resources*

are modeled (abstracted) as *tokens*. Petri nets' graphical notation and abundant verification techniques make it an excellent tool for the modeling and verification of concurrent systems. However, similar to process algebra and other formal approaches, classical Petri nets emphasize on process model and omit the details of resources. The abstraction of resource can reduce the complexity of validating concurrent systems specified in Petri nets, however, the details on resource management have to be considered during the development of system.

Software developers rely on programming languages they are using for the development. The tendency of using Domain-Specific Language (DSL) will allow domain experts to develop operational models/systems for specific kinds of problems. A well-designed DSL should use the vocabularies of a specific domain to easily express domain problems, and have formal semantics for executability and verifiability of models created by this language.

In fact, there are trade-offs between the verifiability (manage of complexity), expressive power, and executability of formal models. It would be interesting to bridge the semantics gap between formal models and domain-specific models by synchronizing their behaviors regardless of the modeling languages, i.e. giving formal semantics to the DSLs.

5.1.1 Principle of Petri Nets Modeling

There are many reasons or purposes of using Petri nets-based modeling languages, for example model validation, simulation, or development of concurrent systems. Different kinds of Petri nets are proposed based on these purposes, in general, a Petri nets-based modeling language should define (maybe not completely) the following elements:

- *Control structure*. Places, transitions, and the flow which connect them together. This is the common basis of Petri nets models. These elements form the essential constraints of the system.

- *Tokens*. Information or entities transported on the flow, they are consumed and produced by transitions.

- *Inscription language*. It is used to specify information on places, transitions, flow, and tokens.

- *State-Transition Semantics*. The rules of consuming and producing tokens with respect to the state of system; how transitions are enabled and executed.

Control Structure. Control structure defines control flow patterns such as sequence, parallel, selection (branching), and synchronization. The control structure of Petri nets shows parallel processing and synchronization explicitly and intuitively, which is an advantage compared to most general sequential programming languages, state-chart diagrams, and other automate-based graphical languages. It allows to specify *non sequential processes* [94], which can have several control points simultaneously.

Tokens. Tokens can represent physical or conceptual entities, states of logical predicates, or pieces of information. The interpretation of the presence of a token in a particular place depends on the meaning of token and the intention of modeler. The state of a system, i.e. the marking, is represented by the distribution of tokens in the control structure and the values of tokens.

A Petri net can use distinguishable (individual token) or non-distinguishable tokens (black tokens). Non-distinguishable tokens don't have values, and a marking of a place is the number of tokens in the place. A marking of nets with distinguishable tokens also depends on the nature of its tokens, e.g. values and identities of tokens. Distinguishable tokens can be typed or not (if only one type is used). Typed token can be categorized as follows:

- value token. For example *0, 1, 2, true, false*. Value tokens are distinguished by their values.
- ID tokens: simple-variable tokens. Equivalent to names of variables of primitive types, e.g. *integer a, string b, float c*, *a,b,c* are simple typed variable tokens.
- ID tokens: reference tokens. Equivalent to references to structured objects, e.g. a *book* object with attributes such as *name, ISBN, author etc.*

Value tokens are distinguished by their values. ID tokens are distinguished by their identities. A collection of tokens can be managed by using an ID token referring to a list (or bag, set) structure containing other tokens. Normally, value tokens are self-contained and their values can be directly used by transitions, while additional mechanisms is needed to solve and manipulate the values/resources associated with ID tokens. However, with value tokens, the manipulate rules have to be given somewhere, e.g. implicit rules or defined by the inscription language. With ID tokens, the rules of using the IDs have to be defined explicitly in models outside of ID-net (i.e. co-models), as we will present later in this chapter.

Inscription Language. An inscription language is a Domain-Specific Language. Inscription language defines allowed annotations on the control structure. These annotations can be used for automatic model processing such as analysis and model transformation. Most of the time, tokens are also defined by an inscription language. Several kinds of inscriptions are used in literature:

- Place inscription: name and type of places
- Transition inscription: name, conditions, operations
- Arc inscription: variables, weights on arcs
- Tokens: types and values of tokens
- Annotations of nets such as name etc.

Inscription languages play an important role for Petri nets-based modeling languages. Since control structure represents control-flow with respect to causality of events, inscription languages are used principally to define data-related aspects such as conditions and data manipulations. In fact, inscriptions define a system side-by-side with the control structure, i.e. it is a Domain-Specific Model (DSM). Inscriptions specify interactions between control structure and the DSM. If the inscription language is a pure functional language, we don't need to consider the state space of the DSM.

An inscription language may have the following parts:

- Naming rules for place and transition
- Data type: primitive types, complex data type definition, and variable declaration
- Conditional expressions (guards)
- Function definition, with or without side-effects

The execution of transition inscription should be *atomic*. Firing the transition and executing of the inscription form a transaction, i.e. both finish successfully, or nothing changes in the state of the system in case of failure.

Semantics and Execution of Petri net. A semantics of Petri nets gives the enabling and execution rules of transitions with respect to state-transition systems, e.g. how to compute the next state of system from actual state, how variables are evaluated and bound to values.

Nondeterministic sequential interleaving semantics An operational semantics of Petri nets should specify the step-by-step evolution of system and how these steps can be achieved operationally, i.e. how to execute Petri nets. For example, a Petri net can be executed with **nondeterministic sequential interleaving semantics** by the following steps:

1. establishing an initial marking,
2. choosing a set of eligible transitions,
3. **firing a transition** among the set of eligible ones (nondeterministic),
4. going back to step 2 until no more transition is eligible.

An eligible or enabled transition is a transition which has enough resource to be fired. The step **firing a transition** can have different operational semantics. From the resource perspective of view, a possible operational semantics can be described informally as follows:

1. select tokens in pre-set places,
2. reserve pre-condition tokens temporally,
3. **evaluate T**, if fails then releases reserved tokens, and go back to 1,
4. remove reserved tokens, i.e. consume tokens,
5. put post-condition tokens into post-set places, i.e. produce tokens.

The step **Evaluate T** consists of evaluating guard conditions of T, and performing side-effects-free actions. The execution of this sequence should be *atomic*. This operational semantics also implies the following operations on the places (as we did for prototyping CO-OPN specifications in [95] and A.1):

- P.take(token): reserve token
- P.release(token): release a token (inverse of take)
- P.put(token): put a token into the place
- P.remove(token): remove a token from the place

True concurrency semantics A Petri net can be also executed with **true concurrency semantics** [64,65], which means all enabled transitions can be fired simultaneously:

1. establish an initial marking,
2. **fire all eligible transitions as possible**,
3. repeat step 2 until no more transition is eligible.

With true concurrency semantics, the executions of transitions are concurrent. The step of **firing all eligible transitions** is non-determinist and may imply conflicts on accessing resources during the evolution of system, i.e. several transitions requiring the same resource are enabled at the same time, but they cannot be executed simultaneously. In this case, the execution of transitions depends on the way of accessing the resources and can become interleaved.

5.2 Objectives of ID-Net

Using *true concurrency semantics*, ID-net is designed for the development of concurrent, distributed systems. Transitions can run in parallel if enough resources are available, in case of conflicts on resources, the execution of transitions become nondeterministic sequential interleaving.

Our modeling framework extends the elements mentioned in precedent section by adding a *synchronization mechanism* allowing ID-net to interact with heterogenous external models at the same time. Each external model, also called the **co-model of ID-net**, can be defined using a domain-specific language. Notice that notion of *co-model* is relative, an ID-net can be a co-model of another ID-net and the inverse can be also true at the same time.

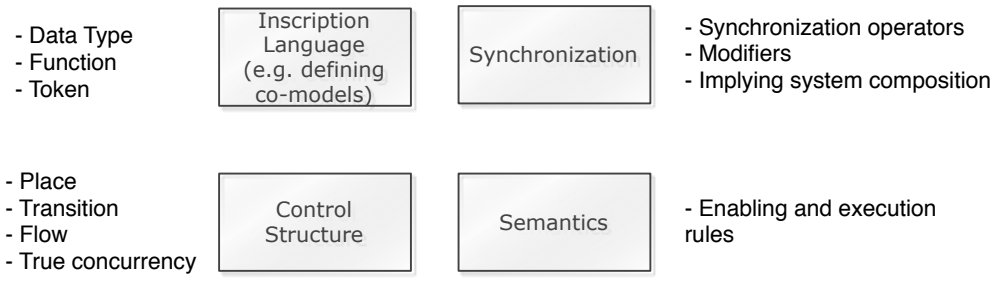


Figure 5.1: Elements of ID-net Modeling Framework

A fundamental of ID-net is the separation between *how to organize resources for different purposes (processes)* and *how the resources can be used and managed*. They represent the two dimensions of a system:

- the *control structure* of processes, i.e. modeled by ID-net, specifies how to organize resources to achieve objectives. This part is supposed to be executed in parallel as possible.
- how a specific kind of resource can be used and managed depend on the nature of the resources, and it can be specified by a domain-specific language. For example, a function can be called concurrently or not, a variable can be read and write but not both at the same time etc.

The interactions between the two dimensions can be specified by synchronizations. From Aspect-Oriented Development (AOD) point of view, each kind of resource may correspond to a specific concern of a system and ID-net specifies how the concerns can be organized to achieve objectives. Thus this approach allows to model and develop large, complex systems.

Incorporate domain-specific languages ID-net can be annotated by different Domain-Specific Languages at the same time, ID-net model coordinates these domain-specific models by executing the synchronizations. The final model will be a composition of ID-net and other domain-specific models (co-models).

Concurrent programming Unlike parallel computing models which concentrate on data parallelism and performance issues, concurrent programming aims to to build reliable systems and manage their complexity. Concurrent programming is a challenging domain in today's computing environment. The true semantics will allow ID-net to be executed in parallel and concurrent environments: the co-models manage the resources and synchronize with ID-net by communicating values and the identifications of the resources.

Incremental development The annotation of ID-net can be performed incrementally by iterations.

5.3 ID-Net: Syntax and Semantics

In this section, we will give the formal definition of ID-net and its operational semantics. First of all, the operational semantics without synchronization is given.

Then we discuss how the synchronized compositions are interpreted during the execution of ID-net model, i.e. how the semantics of external model and ID-net model are composed.

Unlike classical Petri net, i.e. Place/Transition net, which uses undistinguishable tokens, ID-net uses distinguishable identifications as tokens. In Place/Transition net, places contains integers which count the available resources in a system, and transitions increase/decrease the counters during execution. With ID-net, we are able to know which resources are available and which resources will be used at each step.

Because ID-net can represent the control structure of parallel program running in concurrent environment, the two kinds of elements place and transition have different roles in operational semantics of ID-net:

- Places are passive elements which hold and give out resources upon request.
- Transitions are active and dynamic elements. A transition can access its surrounding places.
- Places and transitions are connected by arcs. A transition **requests** resources or information from its *pre-set places*, and it **must give** resources or information to its *post-set places*.
- Tokens in a place are accessed concurrently by connected transitions, thus place has to manage these access.
- A transition performs the sequence of actions as a transaction: consume tokens, perform inscription function, and produce tokens.

5.3.1 Definitions

Except explicitly mentioned, the term *"Petri net"* means *Place/Transition net* in our discussion.

Definition 5.3.1. ***Petri net****. A* ***Petri net*** *is a tuple* $< P, T, F, W >$*, where*

- P *is a finite set of* places*,*
- T *is a finite set of* transitions $(P \cap T = \emptyset)$*,*
- $F \subseteq F^{in} \cup F^{out}$ *is the* flow relation*.* $F^{in} = P \times T$*,* $F^{out} = T \times P$*,*
- $W : F \to (\mathbb{N} \backslash \{0\})$ *is the* arc weight *mapping.*

Let $a \in P \cup T$. The set $\bullet a = \{a' | < a', a > \in F\}$ is called the *pre-set* of a, and the set $a\bullet = \{a' | < a, a' > \in F\}$ is its *post-set*. Each transition *consumes* tokens from its pre-set places and *produces* tokens to its post-set places.

Definition 5.3.2. ***Marking of a Petri net.*** *Given a Petri net $N =< P, T, F, W >$, the marking $M : P \to \mathbb{N}$ is the distribution of tokens in its places P.*

Places hold the *tokens*. F defines the possible token flow. The semantics of Petri net is defined as a transition system. A *state* is a Petri net *marking* $M \in P \to \mathbb{N}$. The state of a net changes when transitions *fire*. For a transition $t \in T$ to fire it has to be *enabled*. If a transition fires, it *consumes* tokens from places in $\bullet t$ and *produces* tokens in places in $t\bullet$.

Definition 5.3.3. ***Petri net with marking***. *A Petri net with marking is a tuple $< N, m^0 >$ where:*

- *$N =< P, T, F, W >$ is a Petri net*
- *m^0 is one marking, called the initial marking.*

ID-net takes a step forward by using typed and distinguishable tokens, i.e. identifications, which give minimum information about the resources.

Definition 5.3.4. ***ID-net***. *An ID-net is a tuple $< D, V, P, T, F, Ains, Tcond >$:*

- D *is a finite set of domain names (or type names)*
- *V is a finite set of variable names. $type : V \to D$ gives the type of a variable*
- P *is a finite set of* places. *$type : P \to D$ gives the type of a place*
- T *is a finite set of* transitions, *$(P \cap T = \emptyset)$*
- *$F \subseteq (F^{in} \cup F^{out})$ is the* flow relation. *$F^{in} = P \times T$, $F^{out} = T \times P$, each flow is a pair as $< p, t >$ or $< t, p >$.*
- *$Ains \in [F \to \mathcal{P}(V)]$ is an arc inscription function. An arc inscription is a set of variable names inscribed on an arc, e.g. $Ains(< p, t >)$ is the set of variable names inscribed on arc from a place to a transition $f =< p, t >$; $Ains(< t, p >)$ is the set of variable names inscribed on arc from a transition to a place $f =< t, p >$*

- *type* : $F \to D$ *gives the type of an arc, i.e. for any arc* $f \in F$,

$$\forall v \in Ains(f) : type(v) = type(f)$$

where $f =< t, p >$ *or* $f =< p, t >$,

$$type(< t, p >) = type(p), type(< p, t >) = type(p)$$

- $Tcond$ *is a function* $T \to \mathcal{P}(Cond_V)$, *where* $Cond_V$ *is a set of conditions on variables from* V, *a condition can be one of the following forms:*
  - $\forall v, v' \in V$, $(v \neq v') \in Cond_V$, *meaning* v *and* v' ***must*** *refer to different tokens*
  - $\forall v, v' \in V$, $(v == v') \in Cond_V$, *meaning* v *and* v' ***must*** *refer to the same token*

Example. Domain names can be *value* or *reference* types, e.g. $D = \{int, bool, char, int*, bool*\}$ meaning D has the value types of *int, bool, char*, and the identification (reference) types of *int* and *bool*.

A **domain** is the set of values of the same type, e.g. $domain(int) = \{0, 1, 2...\}$, $domain(boolean) = \{true, false\}$. $domain(int*) = \{x, y, z\}$ means x, y, z are reference type (variable name) of *integer*, where **domain** is a function which return the set of value for that domain name.

An *identification* is unique in its *domain*. A token can refer to a concrete computing resource or an abstract entity. A concrete resource is *accessible* for those who holds its token, for instance, in a Web application, the client (browser) will get a session ID after the user authentication step and afterwards the session ID should be provided each time in order to access privileged resources.

However, an identification token is not necessary to be unique in an ID-net, i.e. multiple tokens can refer to the same resource or value. In a real execution environment, resources are global and transitions trying to acquire the same resource are coordinated by resource managers or concurrent data structures. This implies the composition of ID-net and these models.

ID-net places and arcs are typed. Each ID-net arc can be annotated by a local variable name, the variable will be bound with the ID token while firing the transition connected with the arc, and the variable is accessible locally by the transition.

An ID-net transition controls which tokens it consumes and produces. $Tcond$ is a set of supplementary conditions on the variables of V to limit the flow of tokens.

Definition 5.3.5. ***Transition Inscriptions***. *For ID-net* $N =< D, V, P, T, F, Ains, Tcond >$, $Tins \in [T \rightarrow \mathcal{P}(V)]$ *is a transition inscription function. A transition inscription is a set of input and output variables names inscribed on its input arcs and output arcs, respectively.*

$$\forall t \in T : Tins_{in}(t) = \bigcup_{p \in \bullet t} Ains(< p, t >)$$

$$\forall t \in T : Tins_{out}(t) = \bigcup_{p \in t \bullet} Ains(< t, p >)$$

$$Tins(t) = Tins_{in}(t) \cup Tins_{out}(t)$$

Syntactically, the scope of a variable name is limited to a transition. Thus variable names will be interpreted locally by a transition and we have the following *implicit rules* for the local interpretation of variable names:

- in the scope of a transition, the same variable name **always** refer to the same token.
- in the scope of a transition, different variable names **do not need** but **may** refer to the same token.

To be enabled, a transition tries to bind tokens from its input places to the inscribed variables of its input arcs, respectively. When the same variable name is inscribed on more than one input arcs of a transition, a *strict join* is formed. On the other hand, a *duplication transition* is a transition has more than one output arcs having the same inscription variable. Figure 5.2 shows strict join and duplication transitions.

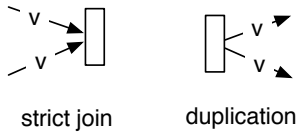


Figure 5.2: Strict Join and Duplication ID-net Transitions

Definition 5.3.6. ***Strict Join Transition***. *For ID-net* $N =< D, V, P, T, F, Ains, Tcond >$, *transition* $t \in T$ *is called a strict join transition iff:*

$$\exists p, q \in \bullet t, p \neq q, Ains(< p, t >) \cap Ains(< q, t >) \neq \emptyset$$

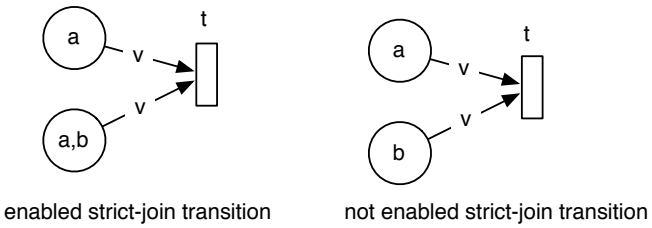


Figure 5.3: Examples of Strict Join Transitions

A strict join transition is enabled only if the same token is bound with the same variable of different arcs. For example, in figure 5.3, transition t is not enabled if it cannot find two tokens with the same identity in its pre-places.

Definition 5.3.7. ***Duplication Transition***. *For ID-net* $N =< D, V, P, T, F, Ains, Tcond >$, *transition* $t \in T$ *is called a duplication transition iff:*

$$\exists p, q \in t\bullet, p \neq q, Ains(< p, t >) \bigcap Ains(< q, t >) \neq \emptyset$$

A *duplication transition* simply send several copies of the same token to its post-places.

Definition 5.3.8. ***Pipe Transition.*** *For ID-net* $N =< D, V, P, T, F, Ains, Tcond >$, *transition* $t \in T$ *is called a pipe transition iff:*

$$Tins_{in}(t) = Tins_{out}(t)$$

A *generator transition* is an ID-net transition which pass input tokens to its output places. It is like a 'pipe' which don't modify the content of token flow.

Definition 5.3.9. ***Generator Transition.*** *For ID-net* $N =< D, V, P, T, F, Ains, Tcond >$, *transition* $t \in T$ *is called a generator transition iff:*

$$\exists v : v \notin Tins_{in}(t) \text{ and } v \in Tins_{out}(t)$$

A *generator transition* is an ID-net transition which uses different output variable names than input variable names. As shown in figure 5.4, variable name $v2$ is inscribed on the output arcs and not on the input arcs of transition, thus the transitions in figure 5.4 are generator transitions.

Unlike normal transitions which forward input tokens without changing them, generator transitions have the rights and responsibility to bind the new output variables, i.e. they may produce new tokens than the input tokens.

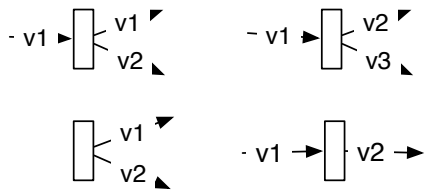


Figure 5.4: Examples of Generator Transitions

5.3.2 Operational Semantics of ID-net

The followed notations are used to describe the operational semantics of ID-net.

Definition 5.3.10. ***Multi-Set.*** *A multi-set m, over a non-empty set S, is a function $m \in [S \to \mathbb{N}]$, the positive integer $m(s) \in \mathbb{N}$ is the number of appearances of the elements s in the multi-set m.*

Notation. We usually represent the multi-set m by a formal sum:

$$m = \sum_{s \in S} m(s)'s$$

By $\mathcal{M}(S)$ or $\mathcal{M}_S$ we denote the set of all multi-sets over S, then $m \in \mathcal{M}(S)$. The positive integers $\{m(s)|s \in S\}$ are called the **coefficients** of the multi-set m, and $m(s)$ is called the **coefficient** of s. An element $s \in S$ is said to **belong** to the multi-set m iff $m(s) \neq 0$ and we then write $s \in m$.

Definition 5.3.11. ***Operations of Multi-Sets.*** *We define the following operations of multi-set: singleton, addition, comparison, size, and subtraction. For all $m, m_1, m2 \in \mathcal{M}(S)$:*

- *Empty multi-set. An empty multi-set over S it is a function $\emptyset_{\mathcal{M}}$ such as $\forall s \in S, m(s) = 0$*
- *Singleton. This function creates a multi-set from a single element using $\{\}_{\mathcal{M}}$,*

  $$s1 \in S, \{s1\} = \sum_{s \in S} m(s)'s$$

  where if $s == s1 : m(s) = 1$, otherwise $m(s) = 0$.
- *Addition $+_{\mathcal{M}}$:*

  $$m_1 +_{\mathcal{M}} m_2 = \sum_{s \in S} (m_1(s) +_{\mathbb{N}} m_2(s))'s$$

- *Comparison* $\neq_{\mathcal{M}}$*:*

$$m_1 \neq_{\mathcal{M}} m_2 = \exists s \in S : m_1(s) \neq_{\mathbb{N}} m_2(s)$$

- *Size* $|\ |$*:*

$$|m| = \sum_{s \in S} m(s)$$

- *Comparison (sub multi-set)* $\leq_{\mathcal{M}}$*:*

$$m_1 \leq_{\mathcal{M}} m_2 = \forall s \in S : m_1(s) \leq_{\mathbb{N}} m_2(s)$$

- *Substraction* $-_{\mathcal{M}}$*. when* $m_1 \leq_{\mathcal{M}} m_2$ *:*

$$m_2 -_{\mathcal{M}} m_1 = \sum_{s \in S} (m_2(s) -_{\mathbb{N}} m_1(s))'s$$

$\geq_{\mathcal{M}}$ *and* $=_{\mathcal{M}}$ *are defined analogously to* $\leq_{\mathcal{M}}$*.*

Proposition 5.3.12. *The following properties are satisfied for all multi-sets* $m_1, m_2, m_3 \in \mathcal{M}$ *and all non-negative integers* $n, n_1, n_2 \in \mathbb{N}$*:*

- $+_{\mathcal{M}}$ *is commutative.* $m_1 +_{\mathcal{M}} m_2 = m_2 +_{\mathcal{M}} m_1$
- $+_{\mathcal{M}}$ *is associative.* $m_1 +_{\mathcal{M}} (m_2 +_{\mathcal{M}} m_3) = (m_1 +_{\mathcal{M}} m_2) +_{\mathcal{M}} m_3$
- $\emptyset$ *is zero element.* $m +_{\mathcal{M}} \emptyset = m$
- $|m_1 +_{\mathcal{M}} m_2| = |m_1| +_{\mathbb{N}} |m_2|$

Definition 5.3.13. ***Binding Function****. A binding function is a mapping from variable names to the values in their corresponding domains. For a set of variable names* V *and their domain names* D*,*

$$Bind : V \rightarrow \bigcup_{d_i \in D} domain(d_i) \cup \{\perp\}$$

where

$$\forall v \in V, Bind(v) \in domain(type(v))$$

$Bind(v) = \perp$ means we do not give a binding for variable v, i.e. invalid (or undefined) binding. Except specifically mentioned in our discussion, a binding should be valid.

Definition 5.3.14. ***Extended Binding Function (overloaded Bind)****. An extended binding function binds a set of variable names to a multi-set of values. For a set of variable names V and a set of domain names D,*

$$Bind : \mathcal{P}(V) \rightarrow \mathcal{M}(\bigcup_{d_i \in D} domain(d_i) \cup \{\perp\})$$

where

$$\forall v \in V, Bind(v) \in domain(type(v))$$

$Bind(V)$ can be computed using $Bind(v)$ as follows:

- $Bind(\emptyset) = \emptyset_{\mathcal{M}}$
- $Bind(\{u\} \cup V) = \{Bind(u)\}_{\mathcal{M}} +_{\mathcal{M}} Bind(V)$

Marking. Let m be a marking of an ID-net. m is a function from P to the multi-set of values of $domain(d)$, $d \in D$:

$$m : P \rightarrow (\bigcup_{d_i \in D} \mathcal{M}(domain(d_i)))$$

A marking of place p, $m(p)$, is a multi-set of values from the domain of $type(p)$:

$$\forall p \in P, type(p) = d, m(p) \in \mathcal{M}(domain(type(p)))$$

Similarly to Petri net, $< N, m^0 >$ denotes an ID-net N with initial marking m^0.

Enabled transition. A transition $t \in T$ is m-enabled with a binding, written as $m \xrightarrow{<t,Bind>}$, iff

$$\forall p \in \bullet t : Bind(Ains(< p, t >)) \leq_{\mathcal{M}} m(p)$$

The set of *m-enabled* transitions is

$$T^m = \{t \in T \mid m \xrightarrow{<t,Bind>}\}$$

Here the $Bind$ function is a resource allocation mechanism, i.e. it reserves tokens for the variables at the level of transition.

Firing an ID-net transition. An m-enabled transition t *may* fire, producing the successor marking m', written as $m \xrightarrow{<t,Bind)>} m'$, where $Bind$ is a function which apply function $Bind$ on all variables in $Tins(t)$, thus:

$$\forall p \in P : m'(p) = m(p) -_{\mathcal{M}} Bind(Ains(p, t)) +_{\mathcal{M}} Bind(Ains(t, p))$$

Because ID-net uses true concurrency semantics, enabled transitions can fire simultaneously if there are no conflict between them. If conflict exists between simultaneously enabled transitions, the choice of firing transitions is non-deterministic and the execution becomes sequential interleaved.

Definition 5.3.15. ***Reachable markings of Petri net.*** *For Petri net PN with initial marking $m^0 < PN, m^0 >$, the set of markings reachable from m^0, i.e. the reachability set of m^0, $reach(PN, m^0)$ is the smallest set of markings, such that:*

- $m^0 \in reach(PN, m^0)$, *and*
- *if* $m' \xrightarrow{t} m''$ *for some* $t \in T$, $m' \in reach(PN, m^0)$, *then* $m'' \in reach(PN, m^0)$

Definition 5.3.16. ***Reachable markings of ID-net.*** *For ID-net N with initial marking $m^0 < N, m^0 >$, the set of markings reachable from m^0, i.e. the reachability set of m^0, $reach(N, m^0)$ is the smallest set of markings, such that:*

- $m^0 \in reach(N, m^0)$, *and*
- *if* $m' \xrightarrow{<t,Bind>} m$ *for some* $t \in T$, $m' \in reach(N, m^0)$, *then* $m'' \in reach(N, m^0)$

Definition 5.3.17. ***LTS of ID-net without checking Tcond***
For an ID-net with initial marking $< N, m^0 >$, its LTS is $\mathcal{A}_{<N,m^0>} =< S, A, T_{\mathcal{A}}, s^0 >$, where

- $s^0 = m^0$
- $S = reach(N, m^0)$
- $A = T$
- $T_{\mathcal{A}} = \{m' \xrightarrow{<t,Bind>} m'' \mid t \in T, m', m'' \in reach(N, m^0)\}$

Definition 5.3.18. ***Path in LTS***. *In a LTS $< S, A, T, s^0 >$, a path from marking m^0 to m is a sequence of LTS transitions, i.e.*

$$m^0 \xrightarrow{<t_0,Bind^0>} m^1 \xrightarrow{<t_1,Bind^1)>} m^2 ... \xrightarrow{<t_n,Bind^n)>} m$$

where $t_0, t_1 ... t_n \in T$, the sequence is:

$$seq =< t_0, Bind^0 >, < t_1, Bind^1 > ... < t_n, Bind^n >$$

The path from m^0 to m can be denoted as

$$m^0 \xrightarrow{<t_0,Bind^0>,<t_1,Bind^1>...<t_n,Bind^n>} m$$

or simply as

$$m^0 \xrightarrow{seq} m$$

A pair of transition/binding $< t_0, Bind^0 >$ *is in sequence seq is denoted as:* $< t_0, Bind^0 > \in seq$. *There may exist zero or more paths from one marking to another marking. We define function* $Path(\mathcal{A}, m^0, m)$ *as the set of paths from marking* m^0 *to* m *in LTS* $\mathcal{A}$, *thus:* $seq \in Path(\mathcal{A}, m^0, m)$.

Considering constraints Tcond in ID-net's operational semantics. The above definitions do not check constraints defined in $Tcond$ which will limit the bindings.

Definition 5.3.19. ***Binding satisfying Tcond.***
In a LTS of ID-net, for a set of conditions $Tcond$ *and a binding on a set of variable* V *at marking* m, *i.e.* $Bind^m_V$:

- $Bind^m_V \models \emptyset$
- $Bind^m_V \models (Tcond \cup \{v1 == v2\}, v1, v2 \in V) \Leftrightarrow$
  $(Bind^m_V \models Tcond) \wedge (Bind^m(v1) == Bind^m(v2))$
- $Bind^m_V \models (Tcond \cup \{v1 \neq v2\}, v1, v2 \in V) \Leftrightarrow$
  $(Bind^m_V >\models Tcond) \wedge (Bind^m(v1) \neq Bind^m(v2))$

denoted as $Bind^m_V \models Tcond$.

Definition 5.3.20. ***LTS of ID-net satisfying Tcond***
For ID-net $N =< D, V, P, T, F, Ains, Tcond >$ *and its initial marking* m^0, *its LTS without checking* $Tcond$ *is* $\mathcal{A}_N =< S_\mathcal{A}, A, T_\mathcal{A}, s^0 >$, *its LTS satisfying* $Tcond$ *is* $\mathcal{B}_N =< S_\mathcal{B}, A, T_\mathcal{B}, s^0 >$:

- $S_B \subseteq S_A$
- $A = T$
- $T_\mathcal{B} = \{m' \xrightarrow{t,Bind} m'' \mid t \in T, m', m'' \in S_\mathcal{A}, Bind \models Tcond\} \subseteq T_\mathcal{A}$

Enabled transition satisfying $Tcond$***.*** *A transition* $t \in T$ *is* m*-enabled with a binding which satisfies* $Tcond$, *written as* $m \xrightarrow{<t,Bind_{Tcond}>}$, *iff*

$$m \xrightarrow{<t,Bind>} \wedge\ Bind \models Tcond$$

The set of **m-enabled** ***transitions satisfying*** $Tcond$ *is*

$$T^m_{Tcond} = \{t \in T \mid m \xrightarrow{<t,Bind_{Tcond}>}\} \subseteq T^m$$

Example of ID-net Bindings. Figure 5.5 shows an example of ID-net transition with inscriptions and ID tokens a, b, c. We suppose that a, b, c are variables of the same type. Transition t has two input variables $v1, v2$ and two output variables $v1, v2$.

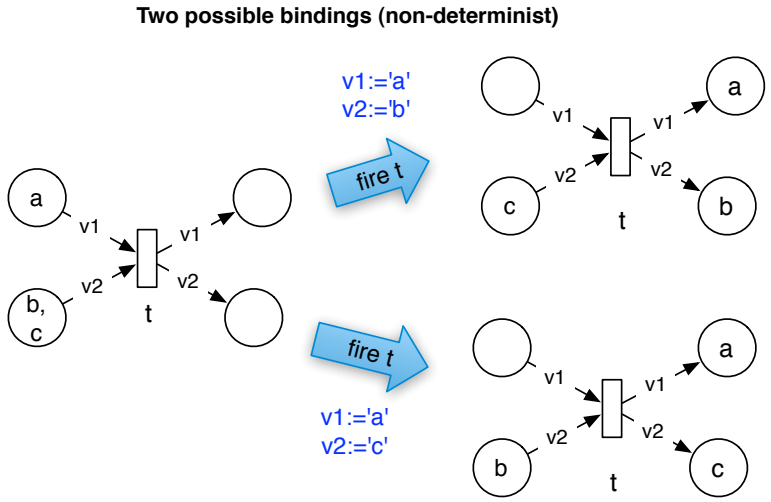


Figure 5.5: An Example of ID-net Bindings

In this example, there are two possible bindings for variable $v2$: b or c. Without further conditions, the choice is non-determinist. Moreover, in the above three states the set of seen tokens is $\{a, b, c\}$ and transition t is not a generator transition.

The binding of variables is dynamics and depends on the marking of ID-net; if the transition is synchronized with external models, the computation depends also on the state of external models.

5.4 Basic ID-net Modeling Patterns

Basic Control Flow Structure. With typed places, arcs with inscriptions, it is easy to model data flow with ID-net. However, depending on the types of domains and relations between the types, there are many possibilities. The essential control flow structure is shown in figure 5.6, they define the first-level constraints of ID-net model where annotations will be inscribed on this structure as second-level constraints. In this section, we will identify several common cases in data flow modeling.

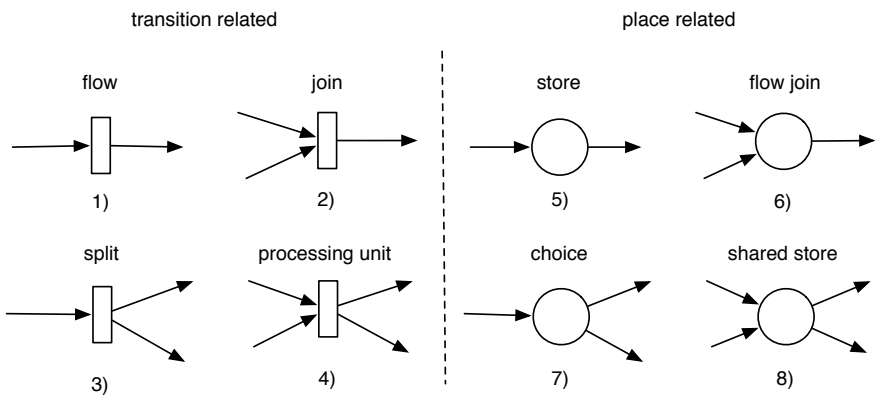


Figure 5.6: Basic Control Flow Structure of ID-net without Annotations (as classical Petri nets)

5.4.1 Data Flow Modeling with Arc Inscriptions

Token Pipes (with Pipe Transition). *If a variable name is inscribed both on a transition's input arc and one or several of its output arcs, it means the tokens consumed by the transition and produced by the transition will be guaranteed as the same*, as shown in figure 5.7.1).

When several output arcs of the transition have the same variable name as inscription, it means duplication of the token, as shown in figure 5.7.2). Figure 5.7.3) is a strict-join transition which produces the same token as it consumes.

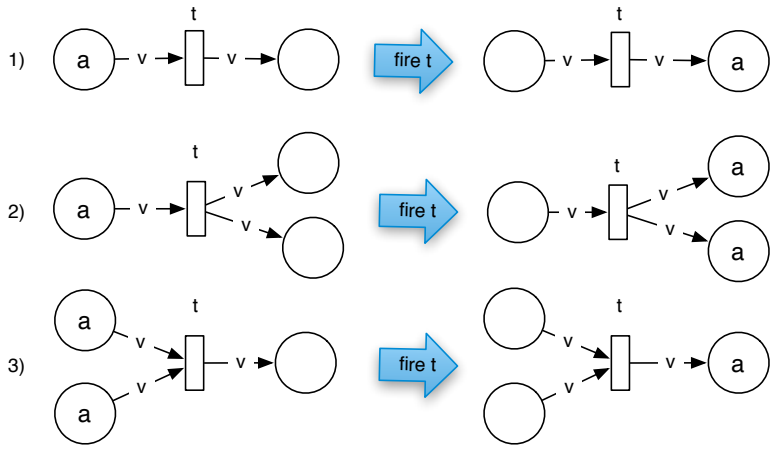


Figure 5.7: Token Pipes

Token Transformers (with Generator Transition). *If different variable names are inscribed on a transition's input arcs and output arcs*, as shown in figure 5.8, *tokens consumed by the transition and produced by the transition are not guaranteed to be the same (there are chances to be so, but it depends on the inscribed function).*

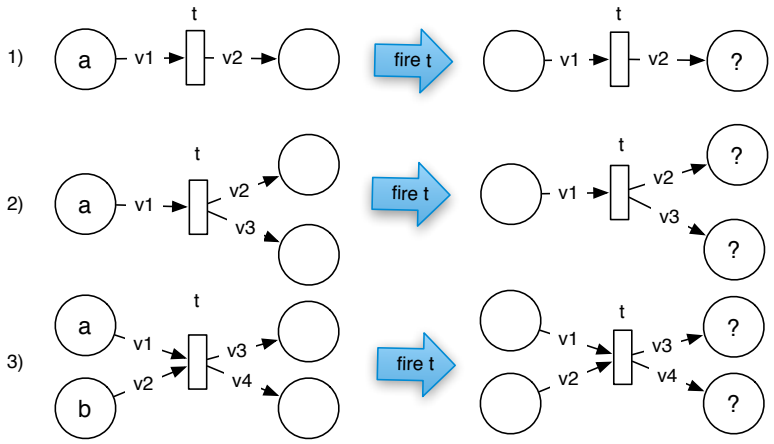


Figure 5.8: Token Transformers

In cases of figure 5.8.1), transition t consumes token a, i.e. $v1$ is bound to a, and the binding of $v2$ after firing t is unknown for this ID-net model, because it will be computed by external models. Similarity for figure 5.8.2) and 5.8.3), the bindings of variable after the transition will be determined dynamically.

For example, if we use a data model to manage the addresses of customer using the relational model $CustomerAddr(CustomerID, Name, Street, City, State, Zip, Phone)$, figure 5.9 shows a possible data flow of changing working tokens.

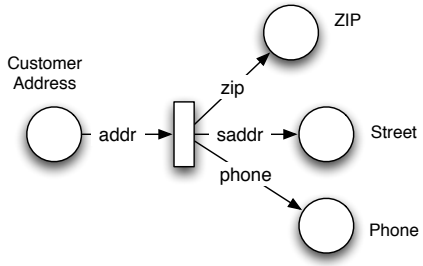


Figure 5.9: A Data Flow of Token Transformer

Constraints in the data model, e.g. functional dependency, can imply constraints while firing a transition. In example 5.8.1), $v2$ is computed by the inscrip-

tion function of t using $v1$ as an input parameter. The relation between $v1$ and $v2$ must exist somewhere: either this relation is specified in the data model, e.g. as an algebraic function $v2 = f_t(v1) = v1 * 2$, or it is defined in a relational model as in the example of customer address. The functional dependencies in the example of customer address are: $CustomerAddr \rightarrow ZIP$, $CustomerAddr \rightarrow Street$ etc.

An Example of Online Auction Process for the Composition of Data model and ID-net. On the left side of figure 5.10, we shows a domain object model (data model) for online auction process represented in UML class diagram. From this diagram, it is straightforward to derive operations to manipulate domain objects, e.g create objects, getters, setters, navigate between the objects etc.

On the right side of the figure 5.10, we show some example of pseudo codes allowing to create and manipulate the objects.

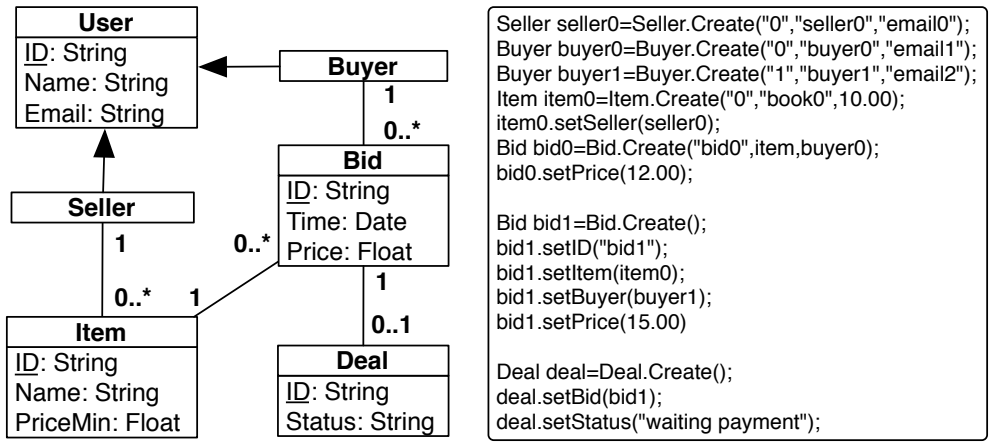


Figure 5.10: Domain Object Model for the Online Auction Process

Figure 5.11 shows the ID-net model of an online auction process using the data objects defined in figure 5.10: *Item, Bid, Deal*. In this example, according to their functional dependencies, *Bid* objects are created based on *Item* objects, and *Deal* objects are created based on *Bid* objects. Some transitions consume and produce different kinds of data objects, e.g.:

- Firing transition *bid* will consume an *Item* object from *P0*, and produce a *bid* object into place *P1*.
- Transition *create deal* will consume an *Item* and its final *bid*, then produce a corresponding *Deal* object.

Notice that there are several ways to model the auction process.

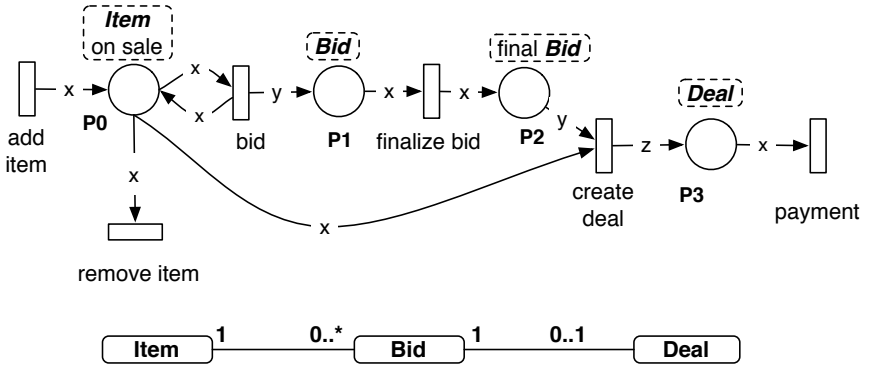


Figure 5.11: ID-net Model of the Online Auction Process

Figure 5.12 shows the details of each ID-net transition annotated by the operations on the domain objects, i.e. how ID-net and the data models are synchronized via the annotations.

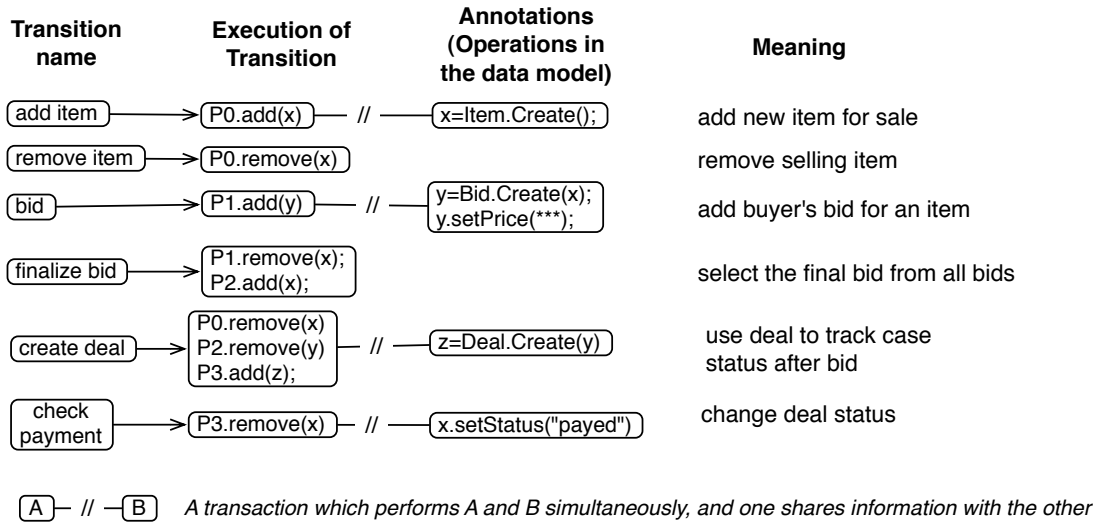


Figure 5.12: Synchronizations between ID-net and the Data Model

5.5 ID-net Controlled System

Since an ID-net specifies the behavior of a system by describing its control flow, models using different kinds of resources (tokens) can collaborate with ID-net. In general, these models collaborate with ID-net and respect the constraints of syn-

chronization. The collaborations between ID-net and external systems are realized by inscribing (synchronizing) external actions on ID-net transitions. We call the synchronized system ID-net Controlled System, or IDCS.

According to the role of ID-net control flow in an IDCS, we distinguish two kinds of IDCS: *ID-net Orchestration* and *ID-net Choreography*.

Co-system and ID-net Orchestration A *co-system* of an ID-net is a system which *passively* interacts with the ID-net during its execution. It can run autonomously and collaborate with ID-net, but it will not trigger ID-net transitions during its execution. Co-systems are often stateful passive *resources* which accept synchronous requests from ID-net, i.e. no call-backs.

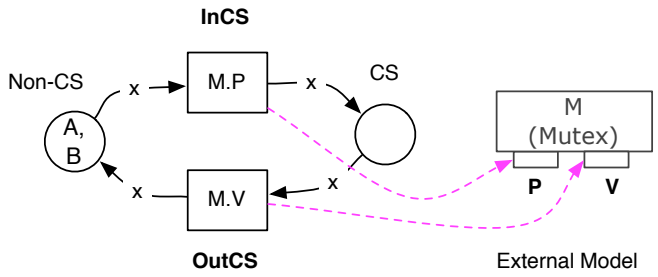


Figure 5.13: Example of ID-net and its Co-system

For example, figure 5.13 shows an IDCS which consists of *two-process A, B* interacting with a *mutex (lock) M*:

- two tokens *A, B* represent two processes which try to acquire the mutex *M* in order to enter the *Critical-Section (CS)*
- ID-net transitions *InCS, OutCS* are annotated by actions *M.P, M.V*, respectively. These annotations imply dynamic bindings and synchronizations of processes *A, B* and *M*.
- The *mutex M* is a co-system of the two-process ID-net model, because the *mutex* is driven by the two-process model in the synchronized system.

A co-system can be another ID-net model which represent passive resources. If the semantics of external model (i.e. mutex *M*) is also given by an ID-net, as shown in figure 5.14, the semantics of synchronized system can be obtained by merging the LTS of the ID-net models. Notice that variable inscriptions are adopted to avoid name conflict.

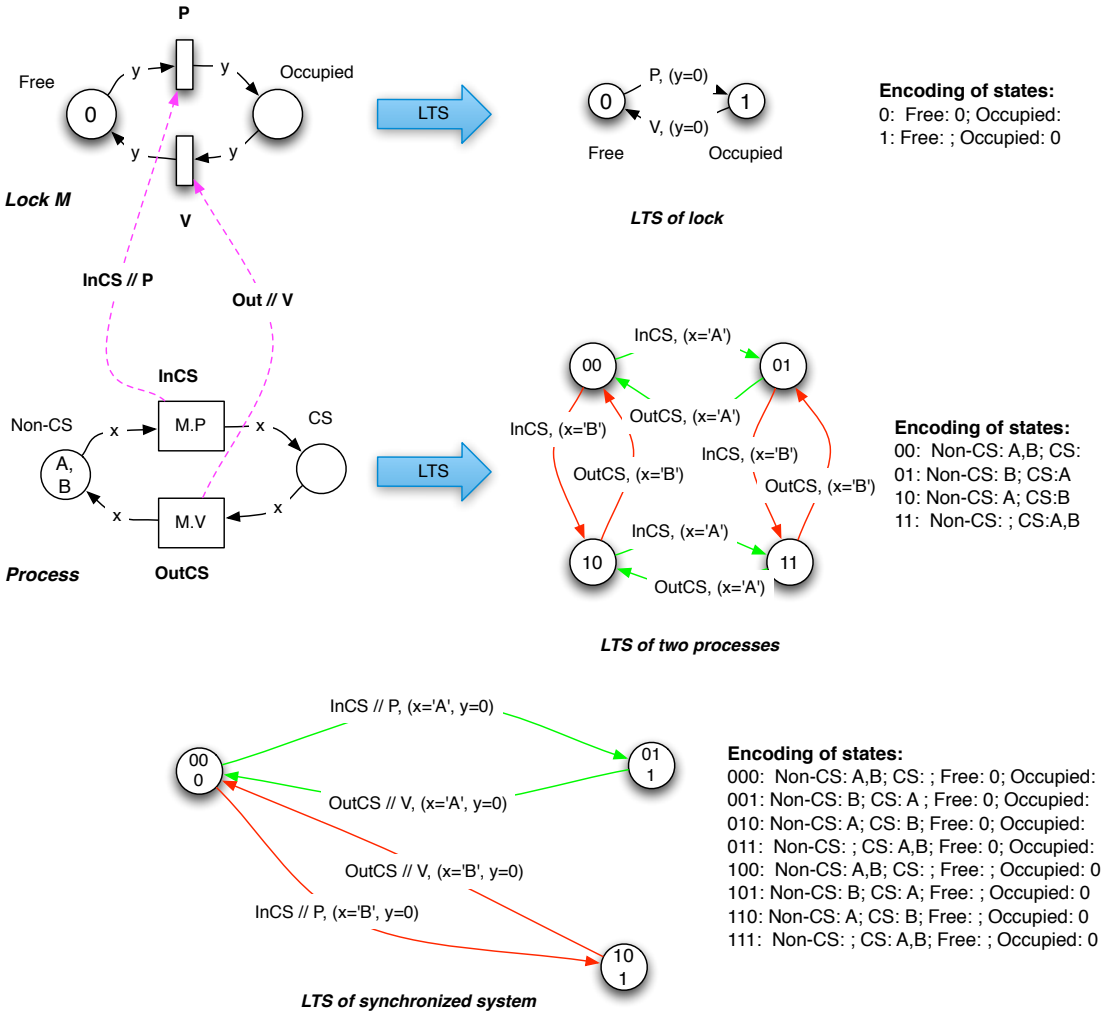


Figure 5.14: Example of ID-net with anther ID-net as Co-model and their LTS

Consequently, the labeled transition system of synchronized model can be computed for the purpose of model verification. In this example, the final LTS is exactly the same as figure 4.7 in previous chapter. Notice that we use a notation with dot "." in this chapter for better readability of inscriptions, while in previous chapter the syntax for actions is different.

An ID-net with one or several co-systems is called an *ID-net orchestration*.

Definition 5.5.1. ***ID-net Orchestration***. *An ID-net Orchestration is a triple* $< N, M, Sync >$, *where*

- $N =< D, P, T, F, V, Ains, Tcond >$ *is the control-flow model of the system, i.e. an ID-net.*
- M *is a set of co-systems of* N
- *Sync is the set of synchronization specifications between the actions of* N *and* M, *which satisfy the following property:*

$$\forall < a, op, b >\in Sync, < a, op, \widetilde{b} >\notin Sync, op \in \{//, \oplus\}, a \in T, b \in A$$

where A *is the set of actions from* M.

Peer-system and ID-net Choreography A peer-system of an ID-net will trigger ID-net transition during its execution, i.e. sometimes the ID-net is driven by events generated by its peer-systems. Peer-systems are often active *processes* which influence and interact with ID-net models, the interaction may follow specific protocols and can be asynchronous.

Definition 5.5.2. ***ID-net Choreography***. *An ID-net Choreography is a triple* $< N, M, Sync >$, *where*

- $N =< D, P, T, F, V, Ains, Tcond >$ *is the control-flow model of the system, i.e. an ID-net.*
- M *is a set of peer-systems of* N
- *Sync is the set of synchronization specifications between the actions of* N *and* M, *which satisfy the following property:*

$$\forall < a, op, b >\in Sync, < \widetilde{a}, op, b >\notin Sync, op \in \{//, \oplus\}, a \in T, b \in A$$

where A *is the set of actions from* M.

5.6 Extension of ID-net

In order to get modeling and verification facilities, some extensions are added into ID-net.

5.6.1 Fresh Token

The definition of ID-net generator transition only implies having different output tokens than the input tokens. In this case, there are two possibilities: the output tokens already exist in the external models, e.g. reuse/recycle existing resources; or the output tokens are "fresh", meaning they are created during the execution of the transition. Because ID-net model does not have knowledge about external models, the semantic of "fresh" has to be defined in ID-net, i.e. "fresh" means "unseen till now".

The behavior of ID-net transitions can be observed partially from its surrounding arc inscriptions: *tokens pipes* will not generate fresh tokens; *token transformers* **may** generate fresh tokens.

Definition 5.6.1. ***"Seen" tokens during execution of ID-net.***
For an ID-net with initial marking $N =< D, V, P, T, F, Ains, Tcond >$ with initial marking m^0, the set of seen tokens w.r.t. execution path seq at current marking m, $Seen(N, m)$, is the set of all appeared tokens from initial marking m^0 to current marking m by following seq, i.e.

$$seq \in Path(\mathcal{A}_{<N,m^0>}, m^0, m)$$

$$Seen_{seq}(N, m) = \{val \mid val \in \bigcup_{<t,Bind>\in seq, Bind \models Tcond} Bind\}$$

where seq is current execution path (trace) from m^0 to m.

The set of seen tokens in ID-net allows the monitoring of resources used by a concurrent system. The number of seen tokens in an ID-net can be *increasing* or *fixed* during the evolution of the system. *Increasing* number means the ID-net uses continuously unseen (new) resources. *Fixed* number means the set of resources used by ID-net is limited. In general, the number of seen tokens will increase until arriving a fix point where all available resources are used at least once.

Definition 5.6.2. ***Fresh Tokens in ID-net.***
For an ID-net $N =< D, V, P, T, F, Ains, Tcond >$ with initial marking m^0, a token val is said to be "fresh" at marking m w.r.t an execution path seq iff:

- $m^0 \xrightarrow{seq} m, seq =< t_0, Bind^0 >, < t_1, Bind^1 > ... < t_n, Bind^n >$
- *val does not exist in any marking which precedes m in seq, i.e.*

$$\forall k \neq n \text{ that } m' \xrightarrow{<t_k, Bind^k>} m'' : val \notin Seen_{\mathcal{A}_{<N,m^0>}}(N, m'')$$

- *for transition* $m''' \xrightarrow{<t_n, Bind^n>} m : val \in Bind^n, Bind^n \models Tcond$

During execution of an ID-net, fresh tokens are "unseen" tokens until firing a transition.

Co-model which manages seen tokens We can define a *concurrently accessible set* to maintain the seen tokens, this set has two operations: *seen* and *fresh*. Supposing s is a set containing the set of seen tokens, the basic operation is to add element into the set e.g.:

- $\emptyset$ is an empty set.
- $x \notin \emptyset$
- $x \neq y, x \notin s \Rightarrow x \notin s + \{y\}$
- $x = y, x \in s \Rightarrow x \in s + \{y\}$
- $\{x\} + \emptyset = \{x\}$
- $\{x\} + s = s + \{x\}$
- $(\{x\} + s) + \{y\} = (s + \{y\}) + \{x\}$

The semantics of the operations to maintain the set s are described as follows, both of them support simultaneous accesses (specified using //):

$$seen : \frac{x \in s}{s \xrightarrow{seen(x)} s}, \frac{x \notin s}{s \xrightarrow{seen(x)} s + \{x\}}, \frac{x \in s, y \notin s}{s \xrightarrow{seen(x,s)//seen(y,s)} s + \{y\}}$$

$$fresh : \frac{x \notin s}{s \xrightarrow{fresh(x)} s + \{x\}}, \frac{x \notin s, y \notin s}{s \xrightarrow{fresh(x)//fresh(y)} (s + \{x\}) + \{y\}}$$

Initially $s = \emptyset$. The different between operation *seen* and $fresh$ is that *seen* is always firable and it puts unseen tokens into s; while $fresh$ is expecting unseen tokens as input parameters and it also put them into s, otherwise operation $fresh$ is unfirable.

This co-model can be synchronized with any ID-net model as follows:

- for each ID-net transition t and the set of variables $Tins_{out}(t) = \{t_1, t_2...t_n\}$, we inscribe transaction $seen([v_1])//seen([v_2])//...//seen([v_n])$ on t.
- if we want to imply a transition t to produce fresh tokens, we replace the transaction $seen([v_1])//seen([v_2])//...//seen([v_n])$ by $fresh([v_1])//fresh([v_2])//...//fresh([v_n])$.

In this case, operation *seen* acts as monitoring service for seen tokens, and operation *fresh* implies supplementary constraints to the IDCS to generate fresh tokens. Notice that annotating *fresh* operation on *pipe transitions* will result unfirable transitions.

5.6.2 ID-net with Encapsulation

ID-net models can be modularized via the notion of encapsulation. Concretely, it consists of distinguishing visible (observable) and invisible (unobservable internal) transitions in ID-net. The syntax of the encapsulation mechanism has slightly extended from CO-OPN, i.e. an ID-net can be put into a rectangle and:

- black boxes on its edges and *outside* the rectangle represent incoming events
- white boxes on its edges and *inside* the rectangle represent outgoing events
- there are may be mixed (black and white) boxes to represent synchronous incoming/outgoing events and outgoing/incoming events.
- events (transitions) on the edges are visible from outside of the model

Other transitions are internal transitions, thus unobservable from outside. Incoming and outgoing events are both ID-net transitions but being triggered differently. In general, outgoing events and internal transitions are driven by the execution engine, and incoming events are trigged by external events or systems.

Definition 5.6.3. ***Encapsulated ID-net****. An encapsulated ID-net N is a tuple $< D, V, P, T, F, Ains, Tins, Tcond, EventIn, EventOut >$, where:*

- $D, V, P, T, F, Ains, Tins$ *are elements of ID-net.*
- $EventIn \subseteq T$*, $EventIn$ is a set of input events of N.*
- $EventOut \subseteq T$*, $EventOut$ is a set of output events of N.*
- $EventIn \cap EventOut = \emptyset$*.*

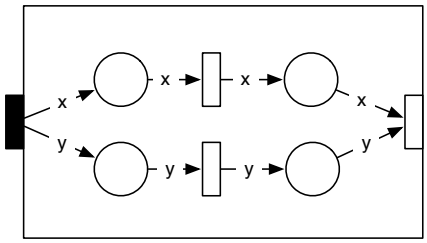


Figure 5.15: ID-Net with Encapsulation

Definition 5.6.4. ***Internal Transitions.***
For an encapsulated ID-net $N =< D, V, P, T, F, Ains, Tins, Tcond, EventIn, EventOut >$. *The set of transition* $T \backslash (EventIn \cup EventOut)$ *is called internal transitions of* N.

While building complex models, *only visible events of ID-net can be synchronized with other models*, this will reduce the complexity of composition.

5.6.3 Encapsulation of ID-net Controlled System

With encapsulated ID-net, the synchronizations between ID-net and its co-models are observable, i.e. similar to the interactions between objects in object-orientation paradigm as shown in figure 5.16. This kind of model emphasizes the concurrent access of shared data models.

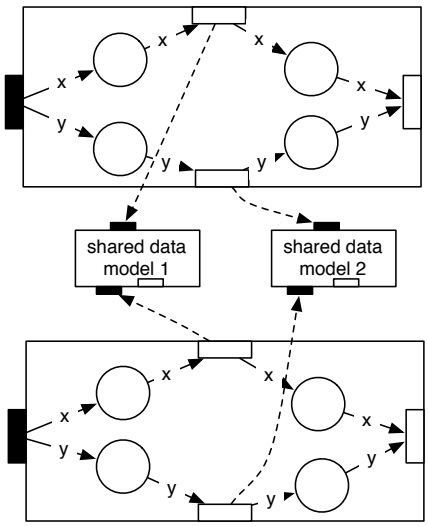


Figure 5.16: ID-Net with Encapsulation and Shared Data Models

It is also possible to encapsulate ID-net and its co-models together and obtain reusable components. As shown in figure 5.17, in this case the details of co-models will be hidden from outside. This kind of encapsulated IDCS is called the *Service Component Model (SCM)* and it provides better reusability of components, we will develop this concept later in chapter 7.

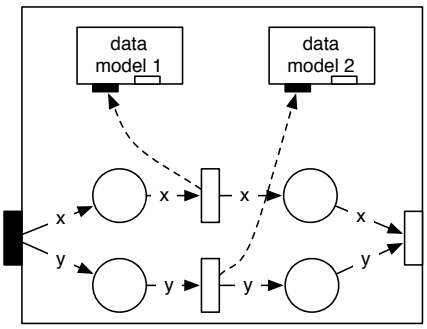


Figure 5.17: Encapsulation of ID-net and its Co-models

Transitions Inscription Function with External Parameters A transition inscription function has at least the parameters coming from arc inscriptions. In addition, they can have *external parameters*. These external parameters are used by the inscription functions which will deal with external models, and they don't have direct relation with ID-net tokens.

5.7 Summary & Discussions

In this chapter, we give the formal definitions of ID-net and some related concepts. ID-net is designed to separate concerns represented by resources (tokens) and resource-related models, and each ID-net transition can be synchronized with different models.

ID-net only represents the control structure of distributed complex systems, other aspects such as resource and data processing should be introduced to ID-net model via the inscription mechanism whose semantic implies model composition of ID-net and its co-models.

Both *ID-net Orchestration* and *ID-net Choreography* are *ID-net Controlled System (IDCS)*. $IDCS$ is the synchronized composition of systems with respect to the constraints of the synchronizations. The composition can be interpreted statically (e.g. during compilation) or dynamically (e.g. during execution). ID-net has the following responsibilities in an $IDCS$:

- *Dependency (control-flow).* It defines the causal relationship between the activities (events/actions) in a system.
- *Concurrency.* It captures the resources sharing problem in concurrent, dynamic environment.
- *Dynamic composition of systems.* By using ID tokens and external models, process models and resources models are dynamically composed together.
- *Information control.* Each transition has some accessible places; identification tokens are transfered across the transitions and places. These tokens are accessible by external models while the transition is being fired.

The first two responsibilities correspond to the classical Petri net semantics, the last two responsibilities make ID-net a collaborative model, i.e. it coordinates with it co-models during its execution. In an ID-net controlled system, an m-enabled transition t may not be fired successfully due to external models. This is because the synchronizations are transactional and IDCS should verify conditions inside and outside the ID-net model during its execution.

Moreover, the development of IDCS can be *incremental* and *iterative*, model verification can be performed at each iteration of development.

Chapter 6

Modular Verification of ID-net Controlled System

As discussed in previous chapter, an ID-net Controlled System (IDCS) is a multi-dimentional model with clear separation of concerns, e.g. control model and data model. This separation of concerns has several advantages for the modeling and verification of system, principally for the decoupling of modeling languages and the decoupling of state space of system.

Decoupling of modeling languages. In an IDCS, each concern or aspect of a system can be modeled by using a domain-specific language, i.e. resulting domain-specific models (DSMs). These DSMs are coupled via the synchronization mechanism and the identification tokens of ID-nets. Decoupling of modeling languages implies the separation of concerns during the modeling phase and it allows developers to concentrate on a specific concern each time.

Decoupling of state space. Because an IDCS is a set of domain-specific models whose behaviors are orchestrated by ID-nets, we can check and verify the properties of DSMs and ID-nets separately. The modular verification of IDCS can be done via three phases:

1. Verify the properties of ID-net
2. Verifies the properties of each DSM
3. Verify the properties of IDCS

Among them, step 3 is the verification of the whole system and it is the most complex one. In our approach, the strategy of doing step 3 can be optimized according to results of step 1 and 2.

For example, in a concurrent system, many processes are sharing limited resources. Defined by models somewhere, each kind of processes have their behaviors, and each kind of resources have their states and specific rules of using them. A natural approach is to verify the process models and the resource models separately, then put them together and verify the whole system.

In this chapter, we will analyze how to deduce properties of IDCS from the properties of ID-net and the DSMs. First of all, we will recall several Petri net properties which are useful and relevant to business process modeling. Then, we will analyze or define/redefine the properties for ID-net.

6.1 Properties of Petri net

6.1.1 Definitions (Recall)

The following properties are related to Petri net $N =< P, T, F, W >$. $< N, m >$ means Petri net N with marking m.

Definition 6.1.1. ***Live.*** *A Petri net N with marking m, i.e. $< N, m >$, is live iff, for every reachable state m' and every transition t there is a state m'' reachable from m' which enables t.*

A Petri net is **structurally live** if there exists an initial state such that the net is live.

Definition 6.1.2. ***Deadlock marking*** *A marking m of a Petri net $< P, T, F, W >$ is called a deadlock iff no transition $t \in T$ is enabled in m (i.e. $m \xrightarrow{t}$ does not hold for any $t \in T$, or $T^m = \emptyset$).*

Definition 6.1.3. ***Petri net with Deadlock, Deadlock Free Petri net.*** *A Petri net N is said to have a deadlock if one of its reachable marking is a deadlock, or deadlock free otherwise.*

Definition 6.1.4. ***Bounded, Safe.*** *A Petri net N with marking m is bounded iff, for every reachable state and every place $p \in P$, there is a natural number n such that the number of tokens in p is less than n. The net is* safe *iff for each place the maximum number of tokens does not exceed 1.*

A Petri net is *structurally bounded* if the net is bounded for any initial state.

Definition 6.1.5. ***Concurrence (Persistence).*** *For a Petri net N, a set of transitions $\{t_1, t_2, ...t_n\} \subseteq T$ is said to be concurrent or concurrently enabled in a marking m of N, iff*

$$\forall p \in P : m(p) \geq W(p, t_1) + W(p, t_2) + ... + W(p, t_n)$$

For example, if t_1 and t_2 are concurrent in a marking m, it means:

- they are both enabled in marking m, i.e. $m \xrightarrow{t_1}$ and $m \xrightarrow{t_2}$, and
- they do not disable each other, i.e. firing t_1 leaves t_2 enabled and vice versa.

Definition 6.1.6. ***Conflict.*** *Transitions t and t' are in conflict in marking m, if they are both enabled and*

$$\forall p \in P : m(p) < W(p, t) + W(p, t')$$

A conflict occurs in a marking m when two transitions t and t' are both enabled, but the set $\{t, t'\}$ is not concurrent. In case of conflict, firing one transition may disable other transitions.

Conflict is one of the factors of *non-determinism* in Petri net because the order of firing transitions will influence the evolution of system.

Definition 6.1.7. ***Independent transitions.*** *Two transitions t_1 and t_2 are called independent iff for markings m in which both are enabled, the following holds: let m_1, m_2 be markings such as $m \xrightarrow{t_1} m_1$ and $m \xrightarrow{t_2} m_2$. Then t_2 is enabled in m_1, t_1 is enabled in m_2, and there is m_3 such that:*

$$m_1 \xrightarrow{t_2} m_3 \text{ and } m_2 \xrightarrow{t_1} m_3$$

Definition 6.1.8. ***Strongly connected.*** *A Petri net is strongly connected iff, for every pair of nodes (i.e. places and transitions) x and y, there is a directed path leading from x to y, i.e.*

$$\forall x, y \in P \cup T : x(F^*)y$$

where $x(F^*)y$ is a path of flow from x to y.

Definition 6.1.9. ***State machine.*** *A Petri net is a state machine iff each transition has at most one input place and at most one output place, i.e. for all $t \in T : |\bullet t| \leq 1$ and $|t \bullet| \leq 1$.*

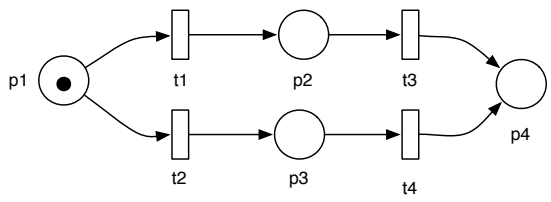


Figure 6.1: A State Machine

A state machine has branches in its structure, but no splits and joins on transitions, as shown in figure 6.1.

Definition 6.1.10. ***Marked graph.*** *A Petri net is a marked graph iff each place has at most one input transition and at most one output transition, i.e. for all* $p \in P : |\bullet p| \leq 1$ *and* $|p \bullet| \leq 1$.

Marked graphs are Petri nets with only unbranched places, as shown in figure 6.2. In marked graph, every place processes exactly one pre- and one post-condition, no conflict situations are possible. An important property of marked graphs is that token count is an invariant on each cycle.

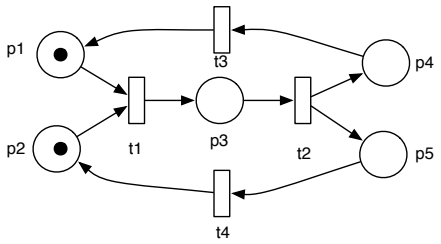


Figure 6.2: A Marked Graph

Definition 6.1.11. ***Free-choice.*** *A Petri net N is a free-choice Petri net iff, for every two place* p_1 *and* p_2 *either* $(p_1 \bullet \cap p_2 \bullet) = \emptyset$ *or* $p_1 \bullet = p_2 \bullet$.

In free-choice Petri net, one transition out of several transitions involved in a conflict may be chosen freely and independently to fire. Free-choice Petri net is a class of Petri nets for which strong theoretical results and efficient analysis techniques exist.

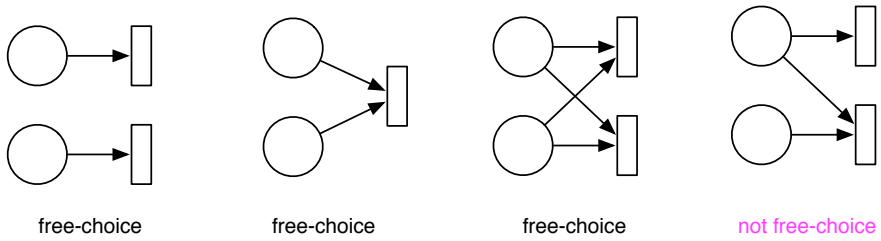


Figure 6.3: Examples of Free-Choice Petri net

6.2 ID-net Properties

6.2.1 Mapping From ID-net to Petri net

The possibility of mapping from ID-net to Petri net (P/T net) is one of the design objectives of ID-net.

Definition 6.2.1. *Projection function from ID-net to Petri net. For an ID-net* $N =< D, V, P, T, F, Ains >$*, its projection to Petri net is a function* $Proj(N) =< P, T, F, W >$*, where:*

$$\forall f \in F, W(f) = |Ains(f)|$$

W *is the set of* $W(f)$*. The projection of marking* $m(p)$*:*

$$Proj(m(p)) = |m(p)|$$

$Proj(m)$ *is the set of* $Proj(m(p))$*, i.e.*

$$Proj(m) = \{Proj(m(p)) \mid p \in P\}$$

Notice that the function $Proj$ is overloaded for $N, m(p)$, and m.

Petri nets are ID-nets using exclusively black tokens, and the number of variables inscribed on an arc is the weight of the arc.

6.2.2 Properties Related to P/T net Projection

Most Petri net property definitions mentioned in the previous section are valid for ID-net, in this subsection, we will discuss the preservation of properties in ID-net with its Petri net projection.

For an ID-net $N =< D, V, P, T, F, Ains, Tcond >$ and its Petri net projection $Proj(N)$. Let us suppose N has an initial marking m, then $Proj(N)$ has marking $Proj(m)$.

Definition 6.2.2. *For a marked ID-net* $< N, m^0 >$*, the set of its reachable markings is* $reach(N, m^0)$*. The projection of its reachable markings is:*

$$Proj(reach(N, m^0)) = \{Proj(m) \mid m \in reach(N, m^0)\}$$

Proposition 6.2.3. $Proj(reach(N, m^0) \subseteq reach(Proj(N), Proj(m^0))$.

Proof. During execution, because ID-net N has more constraints (e.g. conditions on resources or values of tokens) than its projection $Proj(N)$, some transitions may not be enabled due to these constraints.

Proposition 6.2.4. *The mapping from* $reach(N, m^0)$ *to* $reach(Proj(N), Proj(m^0))$ *is non-injective and non-surjective, i.e.*

- *for any reachable marking* $m \in reach(N, m^0)$ *there is a corresponding marking* $Proj(m) \in reach(Proj(N), Proj(m^0))$.
- *there may be more than one markings in* N *which have the same corresponding marking in* $Proj(N)$.
- *not every marking of* $Proj(N)$ *can have its corresponding marking in* N.

Proof. According to two above propositions and definitions of $Proj(N)$ and $Proj(m^0)$.

Proposition 6.2.5. *For* $< N, m^0 >$*, if* $Proj(N)$ *has deadlock at marking* $Proj(m)$ *and* $Proj(m) \in reach(Proj(N), Proj(m^0))$*, then* N *has deadlock at marking* m.

Proof. If $Proj(N) =< P, T, F, W >$ has deadlock, i.e. there exists m that $\nexists t \in T, m \xrightarrow{t}$. This means that no transition has enough pre-condition tokens at this moment. If no transition has enough tokens to be fired, N also has deadlock.

For example, figure 6.4 shows a Petri net with a deadlock marking. For any kind of token used in its corresponding ID-net, no transition will be enabled.

Proposition 6.2.6. *If* N *has a deadlock,* $Proj(N)$ *does not necessary has deadlock.*

It is easy to show this by using an example. For example, figure 6.5 show a simple ID-net with inscription, and its projection N has deadlock when $p1$ contains token with value greater than **10**, $Proj(N)$ does not have deadlock.

Corollary 6.2.7. *If ID-net* $< N, m >$ *is live, then* $Proj(N, Proj(m))$ *is live, too.*

Proof. Similar as for proposition 6.2.5.

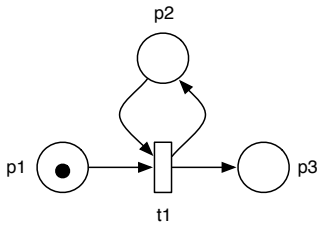


Figure 6.4: A P/T net with Deadlock Marking

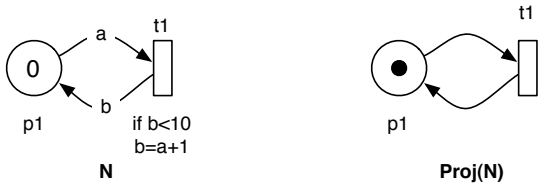


Figure 6.5: A Simple ID-net and its Projection

Corollary 6.2.8. *If* $Proj(N, Proj(m))$ *is live, ID-net* $< N, m >$ *is not necessary live.*

Proof. Similar as for proposition 6.2.6.

Corollary 6.2.9. *Bounded. For marked ID-net* $< N, m >$*, if* $< Proj(N), Proj(m^0) >$ *is bounded, then* $< N, m >$ *is bounded.*

Proof. According to proposition 6.2.3.

Corollary 6.2.10. *If ID-net* $< N, m >$ *is sound, then* $Proj(N, Proj(m))$ *is sound, too.*

Proof. Similar as for proposition 6.2.5.

Corollary 6.2.11. *If* $Proj(N, Proj(m))$ *is sound, ID-net* $< N, m >$ *is not necessary sound.*

Proof. Similar as for proposition 6.2.6.

6.2.3 ID-net Specific Properties

Definition 6.2.12. ***Sub-net.*** *For ID-net* $N =< D, V, P, T, F, Ains, Tcond >$*,* N' *is called a sub-net of* N *iff:*

- $N' =< D', V', P', T', F', Ains', Tcond' >$ *is an ID-net, and*
- $D' \subseteq D, V' \subseteq V, P' \subseteq P, T' \subseteq T, F' \subseteq F, Ains' \subseteq Ains$, *and*
- $\forall t \in T, t \in T'$, $Tcond(t) = Tcond'(t)$

denoted as $N' \subseteq N$.

A sub-net represents a sub-flow of an ID-net which can be observed independently.

Definition 6.2.13. ***Case Handling System.***
An ID-net model $N =< D, V, P, T, F, Ains, Tcond >$ *is a case-handling system (CHS) iff it satisfies the following properties:*

$$\forall t \in T,\ t \text{ is a pipe transition, i.e. } Tins_{in}(t) = Tins_{out}(t)$$

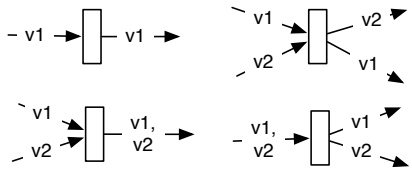


Figure 6.6: Examples of ID-net Transitions in a Case Handling System

Figure 6.6 shows some examples of ID-net transitions in a case handling system, the transitions just take input tokens and distribute them to the down-stream flow.

In a case handling system, all of its transitions preserve the original tokens and do not generate new tokens. If $|D| = 1$, i.e. only one type is used, N is called a *single-type case handling system*, meaning only one type of case is used in this system.

Case Handling System is a common approach for business process modeling. Most of the time, a CHS uses a single type to represent *Cases*, places represent *status of Cases*, and *Cases* move among the places.

Definition 6.2.14. ***Single-Type Case Handling System.*** *An ID-net model* $N =< D, V, P, T, F, Ains, Tcond >$ *is is a single-type CHS iff:*

- $|D| = 1$, *and*
- N *is a CHS.*

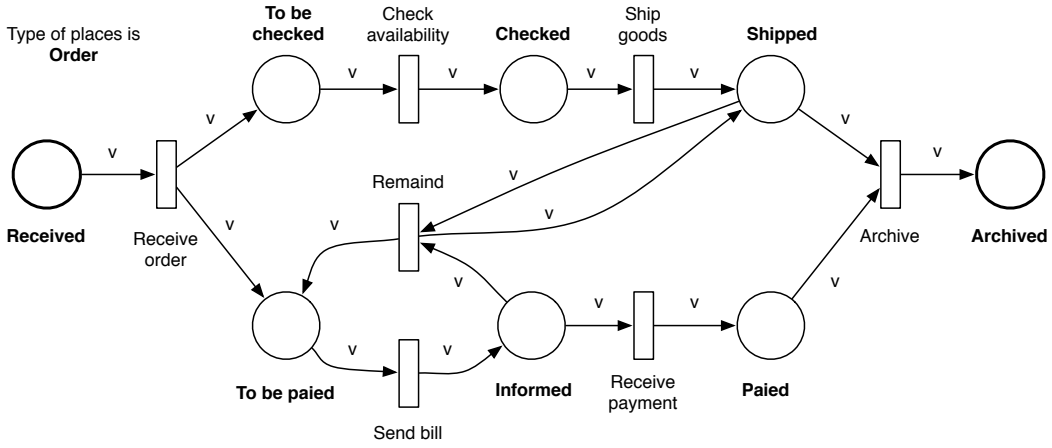


Figure 6.7: Example Case Handling System: Handling Orders

Figure 6.7 shows an example of system handling *Orders*, i.e. each token refers to an *Order*. A single-type CHS can be easily converted to Place/Transition net and to be analyzed.

Definition 6.2.15. ***Relaxed Case Handling System.*** *An ID-net model* $N =< D, V, P, T, F, Ains, Tcond >$ *is a relaxed case-handling system (CHS) iff it satisfies the following properties:*

$$\forall t \in T : Tins_{in}(t) \supseteq Tins_{out}(t)$$

Relaxed CHS allows to "forget" some cases during the evolution of the system, as shown in figure 6.8, $t2$ forgets tokens bound with $v2, v3$ and preserve tokens bound with $v1$. The part of system which process the cases from place $p1$ to $p6$ is a sub-net of the CHS.

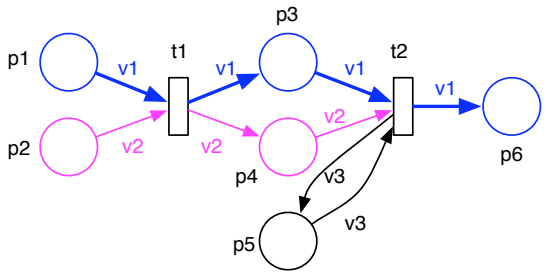


Figure 6.8: Relaxed Case Handling System

Proposition 6.2.16. *CHS and relaxed CHS do not have generator transitions.*

Proof. According to the definition of CHS, relaxed CHS, and generator transition.

Proposition 6.2.17. ***Case Projection.*** *For an ID-net $N =< D, V, P, T, F, Ains, Tcond >$, a Case Projection N' is subnet of N, i.e. $N' \subseteq N$, where:*

- $|D'| = 1$, *i.e. N' is a CHS*
- *N' is connected (always true according to definition).*

A CHS or a relaxed CHS is the synchronization of a set of single-type CHS.

Definition 6.2.18. ***Conflict in ID-net.***
For (well-formed?) ID-net $N =< D, V, P, T, F, Ains, Tcond >$ and its m*-enabled transition set T^m, N is in conflict situation at m if*

$$\exists p \in {\bullet t_i}, p \in {\bullet t_j}, t_i \in T^m, t_j \in T^m, i \neq j : (Bind(Ains(p, t_i)) \cap Bind(Ains(p, t_j))) \neq \emptyset$$

The conflict situation depends on the function *Bind*, i.e. how tokens are bound with the variables during execution. A typical situation is when there are not enough tokens in a place to fire more than one transitions simultaneously, i.e. if one transition fires, other transitions are disabled.

6.2.4 Race Condition and Properties Related to Co-Model

A *race condition* occurs when multiple processes *access* and *manipulate* the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place. Race conditions generally involve one or more processes accessing a shared resource (such a file or variable), where this multiple access has not been properly controlled.

In ID-net, resources are managed by its co-models and the operations of manipulating the resources are inscribed on ID-net transitions. For example, there are two basic operations for a variable: *read (access)*, *write (manipulate)*:

- transition t is synchronized with *read* variables a, b: $t//read(a), read(b)$
- transition t is synchronized with *write* variables a, b: $t//write(a), write(b)$

Further operations on variables can be *create (allocate)* and *delete (dispose)*, when variables are dynamically allocated and disposed.

Figure 6.9 shows an example of race condition where two ID-net transitions try to manipulate a global variable **a**. We use the notation [ ] to de-reference a

token and get the resource. In this example, because both local variables **x** and **y** will be bounded to **a**, the execution of the transition inscriptions of $t2, t3$ will be **a=1**. When more than one enabled transitions are trying to read and write the same variable (variable **a** in the example), these transitions are in "race".

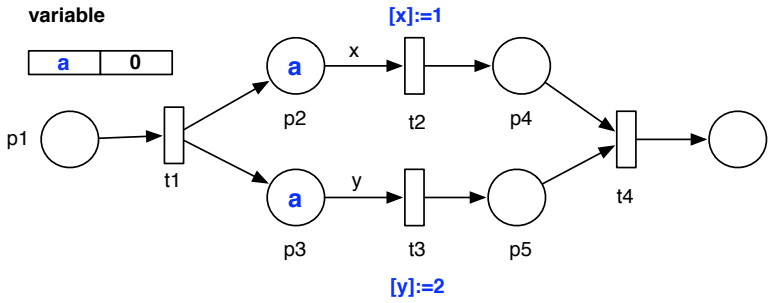


Figure 6.9: Race Condition in Annotated ID-net Model

Definition 6.2.19. ***Race Condition in ID-net.*** *An ID-net with marking m, i.e. $< N, m >$, has race condition if there exists $t_1, t_2 \in T^m$ that:*

$$Bind(Tins(t_1)) \cup Bind(Tins(t_2)) \neq \emptyset, and$$

$$\exists v \in (Bind(Tins(t_1)) \cup Bind(Tins(t_2))), t_1//write(v) \ or$$
$$\exists v \in (Bind(Tins(t_1)) \cup Bind(Tins(t_2))), t_2//write(v)$$

6.2.5 Example: Verification of Multi Exclusion Algorithm

A correct multi exclusion algorithm should at least ensure that: at any moment, maximum one process can be in the Critical Section (CS), and there is no deadlock in the system. Other properties such as *no starvation* and *fairness* may be also considered must will not be addressed in this section.

Figure 6.10 shows an example of multi exclusion algorithm for two processes a and b, described in annotated ID-net. In this example:

- $'a', 'b' \in PID$, i.e. tokens representing process ids and variable names.
- Places *NonCS, Waiting, CS* have the same type
- Transitions $t1, t2, t3$ are annotated by expressions and conditions of the data model. The implied synchronizations are: $t1//write([x]), t2//read([x]), t3//write([x])$, where $[x]$ means the value bound with variable x.
- The arc annotations guarantees that no new tokens are created and no tokens will be disappeared during evolution of the system.

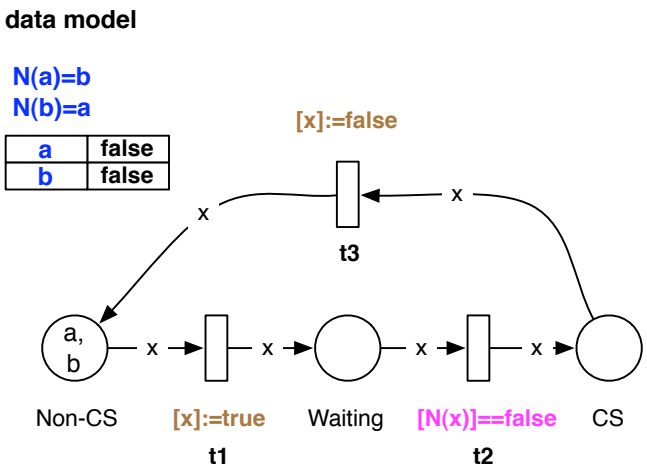


Figure 6.10: A Multi Exclusion Algorithm for two Processes

Transition System of the ID-net The marking graph of the ID-net is shown in figure 6.11. This ID-net has 9 states. A transition is a pair of *(label, binding of variables)*. Note that this ID-net only has *pipe transitions*, thus we can derive its state space directly; when there are *generator transitions*, it is not possible to compute the marking graph of an ID-net without considering the states of its co-models. In this figure, we emphasize that state ⑧ is an illegal state in a correct mutex implementation, and state ③ is a special state after the composition of ID-net with its co-models.

Petri nets Projection The Petri nets projection of the mutex model is shown in figure 6.12. Its marking graph has 6 states where some ID-net states and transitions are merged together.

Data model As the co-model of ID-net, each process has a boolean flag, initialized as *false*:

- $a, b \in VAR$, VAR is a set of global variable.
- The state of co-model $S_{co} = \{a, b\} \times \{true, false\}$, initially $a = false, b = false$ as shown in the figure. Thus $s^0_{co} = \{a = false, b = false\}$
- $N('a') =' b'$
- $N('b') =' a'$

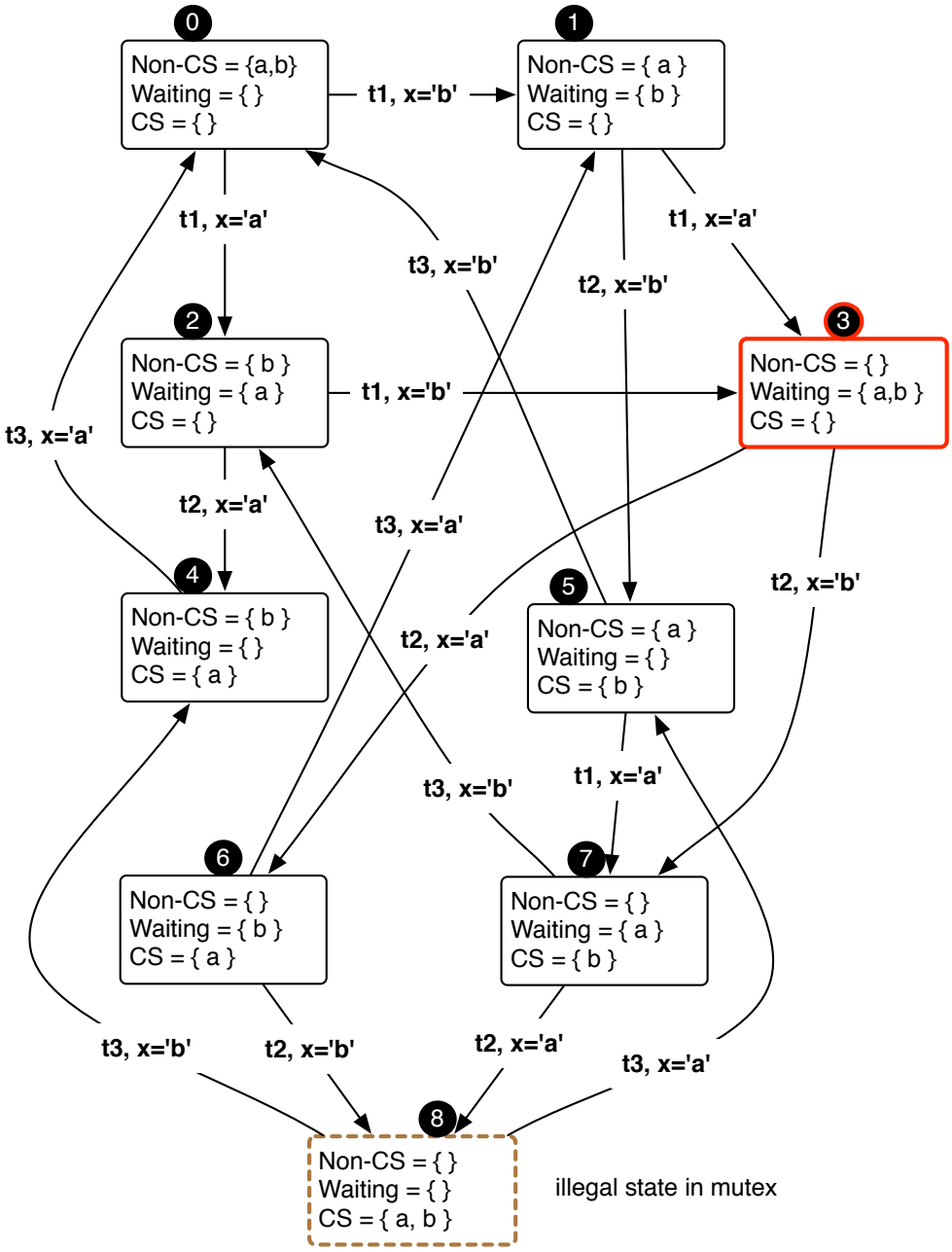


Figure 6.11: Marking Graph of the ID-net Model

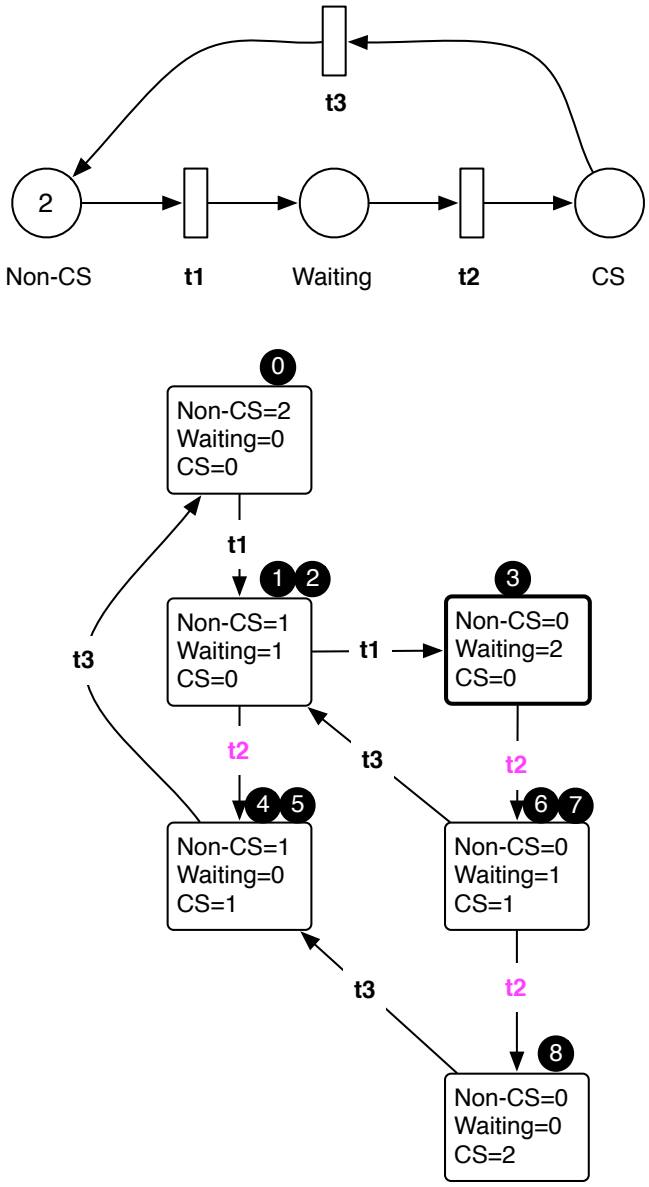


Figure 6.12: Petri nets Projection of the ID-net Model and its Marking Graph

The operation $N()$ allows to get the identification of another process then itself.

The formal semantics of $t1, t2, t3$ in the co-model (data model) are given as follows:

$$t1 : \frac{x =' a'}{S_{co} \rightarrow S_{co}[a = true]}, \frac{x =' b'}{S_{co} \rightarrow S_{co}[b = true]}$$

$$t2 : \frac{x =' a', b = false}{S_{co} \rightarrow S_{co}}, \frac{x =' b', a = false}{S_{co} \rightarrow S_{co}}$$

$$t3 : \frac{x =' a'}{S_{co} \rightarrow S_{co}[a = false]}, \frac{x =' b'}{S_{co} \rightarrow S_{co}[b = false]}$$

The bindings of variable x in above inference rules should be matched with the ID-net transition bindings, according to the semantics of ID-net inscription.

While firing an ID-net transition, the annotated conditions have to be satisfied, moreover, the inscribed actions and the synchronizations between ID-net and data models should also be performed. Thus the annotated ID-net (i.e. the synchronized system) works as follows:

1. every process starts from place *NonCS*.
2. a process sets its flag to *true*, and moves to place *Waiting*.
3. if a process in place *Waiting* finds the flag of another process is *false*, it will enter the *CS*.
4. when a process quits *CS* and moves to *NonCS*, its flag is set to *false*.

Transition system of the IDCS The transition system of the composed system can be deduced by applying the principle developed in Chapter 3, i.e. synchronized composition of two LTS.

In this example, its Petri nets projection is structurally live and deadlock-free. However, the IDCS of this algorithm has deadlock, as show in figure 6.13, i.e. when both processes are in place *Waiting* and both variables have value *'false'*, no one can enter into the *CS*.

The full transition system the IDCS of our example can be found in figure 6.14: each marking of the Petri nets can be instantiated to several pairs of ID-net marking and binding; and not all these pairs are valid states in the IDCS. The state space of the IDCS is a subset of these instances, thus if we find deadlock markings reachable from the initial marking of the IDCS, then the IDCS has deadlocks. In this figure,

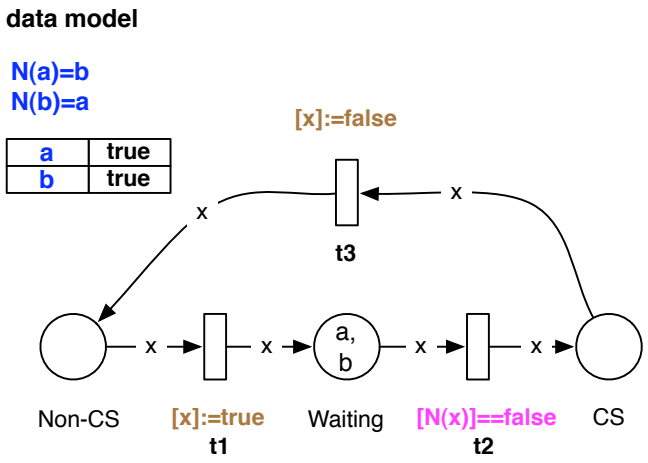


Figure 6.13: Deadlock Situation

we can see that state ⑧ does not exist anymore and state ③ is a deadlock state. Thus, *this IDCS is a mutex but has deadlocks.*

How to detect this kind of deadlock (which only exist in IDCS)? It is difficult to propose a general efficient solution for different kinds of problems and properties, however, some heuristics can be used for deadlock detection by using the marking graph of the Petri nets (for proposition):

1. first of all, check its Petri nets projection $PN = Proj(N)$ of ID-net N, if there are deadlocks, the system has deadlocks and we stop. Otherwise continue next step. As shown in figure 6.12 for our example, the projection has 6 states and it does not have deadlocks.

2. for **each** *ID-net transition with conditions*, check if there are states in PN's marking graph where the transition is the only outgoing transition (transition $t2$ and marking ② in our example), if yes:

   (a) for all the input places of this transition, try to instantiate the ID tokens with values from data model with respect to the number of tokens in the PN's marking, then instantiate the bindings of the inscribed variables on ID-net's arcs.

   (b) check the instantiations and bindings if there are situations where the transition is disabled, i.e. ID-net markings unsatisfying the conditions. If there are these markings (i.e. state of data model + bindings) and the state of data model is a valid state *, the system have deadlocks at these

*Sometimes it is not possible to know whether a state of data model is valid or not, in these

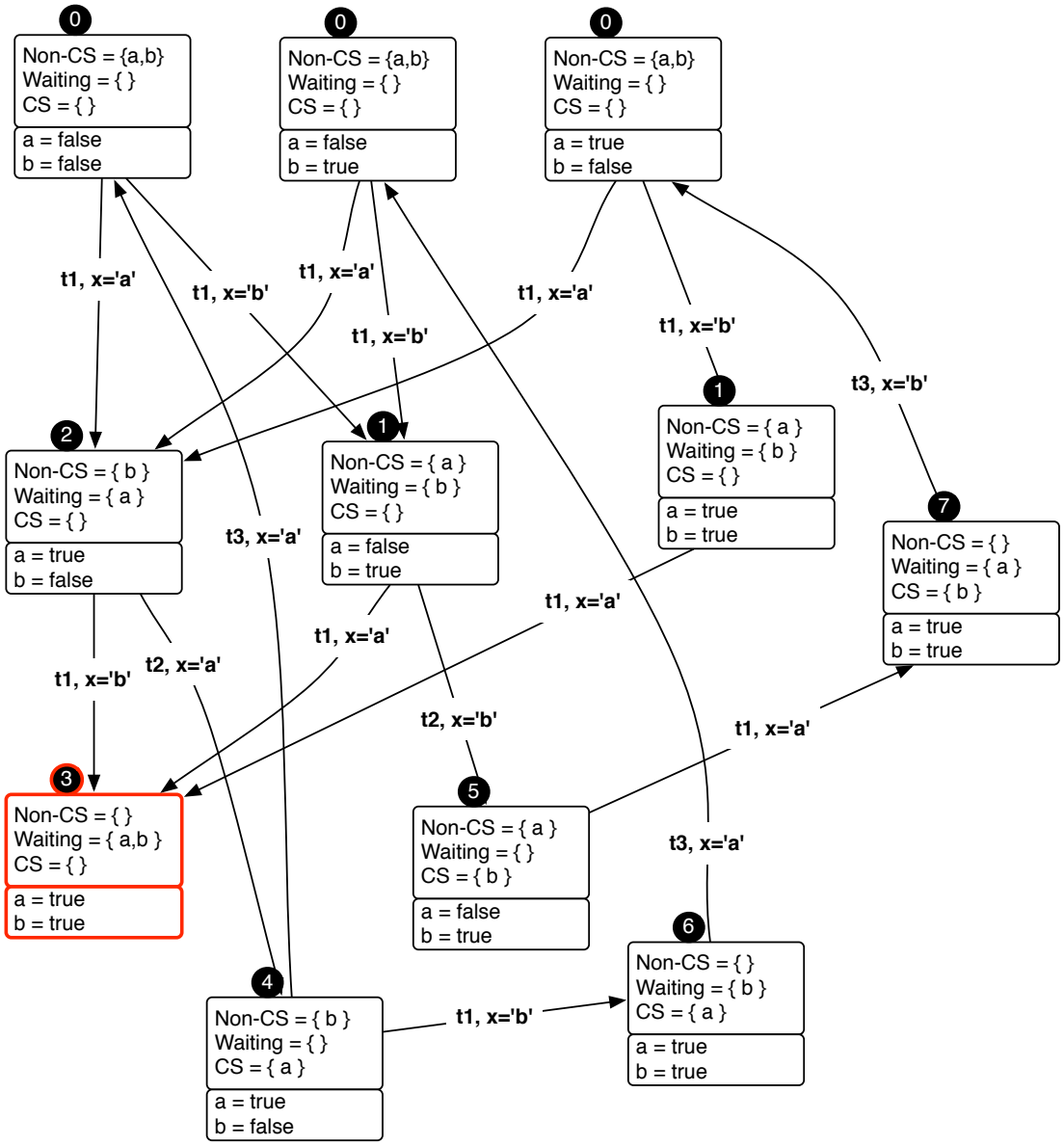


Figure 6.14: Transition System of the IDCS

markings.

3. if the transition is not the only outgoing transition of the PN's marking, i.e. several outgoing transitions exist, perform step 2 simultaneously for all the outgoing transitions of this marking. The situations where all the outgoing transitions are disabled are the deadlock markings of the IDCS.
4. repeat step 2, 3 until all ID-net transitions with conditions are checked.
5. if no deadlock is found until now, we can say that the system is deadlock-free.

6.3 Verification Framework

Figure 6.15 show our verification framework for ID-net Controlled Systems. With this verification framework, we try to answer the questions such as:

- how the properties of sub-systems influence the properties of the whole system, are they still valid?
- how to derive (new) properties of the whole system from the properties of its sub-systems, e.g. if system A has property p, system B has property q, then the composed system C will have properties r etc.

Our proposition for the verification of IDCS is in an initial stage. Nevertheless, we believe that the separation of concerns we introduced during the modeling phase can reduce the complexity of verifing complex systems.

In this Chapter, we developed several properties preservation rules in our framework:

- A property of Place/Transition net $Proj(N)$ will be valid in ID-net N, e.g. if P/T net $Proj(N)$ net has deadlock, then N has deadlock.
- A property of ID-net model will be valid in the composed model.
- A property of external model (co-model) will be valid in the composed model.
- The inverses of above assertions are not necessarily true, e.g. a property of the composed model may not exist in one of its sub-components before the composition.

In next Chapter, we will present the common usage of ID-net tokens for complex system modeling.

cases we need to explore the state space from the initial state.

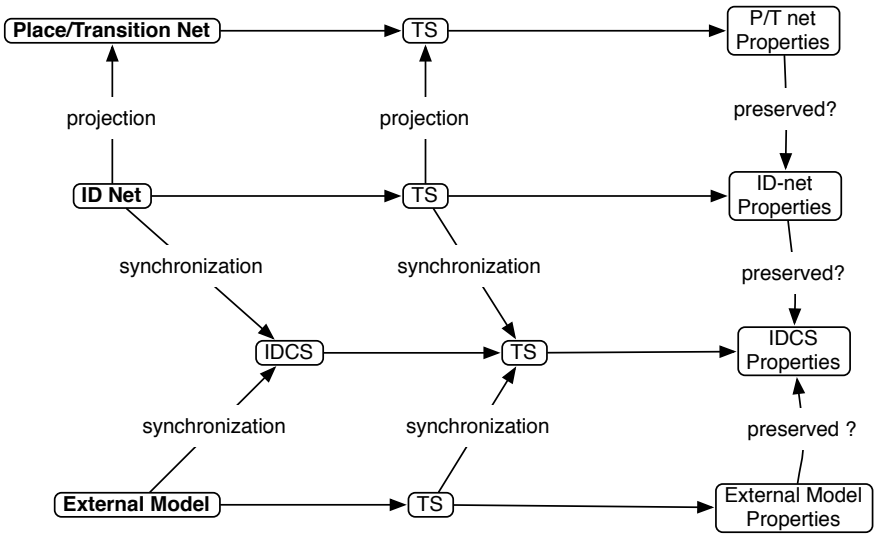


Figure 6.15: Framework to Find the Properties of Models and Their Composition

Chapter 7

Model Composition with ID-Net: Examples

In this chapter, we will illustrate the applications of ID-net in model composition, i.e. by using different types of tokens and external models manipulating resources represented by the tokens. In our approach, each kind of token is managed by a specific model, implicitly or explicitly, it synchronizes with ID-net model via the inscription mechanism.

7.1 Tokens in ID-net

ID-net provides a control structure which supports different types of tokens. Each type of tokens and the mechanism of managing the corresponding resources (which is outside of ID-net) may represent a specific concern of a system. By using different types of ID-tokens and synchronizations, ID-net acts as a "mediator" between heterogenous models for the construction of complex, concurrent, and distributed systems.

Our goal is to build a modeling framework incorporating different kinds of tokens and (domain-specific) languages around ID-net, make the whole system operational, manageable, and verifiable. From this point of view, ID-net is the control structure of parallel programs.

Examples of token types while modeling with ID-net

- Control token, e.g. $\odot$, the black token in Place/Transition net. It indicates the locations in an ID-net where the controllers or execution units are working

on, e.g. similar to the program counter (PC) of CPU. Each control token corresponds to a single-process controller unit and they *enable* transitions. Conditions and logical predicates can be considered as control tokens.

- Simple-Variable token, e.g. a, b, c. Simple-Variable tokens are referring to values of simple data types, i.e. *unstructured* data, such as primitives and enumeration values. Simple-Variable tokens are used to transfer variable references between ID-net transitions.

- Resource token. Resource tokens are referring to resources managed by specific data models. For example, *structured* data and objects. Resource tokens are used to transfer resource pointers between transitions.

- Simple-Value token, e.g. *"true", "false", "0", "1"*. Simple-Value tokens are self-contained entities and are used to transfer values between transitions. Simple-value tokens can be implemented by using simple-variable tokens which don't change their values and they don't have concurrent issues.

The different of using these types of tokens, e.g. between the control token and variable token, are not alway clear and its interpretation depends on the designer.

Merging instances of ID-nets. Similar to Place/Transition nets, ID-nets instances can be merged or isolated (by projection), in order to show single or multiple instances of the same model. In general a running model has multiple instances, an instance may have a group of tokens (e.g. referring to the same case ID).

In addition, no information is lost during the merging and isolating process, because ID-nets preserve the identities of each instance. For example figure 7.1 shows two processes, 1 and 2, which switch between states *NonCS* and *CS* can be merged into a single ID-net without losing information. It is the main different between Place/Transition net and ID-net. Note that not all ID-nets can be projected in this way.

In this Chapter, we adopt the Register Transfer Notation (RTN) to express the access and manipulation of simple variables. In fact, the annotation language for ID-net is partially inspired from RTN: its principle is to retrieve values from variables and ask other units to perform computations on the values.

Register Transfer Notation is a way to describe micro-oprations capable of being performed by the data flow (data registers, data buses, functional units) at the register transfer level of design (RT). It also describes conditional information

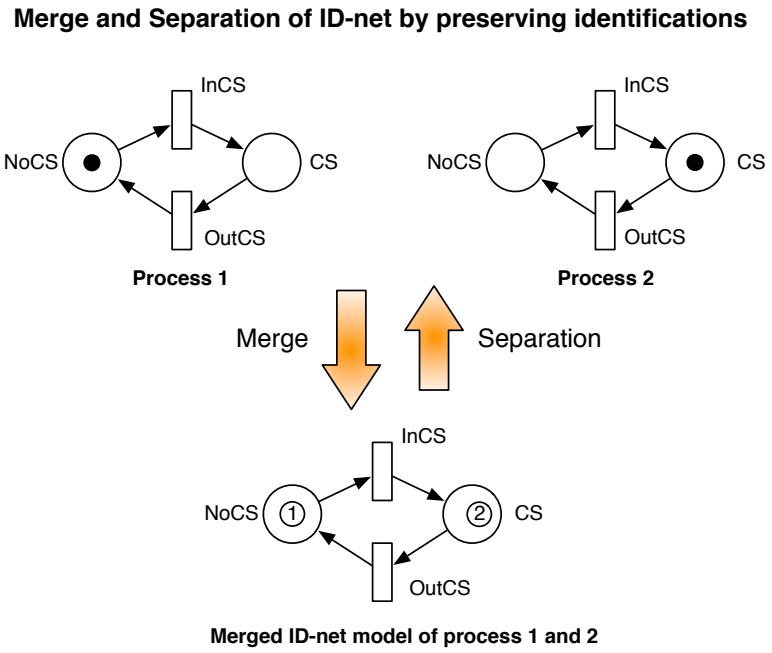


Figure 7.1: Merge and Isolation of ID-net Instances

in the system which cause operations to come about. It is a "shorthand" notation for micro-oprations.

For example:

- $[LOC]$ means the contents of the location LOC
- $R1 \leftarrow [LOC]$
- $R3 \leftarrow [R1] + [R2]$, where $+$ is a supported operation.
- $Condition :: R3 \leftarrow [R1] + [R2]$, where $Condition$ is a legal boolean expression, $+$ is a micro-operation.

Figure 7.2 shows the role of RTN in general programming language. It is an intermediate language between high-level programming languages and parallel hardware implementation. In addition, we think that RTN-alike notation is a good annotation language for ID-net, based on its following characteristics:

- RTN is a DSL which provides primitives (atomic micro-operations) to command hardwares, i.e. control accumulators, read/write registers and memories. These micro-operations can be executed by hardware in real parallel.

- Semantics of high-level programming languages are based on these primitives.
- An instruction of microprocessor is an atomic action, and it is a composition of micro-operations. Basic composition operators are parallel, sequence, and selection. Based on parallel execution of micro-operations, multiple instructions can be executed in parallel.
- Micro-operations command hardwares to carry out data computation. They are interested in the locations of data but not the meaning of data.

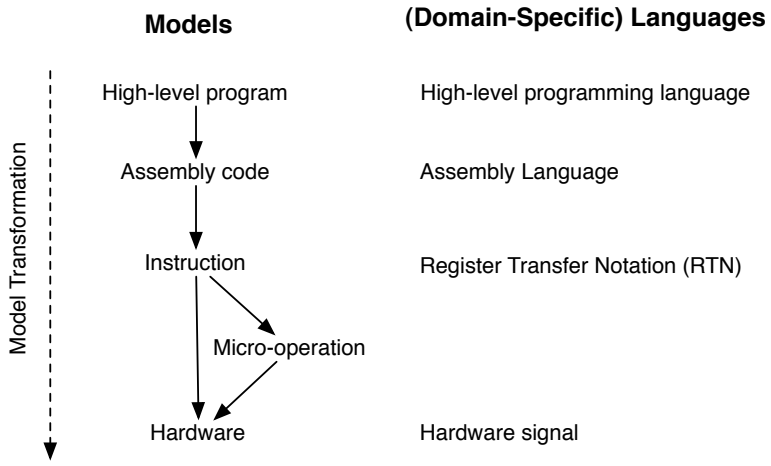


Figure 7.2: Models and Languages in General Programming

In our discussion, we use a RTN-alike notation to express how ID-net exchange resource references with its co-models. For the sake of simplicity, except mentioned explicitly, in followed examples we suppose that the used variables have the same type.

7.1.1 Control Token

Control token is one of the most used types of tokens used in PN modeling. A typical application is for *non-sequential processes*. For example, a non-sequential processe illustrated in figure 7.3 *. The inscription language used in this example is very simple, it has the following instruction set:

*This example comes from Wolfgang Reisig: Petri Nets: An Introduction. Chapter 1 Introduction.

Instruction	Description
read x $\Rightarrow$ x $\leftarrow$ [INPUT]	Read a value from input to variable
write y $\Rightarrow$ OUTPUT $\leftarrow$ [y]	Write a variable value to output
add1 x $\Rightarrow x \leftarrow [x] + 1$	Addition of 1
sub1 x $\Rightarrow x \leftarrow [x] - 1$	Subtraction of 1
ge x,0	Test if x is greater than 0
leq x,0	Test if x is less than or equals 0

Based on this instruction set, the non-sequential process of figure 7.3 takes two inputs x and y, and compute $x + y$. Each transition corresponds to an instruction. Variables used in the inscriptions are global variables, i.e. all transitions in this model can access them.

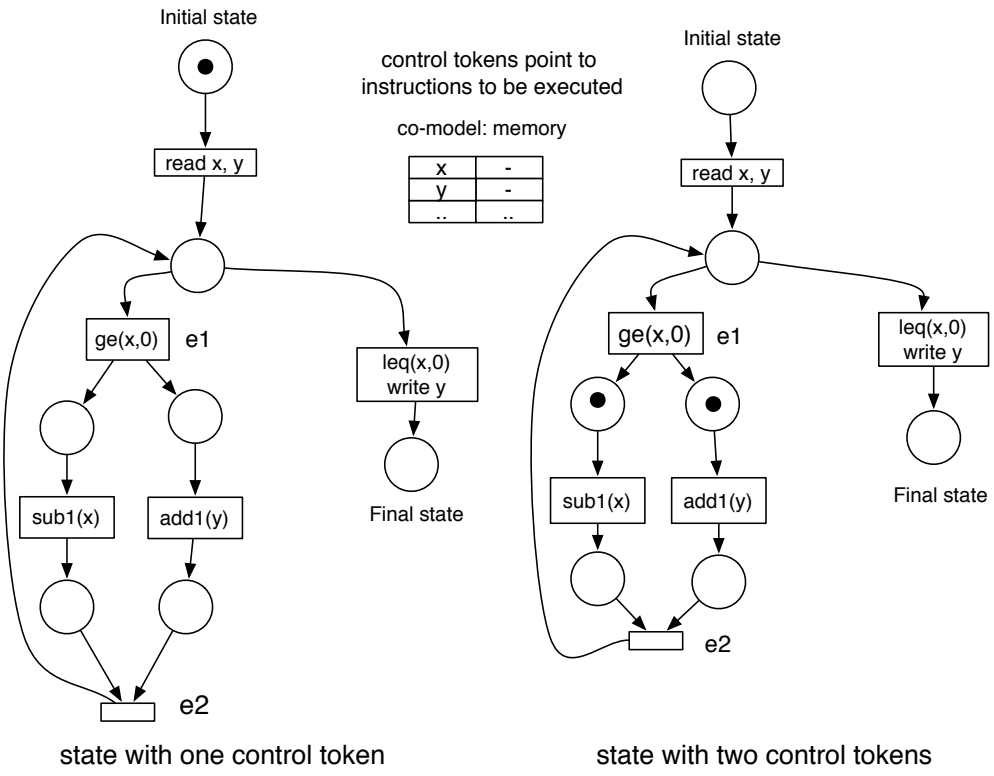


Figure 7.3: An Example of using Control Token for Non-sequential Process

In a non-sequential process, transitions are enabled by control tokens, the existence of simultaneously enabled transitions means the possibility of parallel processing. Concrete execution of non-sequential processes depends on the availability of execution units, or the strategy of scheduler which distributes enabled tasks to the execution units.

Figure 7.4 gives another example of using control tokens. Obviously, this example uses a more flexible inscription language, which allows doing addition and multiplications directly on global variables:

Instruction	Description
add z, x,y	Add x to y, and put result in z
mult z, x,y	Multiply x by y, and put result in z

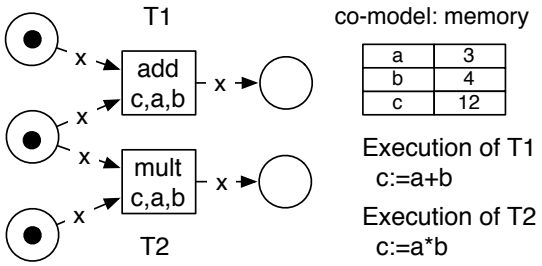


Figure 7.4: Another Example of Control Token with Inscription using Global Variables

Black tokens are indistinguishable. In figures 7.3 and 7.4, transition inscriptions are written using the syntax of the inscription language.

Control tokens have no knowledge about the data to be processed in the model, they only enable transitions. An *execution unit* will execute the inscriptions of enabled transitions as an atomic action. A system may have several execution units and each execution unit works on one atomic action at a time.

If several transition inscriptions which read and modify the same global variable are enabled simultaneously, a concurrent problem such as race condition may occur - after firing the transitions, the value of variable can not be determined because of non-determinism of concurrent systems. The example of figure 7.4 is a situation of *confusion*: T1 and T2 are *enabled* simultaneously, and we don't know which one will be executed.

7.1.2 Simple-Variable Token

Figure 7.5 gives an example of using simple variable tokens with ID-net, it uses the same inscription language as in figure 7.4. Tokens are names or addresses of variables, which can be global or temporal (local). Unlike global variables which are accessible by all the transitions, a temporal variable corresponds to an input arc of a transition, it is created when the transition is going to be executed and only accessible for this transition during this execution.

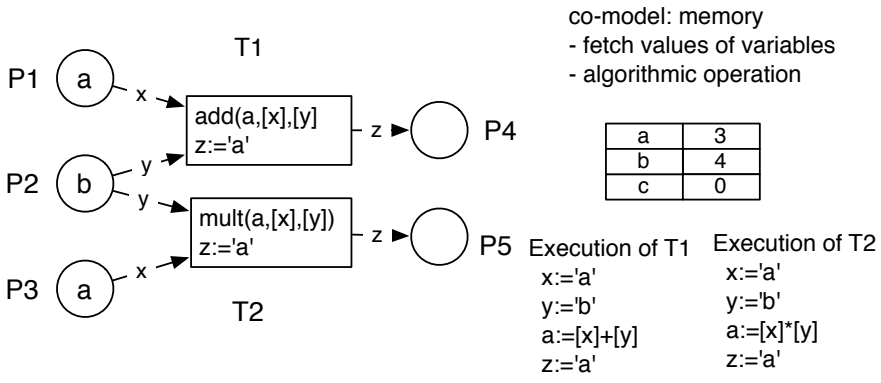


Figure 7.5: An Example of using Simple-variable Tokens

7.1.3 Resource Token

This is the most general case we can meet, where tokens are referring to structured data, e.g. objects having attributes and references to other objects, or a token is representing a collection of tokens. In these cases, previous approaches are not suitable because the relation between objects can be complex and the ways of interacting with the objects are various. Similar to using control token and simple-variable tokens, concurrent access of resource is an issue but can be detected.

The concept of Subject-Oriented Programming (SOP) [96] can be applied in this kind of situations: building the system as compositions of *subjects*, extending systems by composing them with new *subjects*, and integrating systems by composing them with one another. Each subject ensures its responsibilities by providing services, e.g. managing a specific kind of resources.

ID-net can be considered as the composition mechanism from SOP's point of view. Services of subjects are attached on transitions (as inscriptions) and invoked when transitions are executed. In concrete implementation, the design pattern *facade* maybe used in order to reduce the coupling between ID-net model and the set of *subjects*.

In ID-net, places and tokens are typed, tokens are resources identifications, and transitions specify which type of input resources are needed to produce output resources. "A token of type X" means a token (variable) refers to a resource of type X. For example, in figure 7.6:

- A, B, C are types of resources accepted by P1, P2, and P3, respectively.
- Each type of resource is managed by a manager, which provides services to create, modify, and dispose the instances of resource.

- Initially, place P2 has a token b1 of type B.
- T1 will produce token of type A in P1
- T2 will take both token of type A from P1 and token of type B from P2, then produce token of type C to P3 and produce token of type B to P2.
- T3 will consume token of type C from P3.
- T1 interacts with resource manager A, T2 interacts with resource manager B and so forth.
- A transition should provide an ID token while invoking the services of a manager.
- A resource manager can be any kind of system which provides reusable services.
- Dependences and interactions may exist between the resource managers. For example, in figure 7.6 manager A may use the services of manager B in some cases.

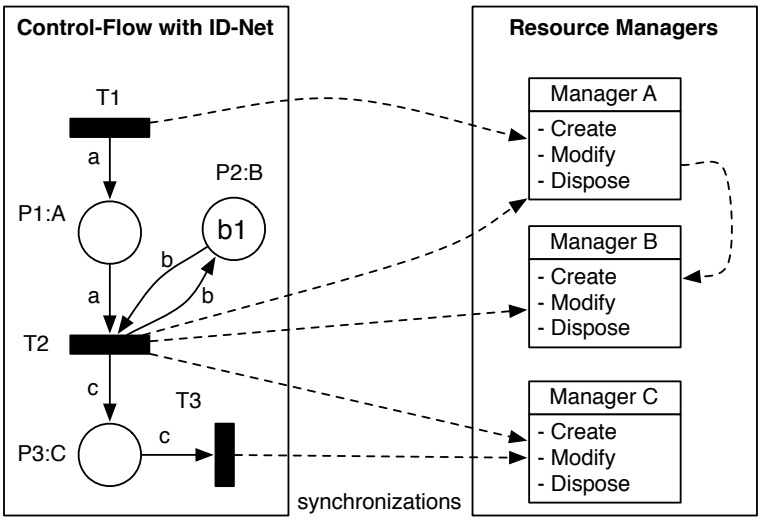


Figure 7.6: Separation of Concerns with ID-net

A priori the actions consume/produce tokens have no relation with the creation/disposal of resources. ID-net holds the identification tokens and has no knowledge about how to use and manage the resources, until the synchronizations between the ID-net model and the resource managers are given.

7.1.4 Use Simple-Value Token in ID-net

ID-net can use constant values as tokens, e.g. putting values such as *"0,1,2"* or *"true, false"* into places. In an ID-net using simple value tokens, tokens are referring to values of some basic, unstructured primitive types such as boolean, integer, etc. For structured objects such as lists and records, resource tokens should be used.

Store Simple Values in ID-net Place. A place which gives and receives reference tokens (variable names) of a data type does not have the same type as a place which gives and receives values of the same type. Thus an ID-net place can be either *reference-typed (containing reference tokens)* or *value-typed (containing simple value tokens)*.

In ID-net, the value of tokens are bound to temporal variables of the inscription function for computation. Each transition inscription is similar to a function, and places are stores for computation results. This approach allows to model data flow during computation, as shown in figure 7.7. This model can be used to specify highly parallel data processing, because no global variables are used and all data dependencies are modeled by the control structure.

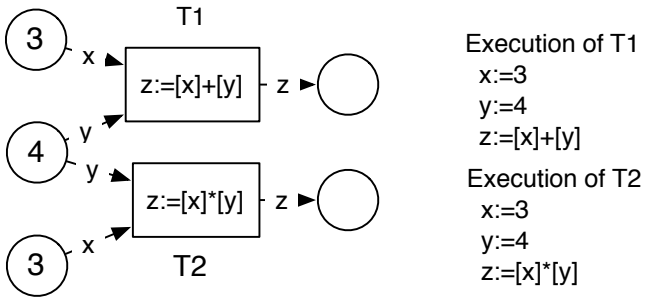


Figure 7.7: An Example of using Simple-value Tokens (without Global Variables)

Depending on architecture of target platforms, ID-net using simple-value tokens can be easily transformed into parallel executable codes.

7.2 Develop Parallel and Concurrent Programs with ID-net

Modern processor architectures have embraced parallelism as an important pathway to increased performance. Facing technical challenges with higher clock speeds in a

fixed power envelope, Central Processing Units (CPUs) now improve performance by adding multiple cores. Graphics Processing Units (GPUs) have also evolved from fixed function rendering devices into programmable parallel processors. As todays computer systems often include highly parallel CPUs, GPUs and other types of processors, it is important to enable software developers to take full advantage of these heterogeneous processing platforms.

OpenCL (Open Computing Language) is a language for programming heterogeneous data and task parallel computing across GPUs and CPUs. Conceived by Apple Inc., OpenCL's execution model has the following elements:

- Compute kernel. Basic unit of executable code, similar to a function. Compute kernels can be data-parallel or task-parallel.
- Compute program. Collection of compute kernels and internal functions.
- Applications queue compute kernel execution instances.
- Define N-Dimensional computation domain. Each independent element of execution in N-Dimension domain is called a work-item. The N-Dimension domain defines the total number of work-items that execute in parallel - global work size.

The instances of compute kernels, i.e. work-items, are distributed and executed as far as possible by compute devices, such as CPU, GPU, and DSPs. A work-item can also specify which compute device it will run on. Each compute device has several compute units which can work in parallel and share memories.

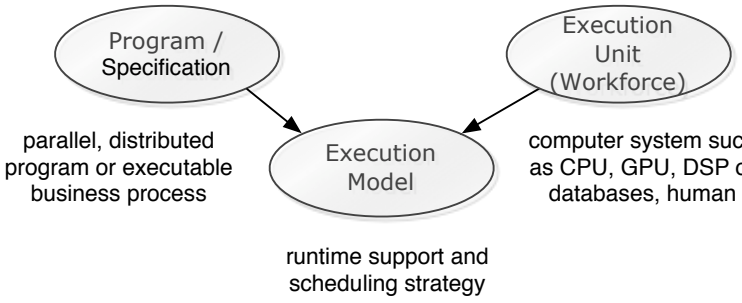


Figure 7.8: Program, Execution Model, and Execution Unit

Many formalisms exist for the modeling and verification of parallel programs, for example, pi-calculus and Petri nets allow to prove behavioral equivalences of processes and detect deadlocks in a system, respectively.

When we compose an *ID-net* with other models, i.e. associating the *inscription functions*, the semantics of inscription functions are merged with the semantics of ID-net transition in the result model. The places of ID-net are the "*keys-holder*" and the keys are used to find or retrieve necessary information or resources to achieve a task. The values of tokens are *assigned* and *used* by external inscription functions, but the *access* of these values are controlled by the ID-net model.

7.2.1 ID-net's Execution Model for Concurrent Programming

As presented in Chapter 5, ID-net has *true concurrency semantics*, which means *all enabled transitions can be fired simultaneously*, the basic execution cycle of an ID-net engine is:

1. establish an initial marking,
2. **fire all eligible transitions as possible**,
3. repeat step 2 until no more transition is eligible.

Depending on the environment and the platform of execution, there are different possible scheduling strategies of doing step 2. One criteria to determine if a strategy is valid or not is the order of firing simultaneously enabled transitions (of "free-choice" IDCS) should not affect the result of execution [†], i.e. "free choice" between these transitions. For example, in an operation system with many concurrent processes, different scheduling strategies should not affect the results of process executions, if a scheduler satisfy the properties such as fairness and no starvation.

ID-net's execution model for concurrent programming plays the role of coordinating programs and execution units of platforms. It has objectives such as:

- ensure concurrent and parallel execution of ID-net while respecting its semantics, e.g. true concurrency semantics, concurrence on tokens (resources), transactional execution of ID-net transitions.
- support configurable scheduling strategies for different purposes: interleaving or parallel simulation, local execution, and distributed execution of ID-net models. In general, the scheduling strategies should not influence the results of execution.

[†]Depending on the ID-net and its co-models, the free-choice property of an IDCS might be undecidable.

- integrate the execution of external models (synchronization) by providing and verifying input and output parameters.
- allow to load, unload, activate (ready to be executed), and deactivate ID-net models.

Some properties of IDCS should be verified before execution (or being scheduled for execution):

- **free-choice** of an IDCS, i.e. the order of firing the ID-net's transitions does not influence the results of execution.
- **termination**. If an IDCS will terminate eventually, e.g. no deadlock or live-lock. Some reactive systems are running in live-lock mode.
- **deadlock-free**. If an IDCS has deadlock, it should not be scheduled.

The above properties are general properties of IDCS with the *idealized true concurrency semantics*, i.e. transitions can be fired (executed) when they are enabled. Because the number of execution units is limited in concrete environments, the execution of the IDCS depends on the platform's scheduling strategy.

The **schedulability** of an IDCS for a concrete platform means: if there exists at least one valid schedule of the ID-net for a particular environment or configuration of execution units. For example, an IDCS may be executed correctly in an interleaving way but parallel execution may cause race condition problems.

7.2.2 An Example of DSL Composition

Figure 7.9 show an APN model which computes the Fibonacci number for a given value in place *P2*. Tokens in places $P1$ and $P2$ can contain integers. The computation will be finished when no transitions are firable and the result will be the value of the last token in *P1*.

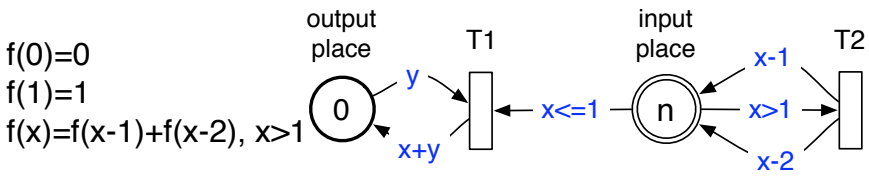


Figure 7.9: APN Model for Fibonacci Number: Tokens are Values

In the APN model, annotation on the arcs are algebraic terms containing variables, variables are implicitly temporal and local to the transition. The rules of building terms are given by axioms of algebraic abstract data types.

Figure 7.10 show an ID-net model which does the same work. Similar to APN, ID-net tokens and rules of manipulating referred resources are defined outside of the control structure; ID-net arc inscriptions only contains variable names; ID-net transition inscriptions are in fact synchronizations between ID-net transitions and actions from external models. In this example, x, y, z are local temporal variables used by the transitions. Place $P1, P2$ can only contain integers; $[x], [y], [z]$ are evaluation of variables which return integer values.

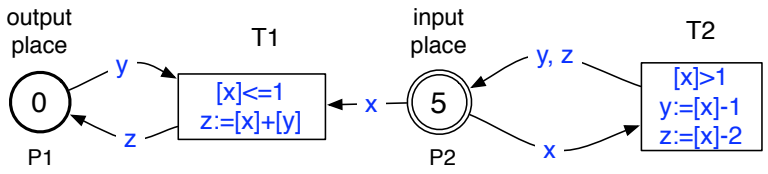


Figure 7.10: ID-net Model for Fibonacci Number

Transform Models to Platform Specific Models (or Code Generation). The above APN and ID-net models are Platform Independent Models. In model-driven development (comparing with only modeling and verification of systems), a further necessary step is to transform them into executable Platform Specific Models or give the models operational semantics. We believe that at this stage ID-net has advantages over other formal models by means of the Separation of Concerns, and its synchronization mechanism.

In this subsection, we will show how ID-net models can be transformed into executable codes for a specific platform.

We use a simple microprocessor as the target system for the transformation (or code generation) of this ID-net model. It has 1 accumulator (Acc) and 8 registers (A-H). It supports several basic instructions and two data types *Integer* and *Boolean*. For the sake of simplicity, we only consider the state of accumulator and the registers and only minimal set of instructions are explained. The idea is to show how annotated ID-net models can be transformed into imperative, parallel executable codes.

The microprocessor can be described as the following state-transition system:

- *Acc* is the accumulator
- $R = \{A, B, C, D, E, F, G, H\}$ is the set of registers

- INT and $BOOL$ are the data types supported by the registers
- The state of the system $S = \{R \cup \{Acc\}\} \times (INT \cup BOOL)$
- Initial state $S_0 = [Acc = nil, A = nil, B = nil...]$, where nil means the register is free
- $S[reg]$ returns the values of reg at state S; $S[reg = val]$ means reg has value val at state S
- Semantics of its operations:

$$LOAD(reg) : \frac{S[reg] = val, val \in (INT \cup BOOL)}{S \to S[Acc = val]}$$
$$STO(reg) : \frac{S[Acc] = val, val \in (INT \cup BOOL)}{S \to S[reg = val]}$$
$$ADD(val) : \frac{S[Acc] = x, x, val \in INT}{S \to S[Acc = x + val]}$$
$$ADD(reg) : \frac{S[Acc] = x, S[reg] = y, x, y \in INT}{S \to S[Acc = x + y]}$$
$$SUB(val) : \frac{S[Acc] = x, x, val \in INT}{S \to S[Acc = x - val]}$$
$$SUB(reg) : \frac{S[Acc] = x, S[reg] = y, x, y \in INT}{S \to S[Acc = x - y]}$$
$$TEQ : \frac{S[Acc] = 0}{S \to S[Acc = 1]}, \frac{S[Acc] \neq 0}{S \to S[Acc = 0]}$$
$$TLT : \frac{S[Acc] < 0}{S \to S[Acc = 1]}, \frac{S[Acc] \not< 0}{S \to S[Acc = 0]}$$
$$TGT : \frac{S[Acc] > 0}{S \to S[Acc = 1]}, \frac{S[Acc] \not> 0}{S \to S[Acc = 0]}$$

The microprocessor operates with its registers and accumulator: it takes operants from its registers, computes result and put it into the accumulator. A program for this processor can be seen as a process constructed using these instructions. Multiple processors may work together by using shared registers (or memories) and communication channels.

As discussed earlier in this chapter, value tokens in ID-net are implemented by reference tokens. There are several possible implementations and we will not discuss the details here. In this example, because all intermediate results are stored in the registers, during execution the system should know where (i.e. in which register) an intermediate value is stored, for example, the value tokens $'0', '5'$ in figure 7.10 are

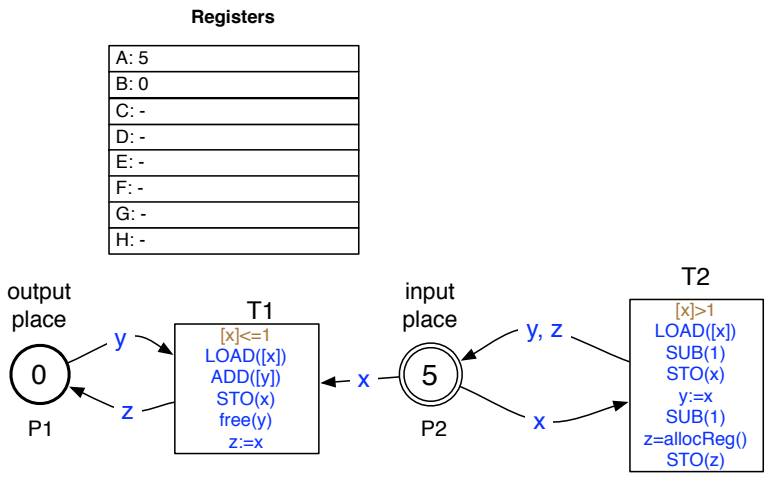


Figure 7.11: Annotated ID-net using Instructions of the Microprocessor

can be referred by B and A (as shown in figure 7.11). For all the other intermediate values, a register is allocated to store a value in a transparent way.

Due to limited capacity of registers, we have to do register allocation while executing high-level models or transforming them into executable machine codes. Thus, a supplementary state and two actions are used to manage the available registers:

R_{nil} is the set of free registers, $R_{nil} \subseteq R$

$$allocReg(reg) : \frac{S[R_{nil}] = X, reg \in R_{nil}}{S \rightarrow S[R_{nil} = X - \{reg\}]}$$

$$free(reg) : \frac{S[R_{nil}] = X, reg \notin R_{nil}}{S \rightarrow S[R_{nil} = X \cup \{reg\}]}$$

Both actions are atomic. *allocReg(reg)* fails when no free register is available, i.e. when $R_{nil} = \phi$. *free(reg)* never fails.

The target model is an implementation of the composed source models using the target platform's language and runtime supports. We suppose that the platform has runtime supports to execute ID-net and do register allocation.

Three models are involved in this example:

- *Source model*: the annotated ID-net model, to compute Fibonacci number
- *Source model*: the register allocation model, as platform's runtime support (e.g. libraries)

- *Target model*: the executable program for this microprocessor

The final system respects the constraints of both initial systems, i.e. the PN model and the register allocation model. For example, with initial state: P1(B→0), P2(A→5), free registers: C, D, E, F, G, H, an non-optimal firing sequence could be:

```
  T2: P1(B->0),P2(A->3,C->4)
  T2: P1(B->0),P2(A->1,D->2,C->4)
  T2: P1(B->0),P2(A->1,D->0,E->1,C->4)
  T2: P1(B->0),P2(A->1,D->0,E->1,C->2,F->3)
  T2: P1(B->0),P2(A->1,D->0,E->1,C->2,F->1, G->2)
* T2: P1(B->0),P2(A->1,D->0,E->1,C->2,F->1, G->0,H->1)
  T1: P1(B->1),P2(D->0,E->1,C->2,F->1,G->0, H->1)
  T1: P1(B->1),P2(E->1,C->2,F->1,G->0,H->1)
  T1: P1(B->2),P2(C->2,F->1,G->0,H->1)
  T2: P1(B->2),P2(C->0,D->1,F->1,G->0,H->1)
  T1: P1(B->2),P2(D->1,F->1,G->0,H->1)
  T1: P1(B->3),P2(F->1,G->0,H->1)
  T1,T1,T1: P1(B->5), P2()
```

After the step marked by $*$, transition $T2$ becomes not firable because there are no free registers. By firing $T1$, the register allocation mechanism frees the unused registers, thus afterwards $T1$ becomes firable again.

Transform ID-net Model into SQL with Transactions In the other words, ID-net can be implemented by SQL supporting transactions. For example, figure 7.12 shows the SQL transformation of ID-net model of figure 7.10.

In this transformation, an ID-net is transformed into a SQL table: each ID-net place is transformed into a column of the table; each ID-net transition is a procedure (ACID transaction) which manipulate the values of the columns corresponding to the transition's input and out places; conditions are checked in the procedure; the arc inscriptions are the input and output parameters of the procedures. The transactional execution of the procedure is guaranteed by the database management system.

7.3 Business Process Modeling with ID-net

A business process consists of a set of activities, the order of finishing these activities are determined by the dependences between them. Similar to parallel programing,

```
create table Fibnum (
 P1 integer,
 P2 integer
) engine = innodb;

insert into Fibnum(P1) values(0);
insert into Fibnum(P2) values(5);
delimiter //
```

```
create procedure T2()
begin
declare x, y, z INT;
declare cur cursor for select P2 from Fibnum where P2>1;
START TRANSACTION;
open cur;
fetch cur into x;
set y=x-1;
set z=x-2;
delete from Fibnum where P2=x limit 1;
insert into Fibnum(P2) values(y);
insert into Fibnum(P2) values(z);
select x,y,z;
close cur;
COMMIT;
end //
```

```
create procedure T1()
begin
declare x, y, z INT;
declare cur1 cursor for select P1 from Fibnum where P1>=0;
declare cur2 cursor for select P2 from Fibnum where P2<=1;
START TRANSACTION;
open cur1;
open cur2;
fetch cur1 into y;
fetch cur2 into x;
set z=x+y;
delete from Fibnum where P2=x limit 1;
delete from Fibnum where P1=y limit 1;
select x,y,z;
insert into Fibnum(P1) values(z);
close cur1;
close cur2;
COMMIT;
end //
```

Figure 7.12: Implementation of ID-net Model of Figure 7.10 in MySQL

arbitrary specifying orders between activities without respecting domain constraints will imply over-specification.

7.3.1 Example of post office workflow modeling

Figure 7.13 shows the control flow of ticket service at post office: when a client arrives at post office, she or he pushes a button and gets a ticket with a sequence number in order to be served. For each push action of *PushButtonNo1* and *PushButtonNo2*, the system prints out a time-stamped ticket with a sequence number. When an employee is available, the system will display the sequence number to be served and the number of window the client should go. The sequence number should be unique during a period.

First phase: modeling control-flow. The ID-net model shows how the tickets are created, stamped, and distributed to working windows. A counter object is used to stamp the tickets one-by-one to ensure the uniqueness of sequence number. When we create this model, we know that:

- Transitions *PushButtonNo1* and *PushButtonNo2* are supposed to *create new tickets*

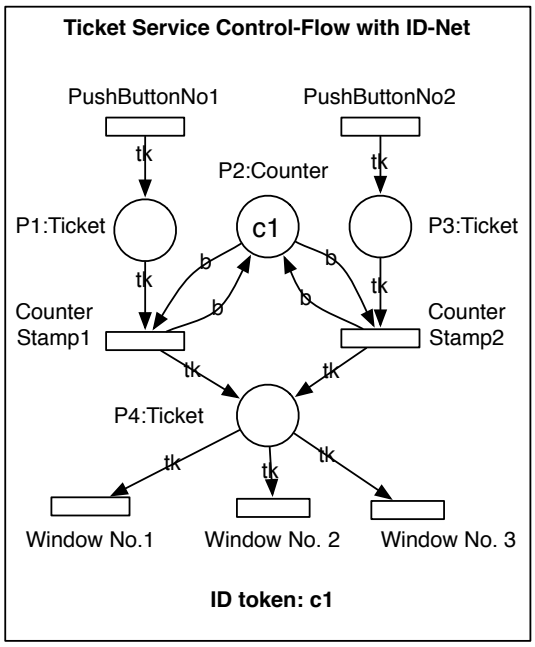


Figure 7.13: Ticket Service at Post Office: Control-flow

- Places P1, P3 hold tokens which refer to unstamped tickets
- P2 holds token **c1** which refers to the global counter
- Transitions *CounterStamp1* and *CounterStamp2* are supposed to *stamp the tickets a sequence number.*
- Place P4 holds tokens which refer to stamped but not served tickets.
- Transitions *WindowNo1*, *WindowNo2*, and *WindowNo3* are supposed to *change the status of unserved tickets, and display the corresponding window number.*

For each input arc and output arc of a transition, we annotate it with a variable name. The scope of an instance of a variable is limited to an execution of a transition, i.e. accessible only by its inscription function during that execution.

Until now, the control-flow in figure 7.13 is abstract because the intentions above are not precise enough to be operational.

Second phase: identifying data objects and the services of managing them. The type of tokens are already identified in first phase, i.e. P1, P3, P4 contain tokens referring to **Ticket** objects and P2 holds the token referring to a

global **Counter**. Also, according to the informal description of the first phase, following services are necessary:

- create new tickets. E.g. tk=Ticket.Create(), where tk is a variable of type Ticket.
- stamp the tickets with time and sequence number. E.g. tk.stamp(seq), where seq=c1.seq().
- change the status of unserved tickets to served, and display the corresponding window number. E.g. tk.setStatus("served"); LED.display(ticket).

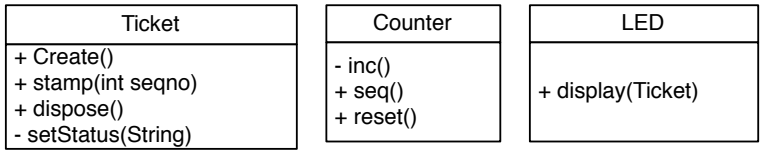


Figure 7.14: Identified Objects and Services of the Business Process

If not existed, these objects and services should be implemented. At this stage, we apply subject-oriented analysis and dynamic behaviors of these objects in the workflow are not specified.

Third phase: write synchronizations (ID-net inscriptions). Basically, each transition is synchronized with one or several external functions, where each function interacts with a specific model. An inscription function takes the values of input variables from the transition's input arcs and returns output values to output arcs of the transition. The whole transaction consists of synchronizing a transition with all its synchronized functions, it is atomic and has short-duration. How transactions can be implemented depends on the concrete environment.

After specifying synchronizations, all the transitions will run in parallel and they interact via the places and ID tokens. In this example, the counter **c1** is the concurrent object which is used to stamp the tickets each time.

7.3.2 Transaction Issues in Business Process Modeling

Execution of ACID transaction should be guaranteed. Each transition of an annotated ID-net is an ACID transaction, all of them constitute a concurrent system. For business processes, because the supports of executing transaction depends on

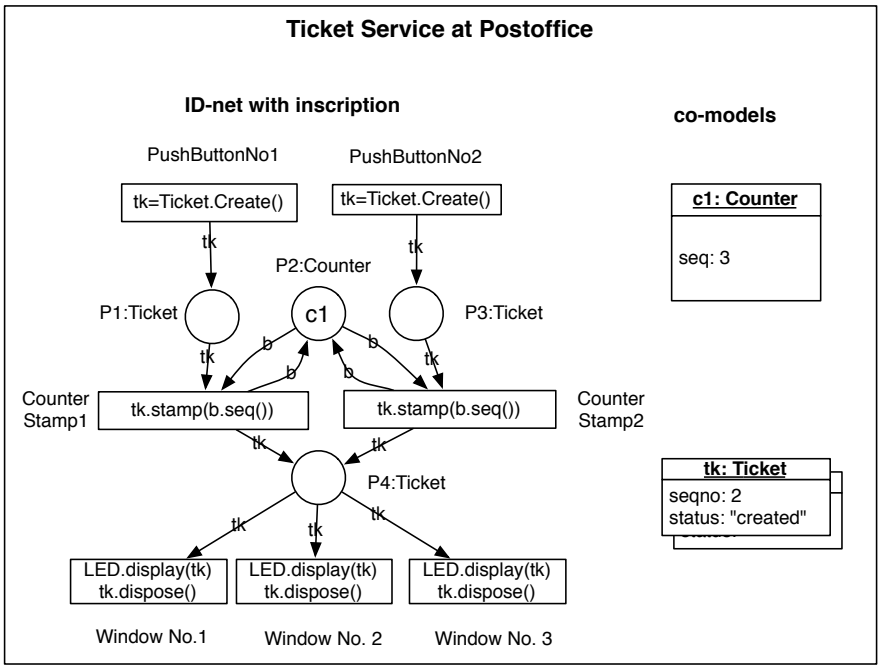


Figure 7.15: Synchronization of Models via ID-net Inscriptions

concrete environment it is possible to manage the execution of transaction in two ways:

- If the execution of transaction is supported by the underlying system, e.g. in a database supporting transactions or web services supporting transactions. In this case, we just transform ID-net transitions into transactions.

- Otherwise, if we are using a language or platform (e.g. Java) without the mechanism of executing transactions. It is possible to provide a mechanism to guarantee the transactional properties during the code generation phase.

Long-Running Transaction. For long-running transaction, the states of transactions are managed explicitly in the process model, i.e modeled by the ID-net or its co-models. Most of the time, the aspect of transaction can be handled separately by a co-model.

7.4 Service Component Model

We will present how to build reusable building blocks by means of encapsulated ID-net controlled systems. The idea of encapsulating ID-nets is similar to CO-OPN contexts, but here the encapsulation is relaxed and we distinguish the control model (ID-net) and the data or resource models (co-models).

Analogously as *component* in Service Component Architecture (SCA) [97], *Service Components (SC)* are configured instances of implementations and they provide and consume services. More than one component can use and configure the same implementation, where each component configures the implementation differently. A system can be specified as the composition of different service components. *Service Component Model (SCM)* is the model of service components, including the composition mechanism.

Service Component. A service component has at least one *service*, zero or more *properties*, and zero or more sub-components. A normal service component hides its internal sub-components. A service component can be *open*, which means that the services of its sub-components are visible and accessible by other components.

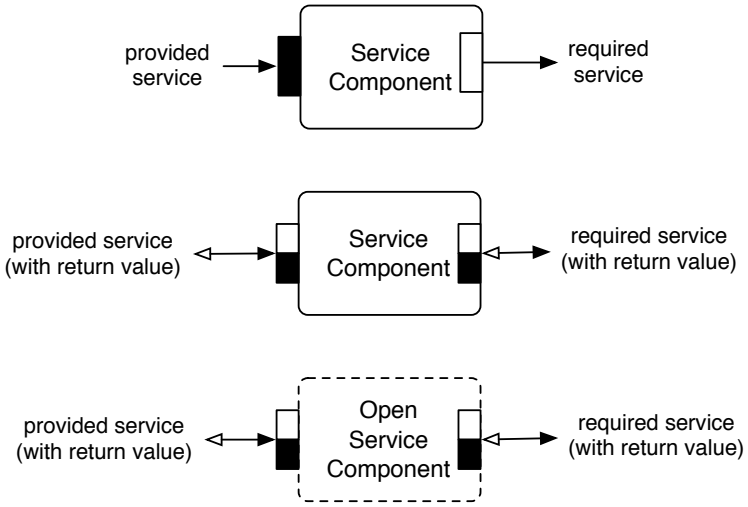


Figure 7.16: Service Components

As depicted in figure 7.16, a SC is represented graphically by a rectangle with round corner. A black box attached outside the SC represents a provided service; a white box attached inside the SC represents a required service, i.e. output events or dependencies of the SC. An open SC is represented by the same notation but with dashed line. As a short-hand notation, a black-white box represent a service which

has return values synchronously. The black triangle of an arc indicates the direction of synchronization (call), and the white triangle of an arc indicates the direction of returning value.

Service. A service is an interaction point between the component it belongs to and other components. Each service has a signature indicating the service's name as well as the types and names of its parameters. The invocation of service implies synchronization between service components.

Property. A property of SC is *observable* without inferring its normal behaviors. SC properties have initial values and they are manipulated by the SC during execution. An SC property is typed and its manipulation can be protected by a *concurrent data object*, e.g. a data object may support simple read/write operations or conditional atomic primitives such as compare-and-swap to manipulate the property.

Service components can be composed via the service ports, as shown in figure 7.17. While composing the services, the type of services should match each other, i.e. ensure the input/output of the service parameters.

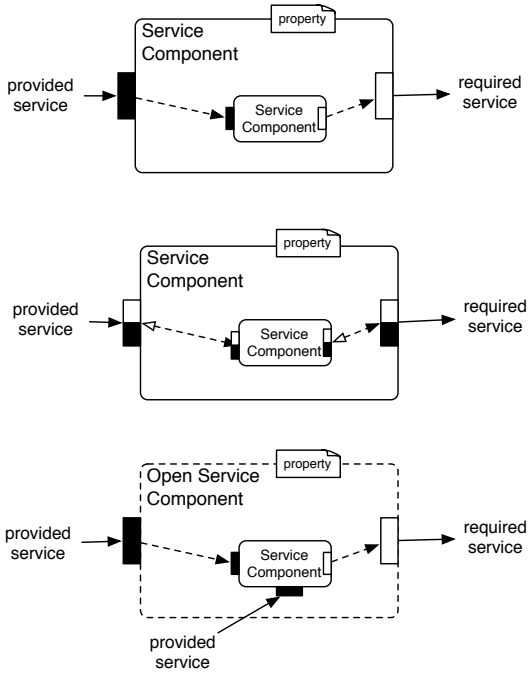


Figure 7.17: Composition of Service Components

7.4.1 Semantics of Service Component Model Based On ID-net

Service component model is a general concept and it can be implemented by encapsulated ID-net and its co-models: services are visible ID-net transitions allowing to access and manipulate the properties; properties are data or resources specified by the co-models.

Concurrent Data Object A concurrent data object consists of a data structure and a set of atomic operations to manipulate the data structure. Example of concurrent data object are registers which support *read/write*, *fetch-and-add*, *read-and-update*, *test-and-set*, *compare-and-swap* operations. In parallel computing, these atomic operations are often implemented by hardware such as microprocessors or transactional memories. In business process management systems, the problem of concurrent data access can be managed by the database transaction mechanism.

With true concurrency semantics, the services of SC are concurrently accessible until the calls are propagated to the concurrent data objects, where concurrent operations are sequentialized. A SC has to ensure the *consensus* during the concurrent accesses of its services [98].

Concurrent data objects are "guards" on data to protect them from inconsistent concurrent accesses. e.g. similar to the role of lock and semaphore. The encapsulation of co-models will guarantee that the data or resources are observable from outside and manipulated exclusively by the service component, which can manage the concurrent accesses of resources.

A concurrent data object can be modeled as an SC. For example, figure 7.18 shows SC which implement atomic actions on registers. Here we use a safe Petri net place to represent the register, and use *read arc* to read the value stored in the place. In figure 7.19, we show that we can use atomic read/write register to implement *compare-and-swap* register (if the specified transaction can be implemented).

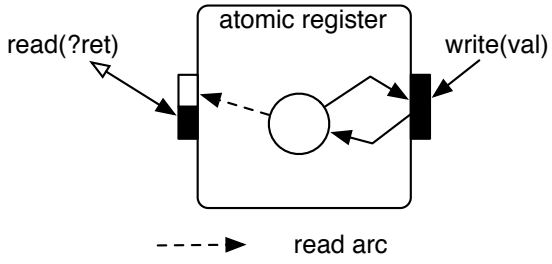


Figure 7.18: An Read/Write Atomic Register SC

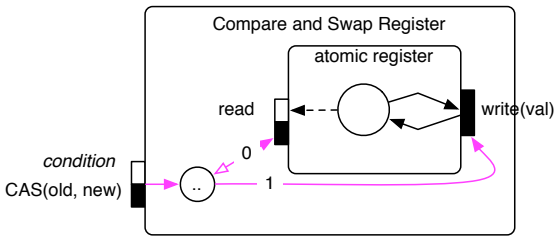


Figure 7.19: An Compare-and-Swap Atomic Register SC

7.5 Summary

ID-net is executable and its semantics implies the composition of domain-specific models. In this chapter, we show how to use different kinds of tokens to develop concurrent parallel programs and business processes and how to use encapsulation to create reusable Service Components with ID-net. The Service Components are supposed to be executed in concurrent environments.

The development of concurrent programs and business processes can be done by incrementally annotating ID-net with actions from data models. Each kind of model can be developed and maintained separately and the synchronization can be specified and tested at each iteration. The semantics of ID-net will ensure the parallel and distributed execution of the developed models.

Despite of its graphical representation, ID-net is a language for developers and it is not a user-friendly language for domain experts who will design high-level business process. However, we can adopt a standard graphical business process modeling language such as BPMN for the frond-end design. The semantics of this graphical language can be given by Service Components powered by ID-net and its co-models, thus the incremental development of business process will be performed on ID-net and its co-models.

Chapter 8

Applications and Case Studies

8.1 Introduction

In this Chapter, we will present the applications of our approach in real-world cases business process modeling and development. As show in figure 8.1, our approach follows the Model-Driven Architecture and the roadmap presented in Chapter 1.

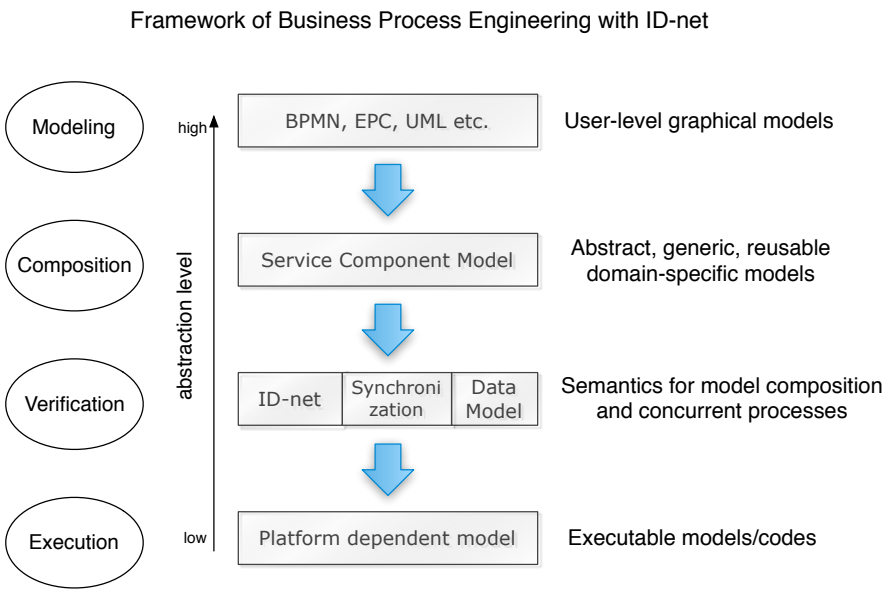


Figure 8.1: Business Process Engineering with ID-net

Concrete software projects are planned and realized on existing infrastructures (or platforms) of the execution environment. For example, some infrastructures will

ensure the non-functional requirements of the system and they can influence the development of an information system:

- database management system, e.g. relational databases and object-oriented databases.
- support of data persistence, e.g. automatic mapping between Object/Relational models in Java such as Hibernate.
- support of transaction, security, and authentication, e.g. in JavaEE application servers.
- support of business process management (BPM) and execution, e.g. business process language and execution engine.

More infrastructural supports means that we can create and execute models without managing some details, thus working at a higher abstraction level.

Granularities of Business Process Models The granularity of model for business process depends on the environment where the business process will be deployed into. Basically, we distinguish two granularities of business process modeling: in environment *with BPM supports* and *without BPM supports*.

With an environment with BPM supports, we use a business process modeling (or description) language to specify business processes. The business process modeling language is usually a high-level domain-specific language (such as BPMN, BPEL) which can be refined and transformed into executable codes or executed in Business Process Management Systems (BPMS). In this case, the development of business processes is limited by the expressiveness of the domain-specific language and its tools supports, i.e. if the language is not enough flexible and expressive to model domain problems, a developer cannot pass the limit of the language and he/she has to work around to solve the problem. Methodologically, we should avoid this kind of practices.

Moreover, with BPM supports, the user interfaces (e.g. Web interface) is another dimension problem than the business process model, which is modeled and managed by other infrastructures. An environment with BPM supports usually supports the dynamic changes and deployments of business processes.

Because currently BPMS solutions are not yet as widely accepted and standardized as Database Management Systems (DBMS), many business processes are implemented in an ad-hoc manner in development. In this case usually the system supports a fixed number of business processes which do not change frequently. For

example e-commerce web sites have several fixed business processes such as ordering, payments, and delivery.

In this Chapter we will present the applications of our approach to develop business processes in both environments: with BPMN and for Web-based applications.

8.2 Develop Business Process with Business Process Modeling Notation (BPMN)

We can develop business processes with BPMN in three steps:

1. domain experts create abstract business processes with BPMN graphical editors, BPMN diagrams are annotated with platform-independent information. The annotations can be considered as documentation and requirement analysis for the business processes.

2. based on the BPMN diagram and previous annotations, software developers create data models for the business processes, and create a formalized annotation layer for the BPMN diagram. The BPMN diagrams with formalized annotation and the data models are transformed into Service Components whose semantics are given by ID-net Controlled System (IDCS).

3. tools can check the properties of the Service Components, and transform them into executable models. The final executable model will be deployed into a BPMS.

Step 1-3 can be repeated, i.e. via incremental development, until the results satisfy the domain experts and software developers. We will present how to transform BPMN diagrams into encapsulated ID-nets and how to add data model and formalized annotations into the models.

8.2.1 Core Elements of BPMN

BPMN [13] was developed by the Business Process Management Initiative (BPMI) as a standard notation for business process modeling. The primary goal of the BPMN effort was to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform

those processes, and finally, to the business people who will manage and monitor those processes. BPMN will also be supported with an internal model that will enable the generation of executable BPEL4WS. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.

BPMN defines a Business Process Diagram (BPD), which is based on a flowcharting technique tailored for creating graphical models of business process operations. A Business Process Model is a network of graphical objects which are activities and the flow controls that define their order of performance. BPMN has four types of elements:

- Flow Objects: Event, Activity, Gateway
- Connecting Objects: Sequence Flow, Message Flow, Association
- Swimlanes: Pool and Lane
- Artifacts: Data object, Group, Annotation

Figure 8.2 shows the main elements of BPMN. Since BPMN is considered to support only concepts in business process, other concepts such as organizational structure, functions, and data models are out of the scope of BPMN. However, the flow of data, i.e. messages, and their association with activities, are represented in BPMN. More details on BPMN can be found in OMG's BPMN specification [13].

Figure 8.3 shows an example credit approval process in BPMN. Each pool or swimlane represents a process participant, which has internal activities connected by flow objects. In order to realize the business process, participants collaborate by exchanging messages. Messages are typed.

Formalized Annotations in BPMN In BPMN specification, BPMN annotations such as labels, texts, conditions, and data, are descriptive and informational, thus not formalized. THe formalization of BPMN annotations is necessary to make executable business processes, i.e. the annotations should be executable actions or operations defined somewhere as part of business processes.

Figure 8.4 shows a simple data model we can used to annotate the credit approval BPMN diagram. It has only one class *Request* and two enumeration types *Risk* and *Approval*. As many UML tools support currently, the data model can be transformed into executable codes allowing the creation and manipulation of data objects, e.g. generating Java objects with getter and setter methods. Based on this assumption, we will annotate the BPMN diagram with correct syntax and make the final model executable.

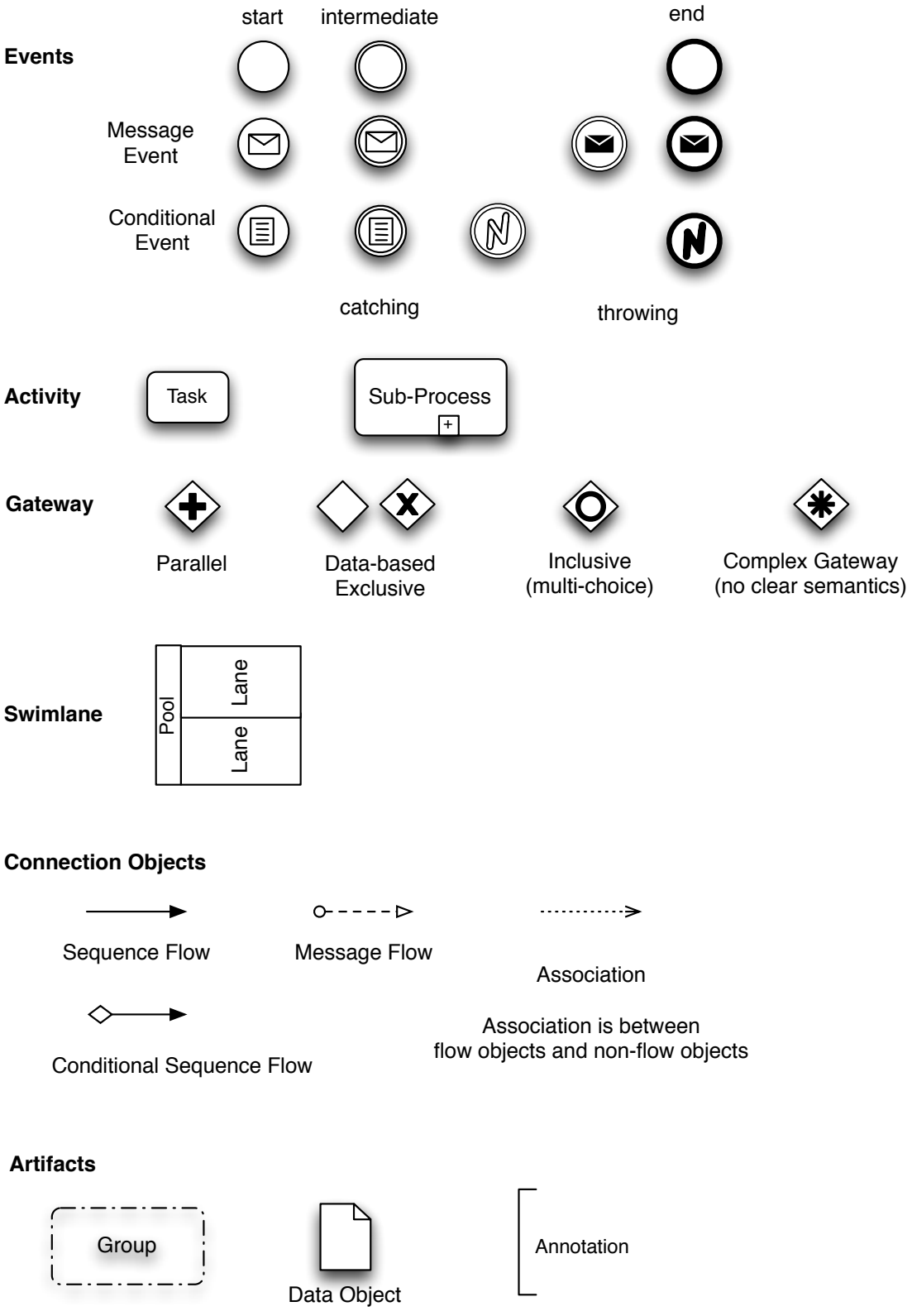


Figure 8.2: BPMN Elements

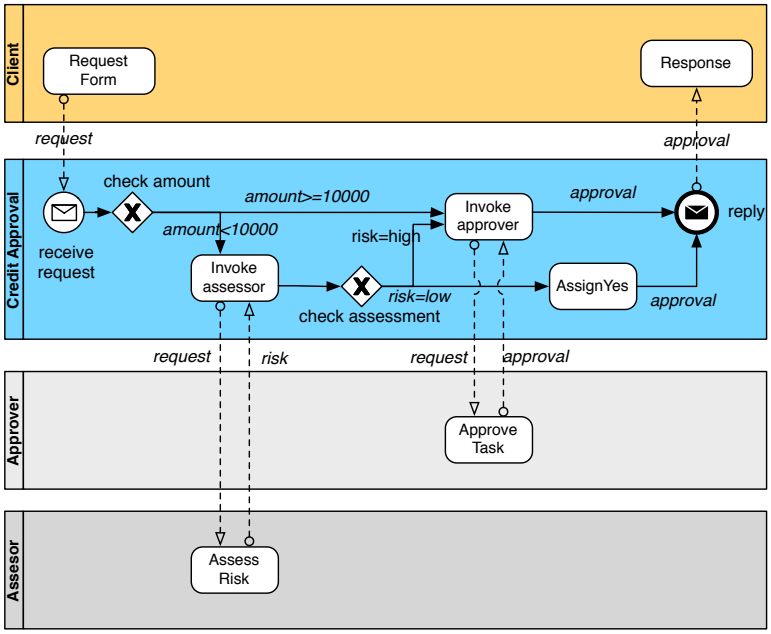


Figure 8.3: BPMN Diagram of Credit Approval Business Process

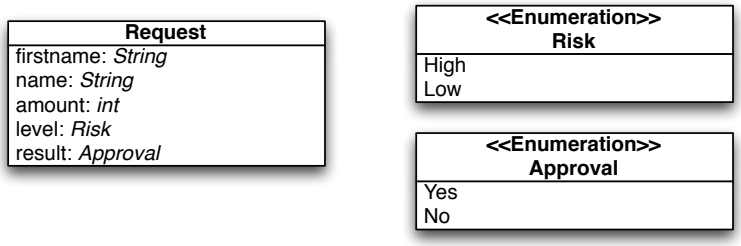


Figure 8.4: A UML Data Model for the Credit Approval Business Process

With this data model, the annotations of figure 8.3 can be formalized and concretized. For example the conditions can be concretized as follows:

- $amount >= 1000$ becomes $request.amount >= 1000$
- $amount < 1000$ becomes $request.amount < 1000$
- $risk = high$ becomes $request.risk == High$
- $risk = low$ becomes $request.risk == Low$

BPMN and Service Component A BPMN swimlane can be mapped to a Service Component, as shown in figure 8.5. Each swimlane interacts with others via the services. Messages exchanged in BPMN are mapped into parameters of the services and the internal behavior of an BPMN activity will be transformed into annotated ID-nets.

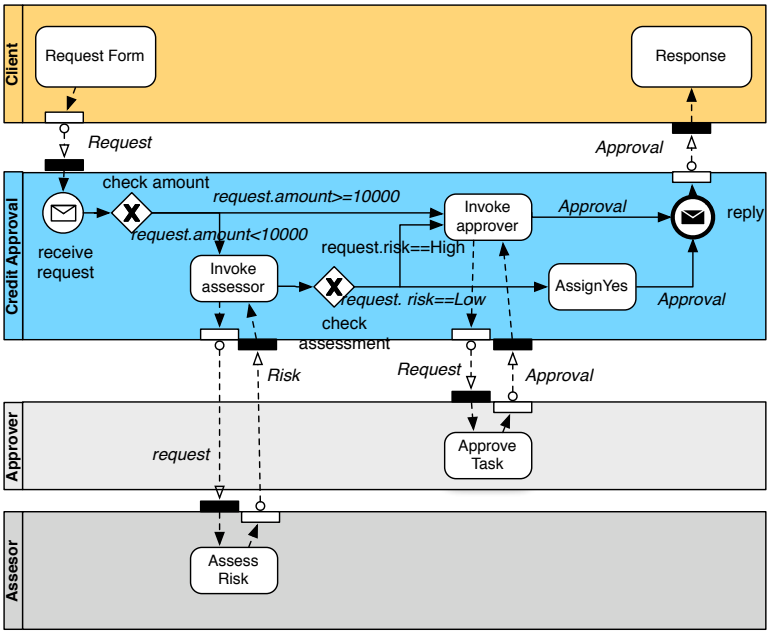


Figure 8.5: Credit Approval BPMN Diagram Represented as Service Components

8.2.2 Semantics of BPMN Control Flow

BPMN has precise semantics in terms of control-flow which is based on Petri nets. Transformation of BPMN control flow to Petri nets is quite straightforward and it

has been studied by [99] and [100]. We will present the techniques and rules for the transformation of BPMN to ID-nets. Please note that there are several possibility of transformation and it may depend on the data models. In our case the transformed ID-net is a Case-Handling System (CHS) and it is flexible and easy to integrate any data model.

First of all, the following rules are applied for the names and types of BPMN and ID-net elements:

- there is a single type for the process instances in ID-net, e.g. type of *Request* in our example.
- each BPMN message has its own type, a BPMN message is transformed into an ID-net place which may have a different type then the type *Case*, i.e. *message place*. The type of the message can be found in the data model.
- a *message* place is optional and it can be replaced by an event (i.e. external transition) which synchronizes with an ID-net transition.
- in order to be executable, the BPMN conditions should be legal logic expressions based on the data models.

We propose 13 rules to transform BPMN structures into ID-net structures, this transformation is in face a formalization of BPMN control flow.

Rule 1: Basic Events and Tasks Figure 8.6: Basic events (*start*, *intermediate*, and *end* events) and tasks are transformed into places, and their names are preserved as place name. A task with an output arc is transformed into a place and a transition with the same name with different prefixes, i.e. $P\_$ and $T\_$, respectively.

Rule 2: Flow and Conditional Flow Figure 8.7: Sequence flow and conditional flow are transformed into ID-net transitions. Flow conditions are preserved as transition conditions. The name of this kind of transitions are given during the transformation (anonymous transitions).

Rule 3: Conditional Start Event Figure 8.8: A conditional start event is transformed into an ID-net transition, its conditions are kept as the conditions of the ID-net transition.

Rule 4: Conditional Intermediate Event Figure 8.9: A conditional intermediate event is simply transformed into an ID-net transition, its name and conditions are kept as the same.

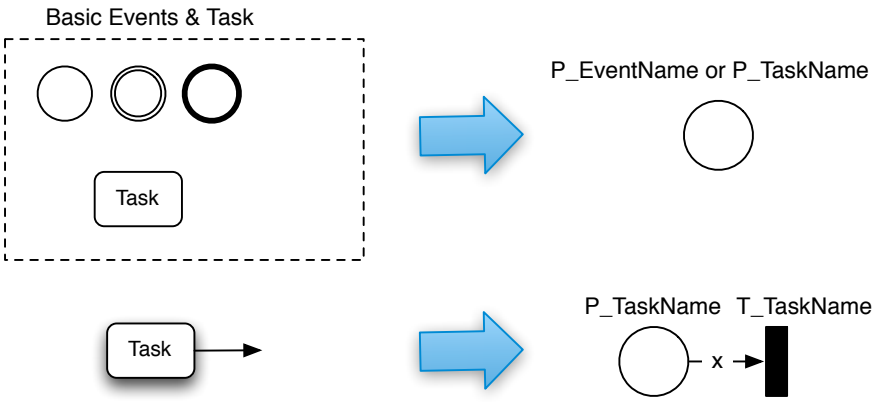


Figure 8.6: Rule 1: Basic Events and Tasks

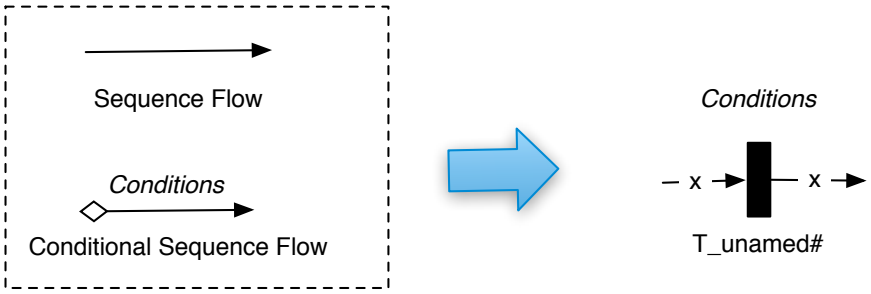


Figure 8.7: Rule 2: Flow and Conditional Flow

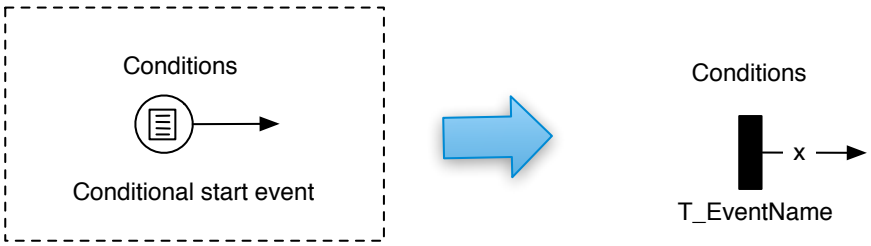


Figure 8.8: Rule 3: Conditional Start Event

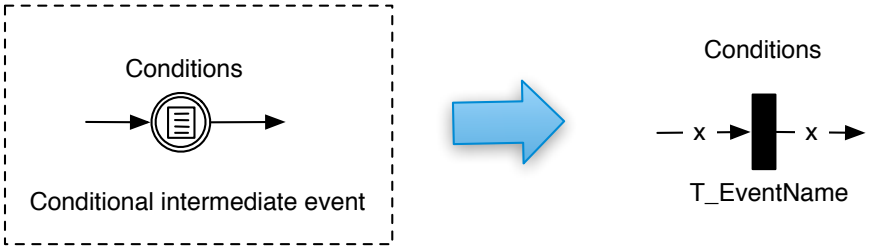


Figure 8.9: Rule 4: Conditional Intermediate Event

Rule 5: Message Start Event Figure 8.10: A message start event can be transformed either as a place waiting message to trigger an ID-net transition, or an external event which can synchronize with the transition. In both cases, it is possible to pass parameters or messages to the transition.

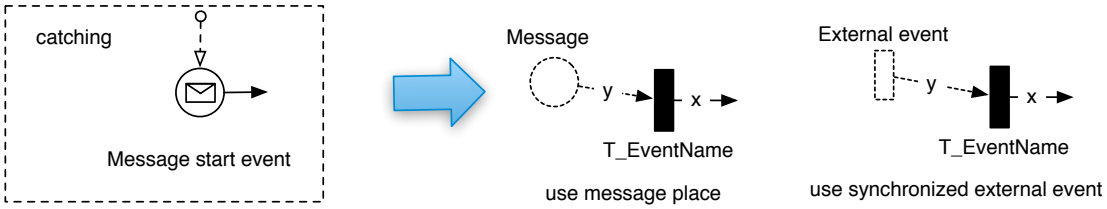


Figure 8.10: Rule 5: Message Start Event

Rule 6: Message Intermediate Event (Receive) Figure 8.11: The transformation of a receive message event is similar to the transformation of a message start event, but with an input place for the control flow. This transition is blocked until receiving a message or triggered by an external event.

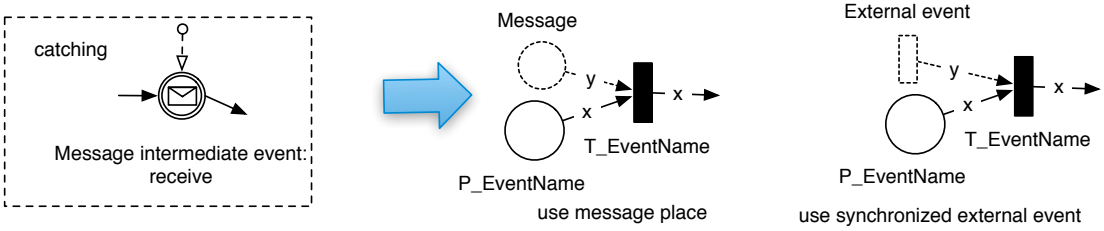


Figure 8.11: Rule 6: Message Intermediate Event (Receive)

Rule 7: Message Intermediate Event (Send) Figure 8.12: A send message event is transformed into an ID-net transition with input and output arcs (and places depending its neighbor elements), and an output place which holds the sent message. Or it can be a synchronization between this transition and an external event. This event is non-blocking, meaning the control flow proceeds forward after sending the message.

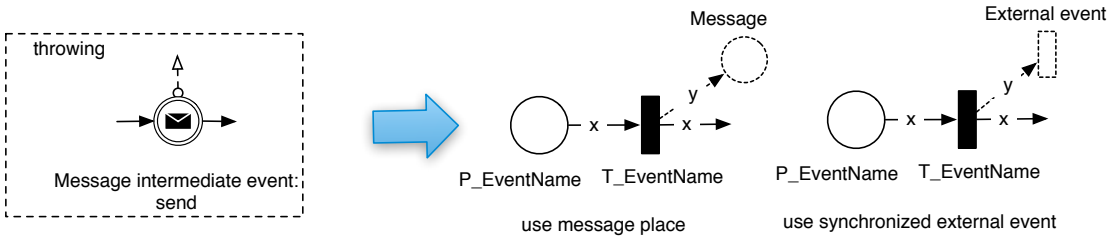


Figure 8.12: Rule 7: Message Intermediate Event (Send)

Rule 8: Message End Event Figure 8.13: A message end event is transformed into an ID-net transition which has an output message place or synchronized with an external event.

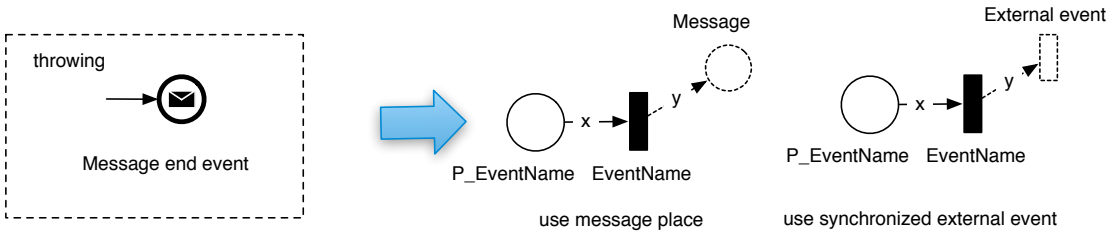


Figure 8.13: Rule 8: Message End Event

Rule 9: Parallel Gateway (Split) Figure 8.14: A parallel split gateway is transformed into an ID-net transition with two or more output arcs (workflow pattern AND-SPLIT).

Rule 10: Parallel Gateway (Join) Figure 8.15: A parallel joint gateway is transformed into an ID-net structure with two input flow, which implements the workflow pattern AND-JOIN.

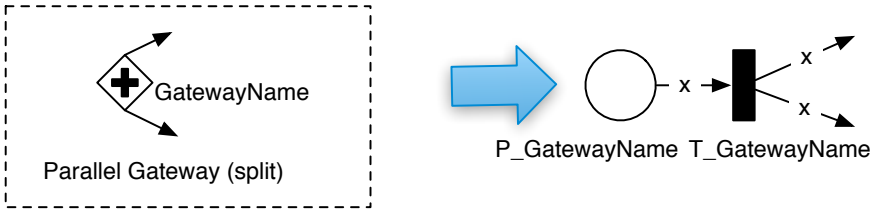


Figure 8.14: Rule 9: Parallel Gateway (Split)

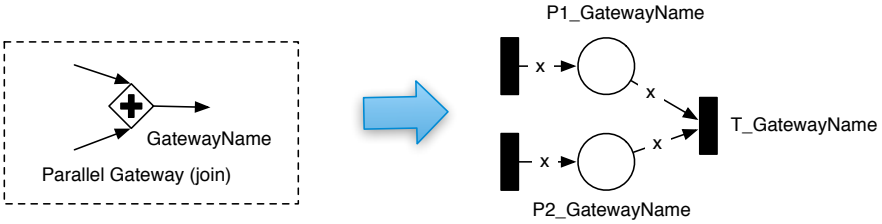


Figure 8.15: Rule 10: Parallel Gateway (Join)

Rule 11: Exclusive Gateway (Split) Figure 8.16: An exclusive split gateway is transformed into and ID-net structure with one input flow and two output flow, which implements the workflow pattern XOR-SPLIT.

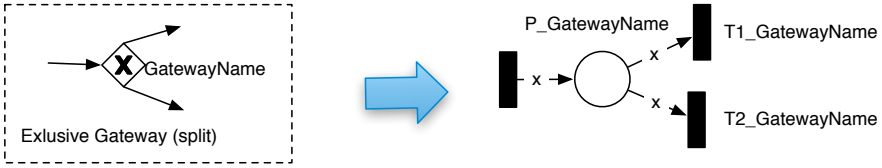


Figure 8.16: Rule 11: Exclusive Gateway (Split)

Rule 12: Exclusive Gateway (Join) Figure 8.17: An exclusive split gateway is transformed into and ID-net structure with two input flow and one output flow, which implements the workflow pattern OR-JOIN.

Rule 13: Chained Gateway Figure 8.18 show the situation where several gateways are chained. In the transformation, two chained gateway can share the same transition in ID-net.

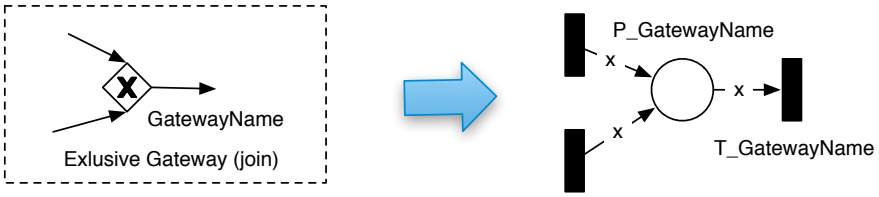


Figure 8.17: Rule 12: Exclusive Gateway (Join)

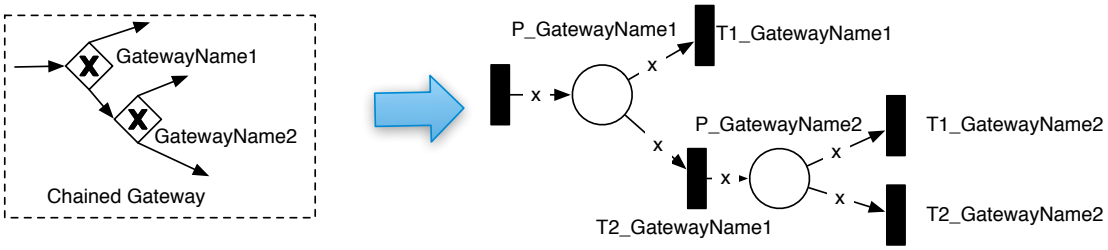


Figure 8.18: Rule 13: Chained Gateway

Example of transformation As an example, the control flow of the annotated BPMN diagrams for credit approval are transformed into encapsulated ID-net as shown in figure 8.19. Notice that merge may be applied to remove redundant places and transitions during transformation. Some services with input and output parameters, which are delegating external events, are synchronized with ID-net transitions. For example, the synchronizations of the transitions are:

- Transition $T\_invokeApprover$ is synchronized with service $InvokeApprover$, which will return the result of manual approval.
- Transition $T\_InvokeAssessor$ is synchronized with service $InvokeAssessor$, which return the result of risk evaluation.

Moreover, Transition $T\_assignYes$ is synchronized with data model operation:

$$request.approval := Yes$$

Code generation, model verification, and test generation techniques can be performed with this ID-net and the data model presented in figure 8.4.

Other aspects of business processes, such as organizational models and resources models can be also synchronized with the ID-net models. Or, we can use the actions of these models to annotate the BPMN diagram before the transformation.

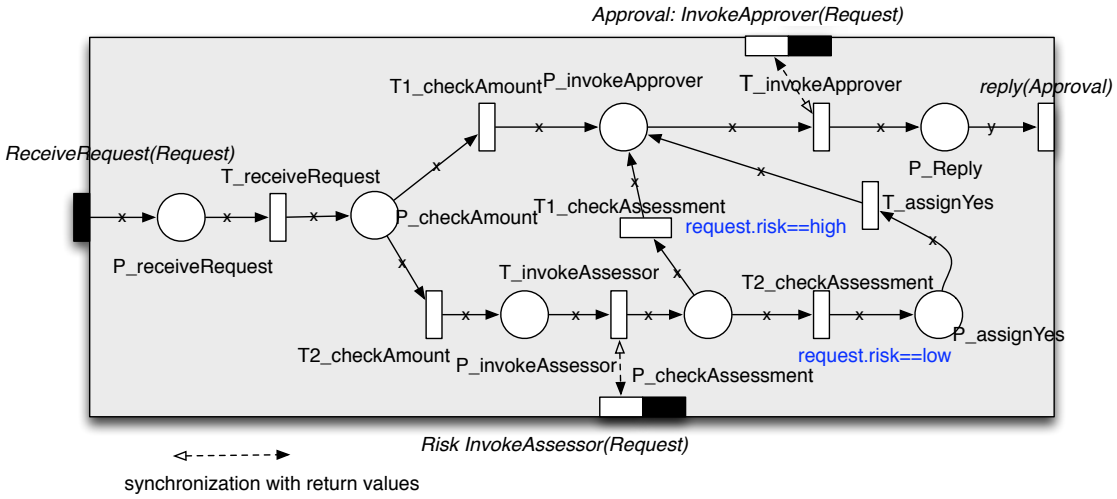


Figure 8.19: Control Flow of BPMN Process Transformed into Encapsulated ID-Net

8.3 Develop Web-based Business Processes

Currently many business processes are developed in pragmatic ways without using BPMS and the N-tier Browser/Server architecture is the standard architecture for collaborative enterprise applications. In this case, business processes are presented as flow of Webpages, where users interact with the business process via the Web components such as Links, Buttons, TextFields etc. The Web flow business processes are fine-grained and more concrete then the BPMN models, and it has two dimensions (or levels): the *user interface and navigation* dimension and the *core business process* dimension.

- the dimension of *user interface and navigation* consists of displaying information to the users, collecting/validating user inputs, and controlling the navigation between different Webpages.
- the *core business process* dimension is more abstract and it only considers important events/actions and data regarding to the states of business processes.

In a Web application, often the core business process is embedded into the Web flow navigation. However, usually the developers have to start from the core business process model to develop the Web application.

In this section we will show that ID-net can be used to develop fine-grained Web flow business processes. Web flow business processes are modeled directly as the synchronization of ID-nets and the related models, i.e. the navigation model and the data model.

8.3.1 An Online Insurance Subscription Application (TestIndus Project)

Figure 8.20 shows the Web flow navigation diagram we created for the application *ServiDirect*. This diagram represents the workflow of online insurance subscription. It can be directly and automatically derived from the HTML pages created in the early stage of development. In this diagram:

- each Webpage is represented by a rectangle.
- each navigation arc represents an action (related to a link or a button) which will change the state of the displayed Webpage or navigate to another Webpage.
- some activities are automatically carried out in background, thus invisible from Web users.
- The navigation (transition) between the pages is triggered by a user. Automatic activities can decide which page the user will be forwarded to.

The transformation or formalization of this diagram with ID-net is straightforward. As shown in figure 8.21, the Web flow is a Case Handling System with a single control point, i.e. on the page where the Web user is.

A co-model for the Webpage In order to model the behaviors of (inside) a Webpage, we propose a simple state-transition model (shown in figure 8.22) for Webpage:

- a Webpage is identified by a URL.
- a Webpage is at its initial state at the moment of arrival, e.g. at state 0.
- an action may *change the state of the page or lead to another page.*
- an action can be a click of a button or on a link etc.
- an input element is a parameter which will be sent to the controller, e.g. textbox, checkbox, polldown menu etc.
- input values may change the state of Webpage, e.g. online validation with Javascripts

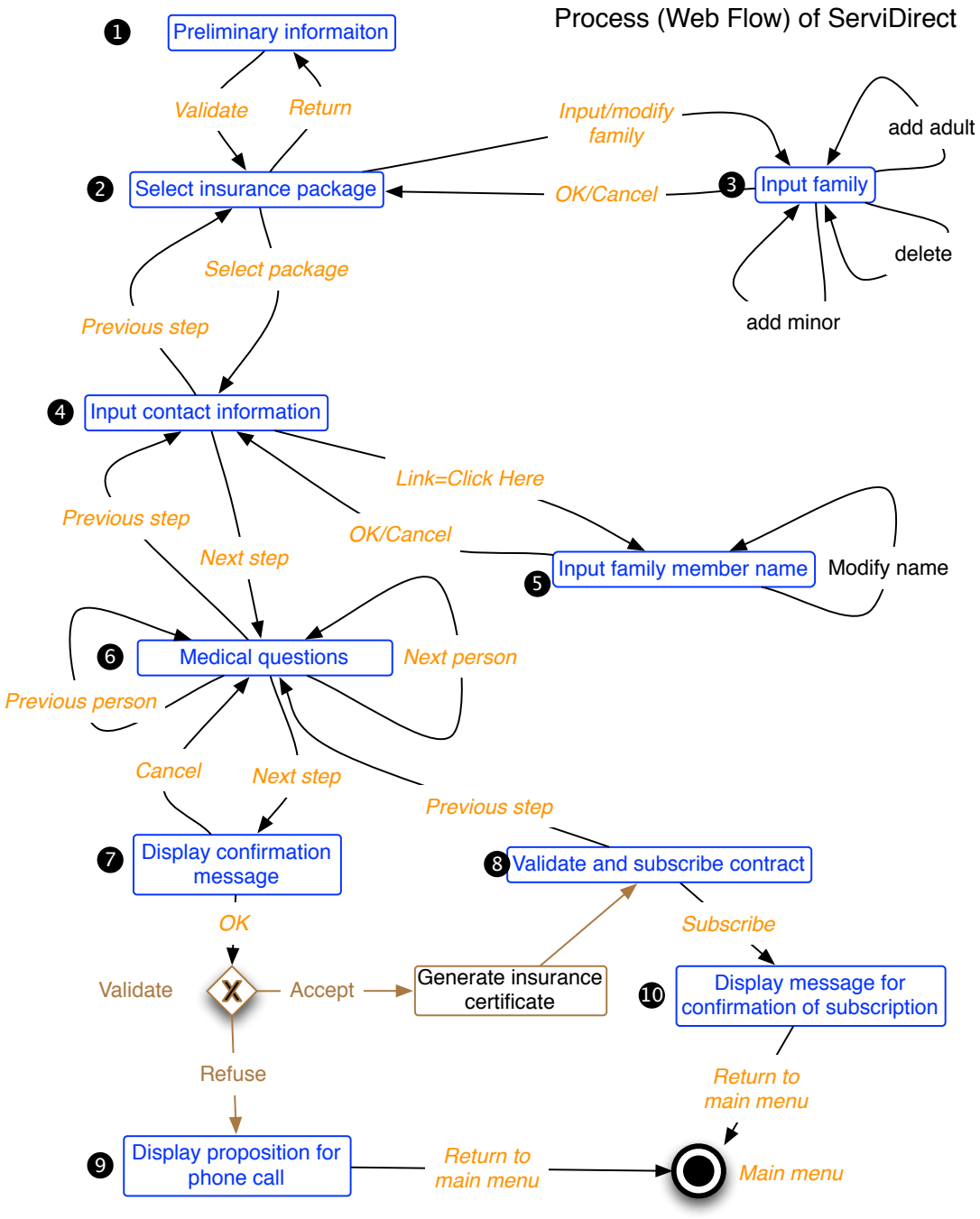


Figure 8.20: Web Flow of ServiDirect Online Insurance Subscription

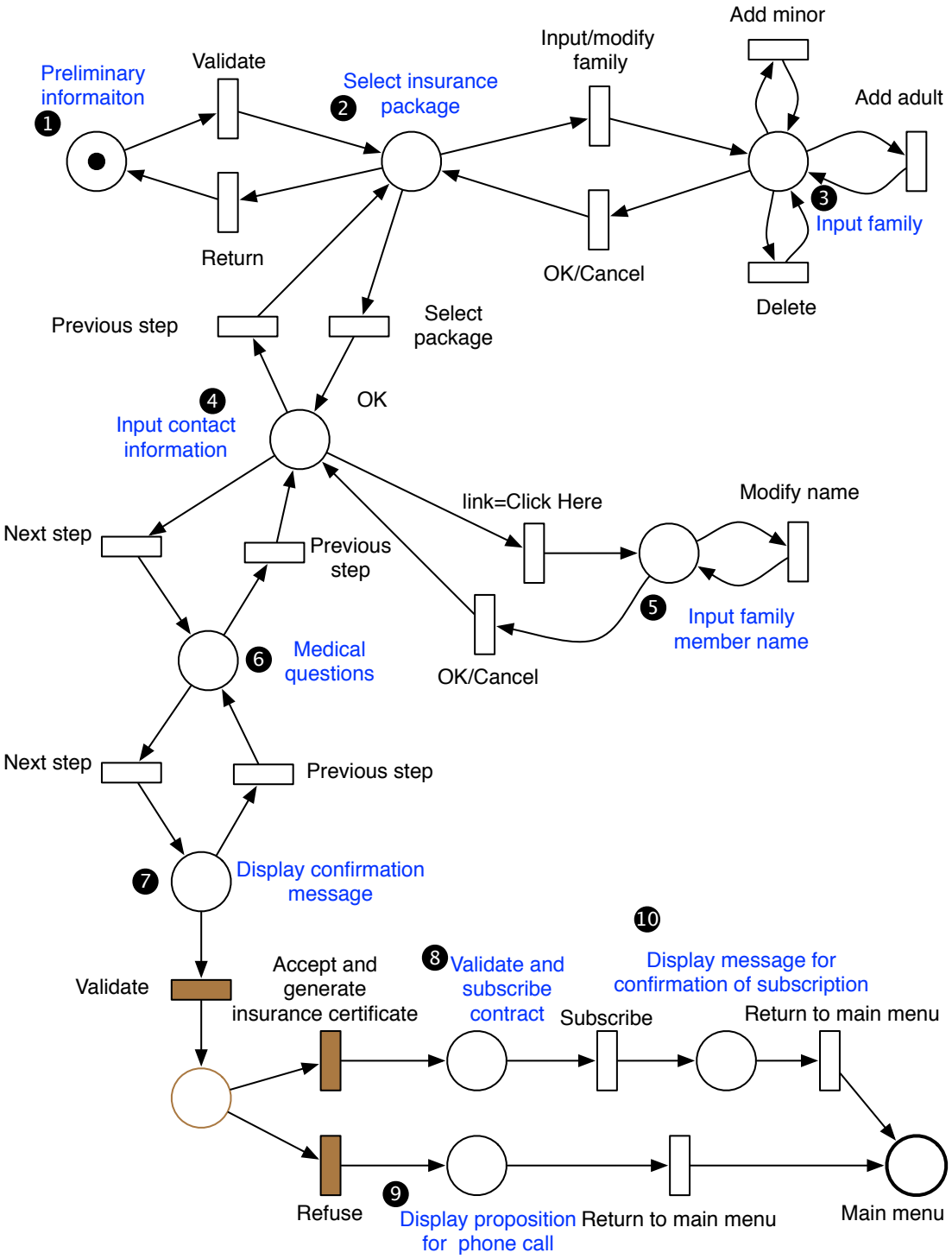


Figure 8.21: ID-net Representing the Web Flow

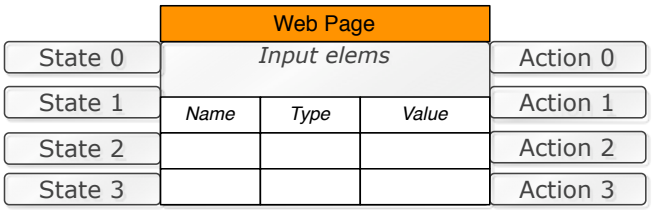


Figure 8.22: A Webpage model

- the state of a page includes: the state of the inputs and the state of page, sometimes they are synchronized automatically by the inline scripts (e.g. Javascripts), i.e. if the inputed value has wrong format, display error immediately.

Figure 8.23 shows an example of the Webpage model: the page of *Saisir information préliminaires* has tree inputs components: *NPA, Birthdate, Sex*, one action *Valider*, and two page states: *normal and error*. Each time a user click on button "Valider", the system will check if the input values are valid. If all input values are valid, the user will get to next page, otherwise the same page is return with error messages. All the other Webpages have the same structure.

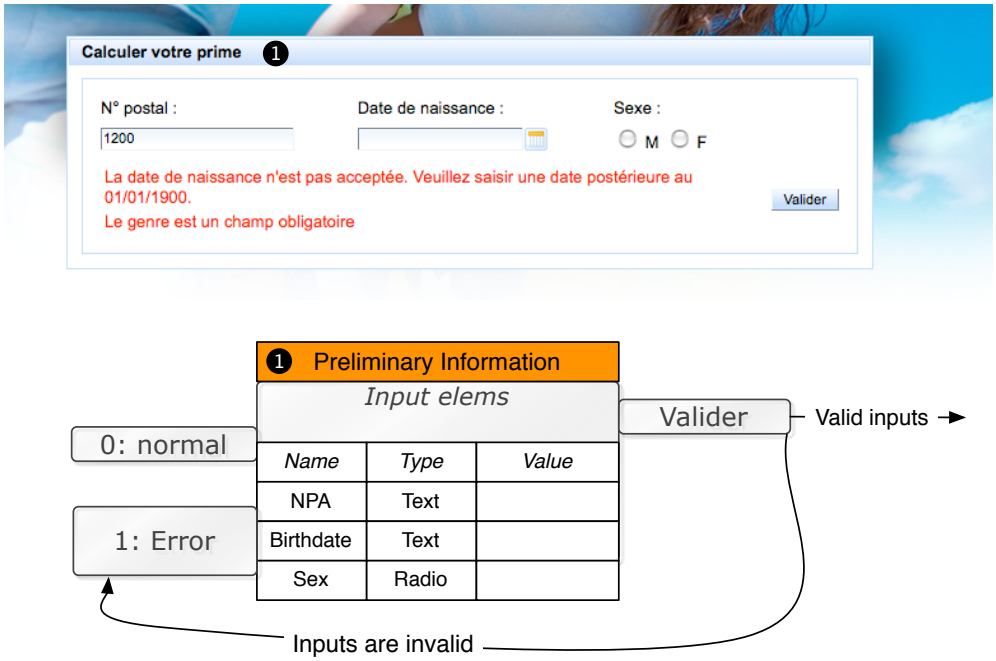


Figure 8.23: An Example of Webpage Model

This co-model represents the dimension behind the rectangles of figure 8.20 or the places of figure 8.21. Not all the actions in the Webpage model will be synchronized with the main control flow model. However, in this example, if an action leads the navigation to anther Webpage, it is synchronized with the main control flow model because the main control flow represents the Webpage navigation.

Develop Business Process based on the ID-net Model The development of business process based on the ID-net model is the same as the post office example presented in Chapter 7: by providing a data model and synchronizing (annotating) ID-net with the data model operations. There are many ways of modeling data for an application and we will not give much details about this.

8.3.2 Automatic Test Generation for Web Application Testing

As a surplus of using formal notations in software development, our models can be used to generate tests for the (manually) implemented Web applications. Figure 8.24 shows the chain of test generation in the TestIndus Project:

- in our implementation, the control flow model and the data models are encoded (manually) as facts in Prolog. This step can be automatic in the future.
- the automatic test generation is performed based on the Prolog facts. It is possible to add more business rules and constraints to target more precisely the intention of tests. The generated tests are scenarios which should have successful execution. Basically, we can specify the maximum steps of the sequences, combine the input parameters of each Webpage component, and verify if the values displayed on a Webpage are correct. The generated tests are stored in an intermediate format, similar to a scripting language. This intermediate format contains the actions and parameters used in precedent models, and it is domain and application-independent.
- finally, the intermediate scripts are transformed into Selenium scripts for Web interface testing. This transformation will map the actions and parameters used in the intermediate format to the actions and elements in the Webpages. We implemented this step by writing a Perl program which takes the intermediate format scripts and the mapping file, and generates XML format Selenium [101] scripts.

The complete explanation about the automatic test generation based on Algebraic Petri nets can be found in [102].

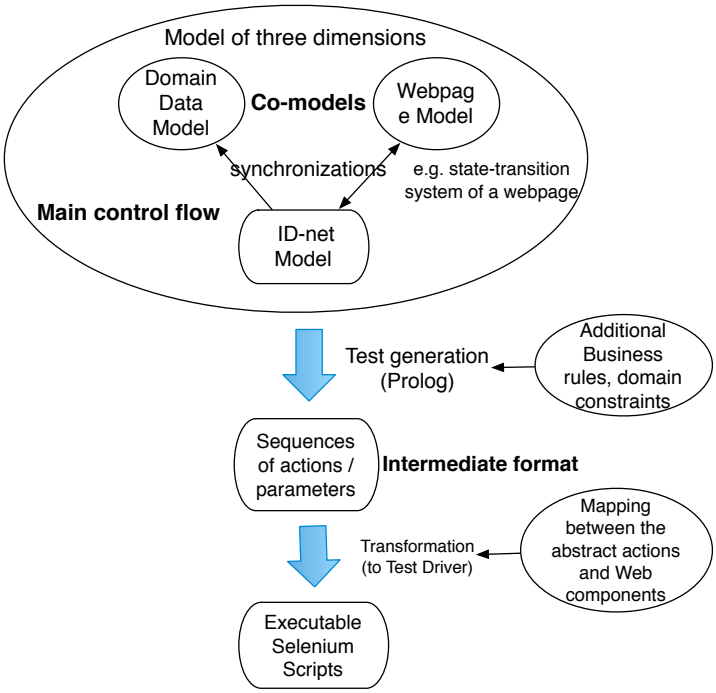


Figure 8.24: The Chain of Test Generation in TestIndus Project

8.4 Summary

In this Chapter, we presented two applications of our approach in developing business processes: by using a high-level graphical business process language (BPMN); and develop fine-grained business processes as Web applications.

In the first case, BPMN acts as the front-end graphical design language for business process modeling. The model we developed has two dimensions: control flow and data model. Control flow modeled by BPMN diagrams are mapped to ID-net models, which can be incrementally annotated (using the syntax of the data model) and become executable. Data model for the business processes can be developed in parallel with the control flow model.

In the second case, the model we developed has tree dimensions: control flow, data model, and the Webpage model. We can use ID-net directly to model the Web flow or generate ID-net from the HTML pages. In general, the ID-net model represents the core business process which is synchronized with data models and the Webpage models. Moreover, we can use these models to automatically generate tests and simulate human interaction with the Web applications.

Appendix A

CO-OPN Prototype Generation and Runtime Support

A.1 Prototype Generation

Runtime of the Generated Code . Figure A.1 shows the role of the runtime in CO-OPN code generation. The runtime supports transactions and interfacing the generated prototypes with external Java modules. In this section, we will give some technical details about how CO-OPN transactions are implemented in Java.

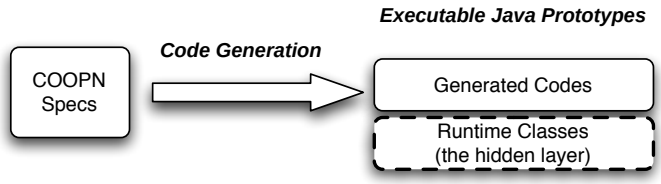


Figure A.1: Generated Code and the Runtime Classes

A.1.1 Prototype Integration Use Cases

Simple Method Call and Gate Event Listening Suppose that a prototype the "Alarm Counter" has been generated from CO-OPN specification, we want use the generated classes in our program.

1. create a new CoopnTransaction object:

   ```
   CoopnTransaction T=new CoopnTransaction();
   ```

2. create an instance of the prototype, e.g. for the acounter:

   ```
   alarm.acounter ac = new alarm.acounter(T,
           CoopnContextRT.getDefault(), "AC");
   ```

   where *CoopnContextRT.getDefault()* return a default coordination context, **AC** is the object's name. **T** is used to mark the creation time of object in a dynamic context, if **T** equals **null**, it means the object is created initially.

3. in order to capture the output event **display_: natural**, we add an anonymous listener:

   ```
   ac.adddisplay_Listener(new  acounter.display_Listener(){
     public void display_(acounter.display_Event e) {
       System.out.println("display: "+e.getArg1());
       }
     });
   ```

this listener will print out a message with the given parameter each time the gate event occurs.

4. the last step is to call the **tick** method, we need to create a new *CoopnTransaction* object to support this transaction. To be aware of either the transaction succeeds or fails, we need to catch the *CoopnMethodNotFirableException*:

```
try {
      ac.tick(T);
      T.commit();
      } catch (CoopnMethodNotFirableException e) {
            System.out.println("Method not firable.");
            // do things when the method can't be fired.
      }
```

Note the *CoopnMethodNotFirableException* is a subclass of Java *RuntimeException*.

According to the axioms, the **display_: natural** event occurs each time the **tick** method is called, we can see a message like:

```
display: 0
```

or

```
display: (succ 0)
```

if we call the method **tick** at the second time.

Redo with Transaction To explicitly force the component to perform a redo, we call the same method another time with the same transaction object.

```
try {
      ac.tick(T);
      ac.tick(T); // redo
      T.commit(); // commit the transaction
      } catch (CoopnMethodNotFirableException e) {
            System.out.println("Method not firable.");
            // do things when the method can't be fired.
      }
```

Note that the **redo** is semantically defined as non-determinist. The order of evaluation is not guaranteed to be always the same, i.e. choice of tokens and choice of axioms. For implementation, we adapt a general order for the backtracking, i.e. a determinist algorithm:

1. find the next axiom corresponding this transaction, if no next matched axiom found, throws an exception.
2. find the next eligible tokens (resources allocation solution) for the complete transaction, if no next solution found, the transaction fails and goto step 1.
3. evaluate the conditions of axioms, if the conditions are satisfied, **put the post-condition tokens** and returns ok, else goto step 2.

Basically, the idea is to try one-by-one axiom until find a solution. For each axiom, try all available tokens. The backtrack firstly occurs on the selection of tokens, then occurs on the selection of axioms.

Synchronize Gate with External Application In general, **Gates** in CO-OPN are transactional and always synchronized with internal transactions. If an external application is listening on a gate and want to synchronize with internal transactions of CO-OPN component, it can throw a *CoopnMethodNotFirableException* to force the CO-OPN component to make another possible attempt, e.g. with a different output parameter. This "veto" implies backtracking inside the CO-OPN component, so the component will try another way to finish the transaction until failure. If the external application does not throw the *CoopnMethodNotFirableException*, the CO-OPN component will considers that the transaction is finished.

For example, here is a listener who accepts only values greater than 2:

```
ac.adddisplay_Listener(new  acounter.display_Listener(){
  public void display_(acounter.display_Event e) {
    if (!cfc.std.basic.BooleansImpl.instance().coopnboolEquals(
      cfc.std.basic.NaturalsImpl.instance().gt(
           e.getArg1(),
           cfc.std.basic.NaturalsImpl.instance()._2()),
      cfc.std.basic.BooleansImpl.instance().TRUE())) {
        System.out.println("Veto: illegal parameter.");
        throw new CoopnMethodNotFirableException("should greater than 2");
        }
    System.out.println("display: "+e.getArg1());
    }
  });
```

This external event handler will force the component to change its internal behavior until an output value greater than 2 is given.

A.2 Design and Implementation of the Runtime

A.2.1 The Model

The main responsibility of the runtime classes is to implement the operational semantics of CO-OPN and provide services to the generated prototypes. The runtime is composed by the following objects:

- **Transaction**: a transaction is the essential unit to compose hierarchical transactions.
- **CoopnTransaction**: a CO-OPN specific implementation for the composition of transaction, which supports all kind of CO-OPN transactions. The **transaction tree** is composed by *CoopnTransaction* objects.
- **CoopnToken**: an encapsulation of data, whose instances represent tokens in Peri Nets.
- **CoopnPlace**: an abstraction of place in Petri Nets: basically it is a multi-set of tokens where we can add tokens into and remove tokens from the place.
- **Transition**: a transition specifies what (tokens) should be consumed and what will be produced when a transition fires.
- Additional stuffs: *TransactionException*, *TransactionMgr* etc.

Most of our discussions are based on the transaction tree and tokens with time stamps. Figure A.2 show the UML model of the runtime classes*.

A.2.2 Transaction and Transaction Tree

Let $T_s = \{t_1, t_2, ...t_i\}$ the set of transitions/transactions in a system, T the set of possible transactions, thus $\forall t_i, t_j \in T_s$:

- t_i *with* $t_j \in T$, where *with* is the *synchronization* operator

*Note that the current source codes for CoopnTransaction and other transactions are slightly different from this diagram, for the needs of simplicity and optimization.

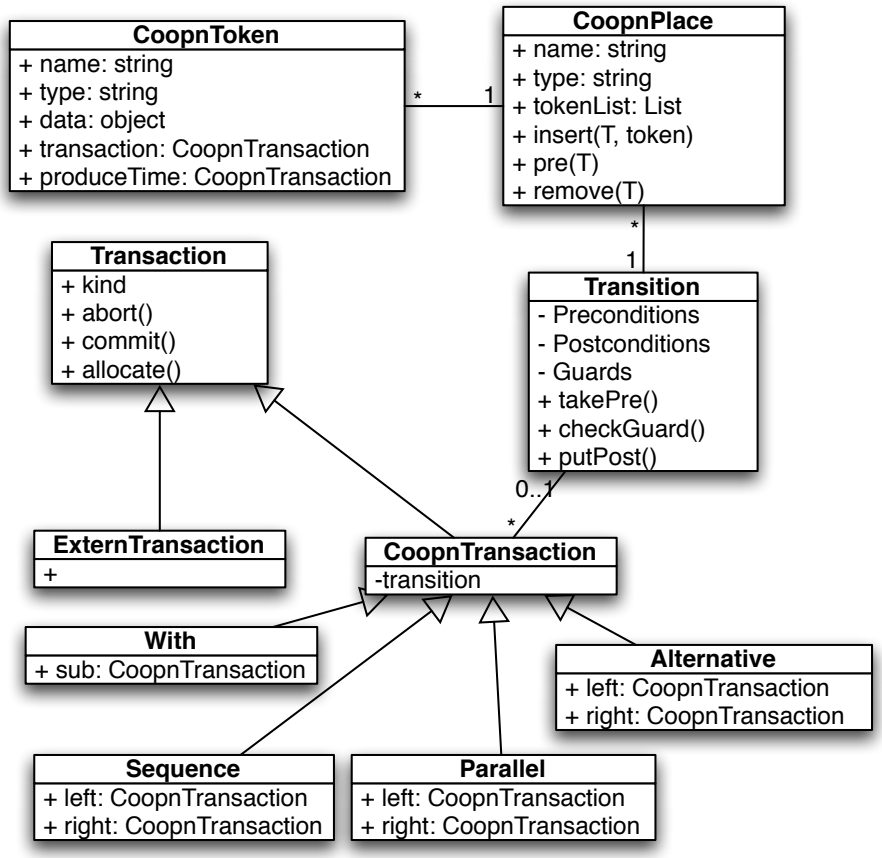


Figure A.2: Class Diagram of the Runtime Model

- $t_i // t_j \in T$, where $//$ is the *parallel* operator
- $t_i .. t_j \in T$, where $..$ is the *sequence* operator
- $t_i \oplus t_j \in T$, where $\oplus$ is the *alternative* operator

Transaction implies all or nothing effect for involved sub-transactions which corresponds success or failure of transaction.

Let **t** the root of a transaction tree, and **T** the set of involved sub-transactions, we identify the following relations between two transactions:

- $RootOf(t_i) = t, \forall t_i \in T, t_i \neq t$
- **succeed:** if t2 succeeds t1, tokens produced by t1 will be available for t2. e.g. $t1..t2$, $(t1..t3)..t2$, $(t1//t3)..t2$
- **concurrent:** if t1 concurrents with t2, t1 and t2 share the same set of tokens during the transaction. e.g. $t1//t2$, $t1$ *with* $t2//t3$
- **alternative:** if t1 is alternative with t2, t1 and t2 will be executed exclusively. e.g. $t1 \oplus t2$
- **alternative concurrent:** t1 is alternative concurrent with t2 means t1 and t2 **may** share the same set of tokens during the transaction. e.g. $t1//(t2 \oplus t3)$
- **alternative succeed:** t1 is alternative succeeds t2 means t1 **may** be succeeded by t2 which **may** uses tokens produced by t1. e.g. $(t1 \oplus t3)..t2$
- **parent and sub-transaction:** t2 is sub-transaction of t1 if t1 is the parent transaction of t2. e.g. transaction $t1//t2$ is the parent transaction of *t1* and *t2*.
- **equal:** t1 and t2 are equal if they are the same node in the transaction tree.

The most obvious **token eligibility rule** is:

Tokens produced by **t1** are available for t2 only if **t2 succeeds t1**.

The visit of the transaction tree is done in a **deep-first** manner, the pre-, post- conditions, and guards of each node are evaluated before visiting its children. There are three types of visit for a transaction tree: **allocate(reserve)**, **abort**, and **commit**:

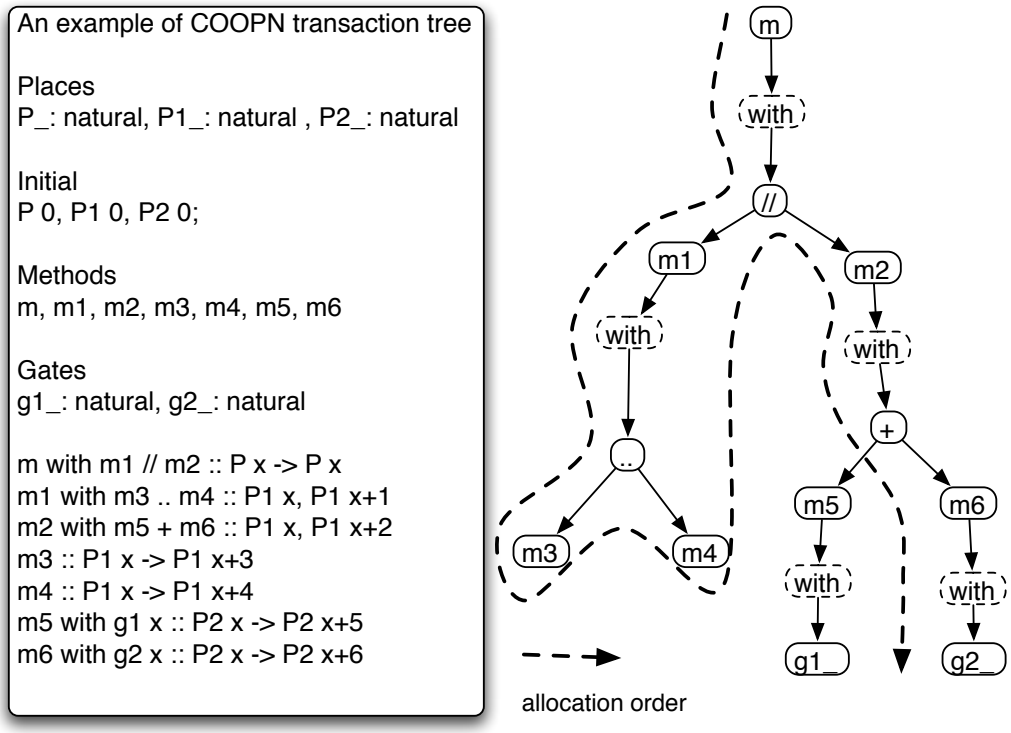


Figure A.3: An Example of Transaction Tree

- **allocate** or **reserve** corresponds the **prepare** primitive in multi-phase transaction systems. A complete visit of the transaction tree with **allocate** means a successful allocation of transaction resources, i.e. all necessary tokens are reserved (tagged). Backtracking occurs during the allocation phase until failure or a solution is found. Failure of an allocation means the transaction is not possible with actual markings. If an allocation fails, the **abort** visit is called to reverse the state of system as **"nothing happened"**, i.e. remove all intermediate tokens and untag the reserved tokens.
- **mi-abort** will reverse the side effects done by the precedent **allocate** process, but the allocation process can be re-executed (backtracking or redo).
- **abort** will abort the transaction. The transaction becomes dead after the **abort**, so no further action is possible on the transaction.
- **commit** will commit and finalize the transaction with the allocated (tagged) resources. It simply remove all reserved tokens in CoopnPlaces, and changes the *free intermediate tokens* to *free initial tokens*.

A.2.3 CoopnTransaction as Time Stamp

A.2.3.1 CoopnToken

General Description There are 4 kinds of tokens in the system:

- Free initial tokens
- Reserved initial tokens
- Free intermediate tokens
- Reserved intermediate tokens

Figure A.5 depicts the life-cycle of a CoopnToken: a CoopnToken can be either initially created or produced by a transaction; it can be consumed by the succeed transaction. The notion of *succeed* is defined by the relation of two transactions on the same transaction tree.

Concretely, a CoopnToken uses two variables to indicate its actual state, each variable refers to a *CoopnTransaction* instance, i.e. a time stamp:

- **transaction**: the transaction which has reserved this token. *null* means this token is free and not reserved by any transaction.

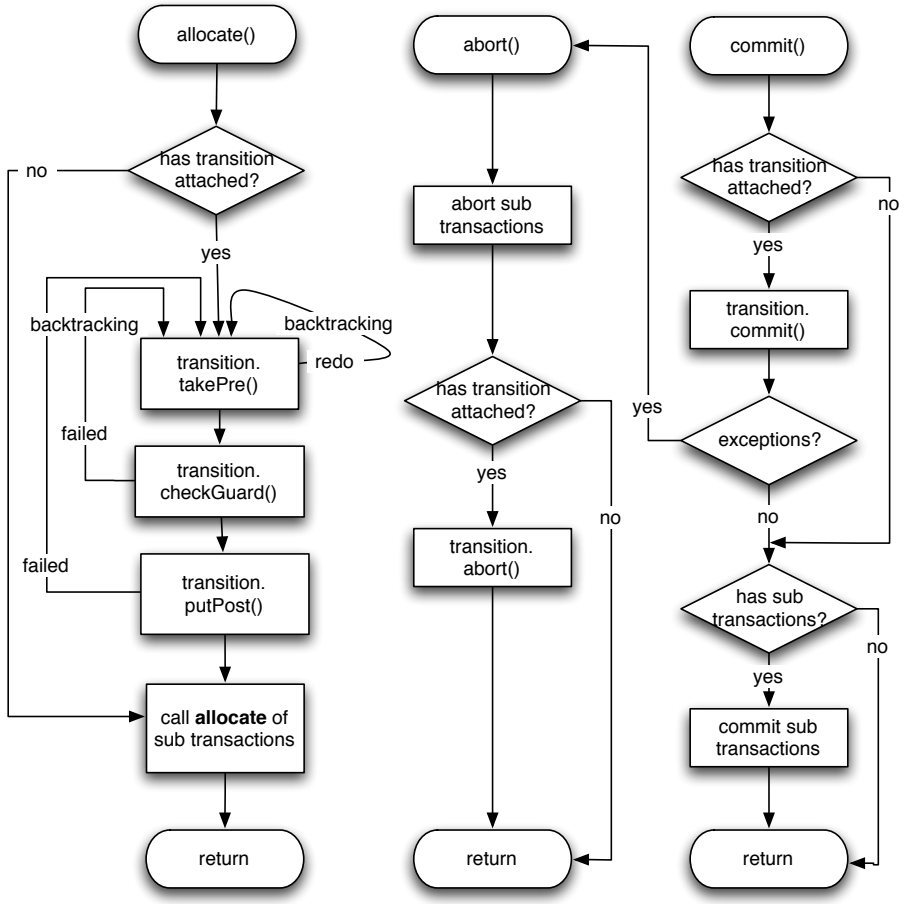


Figure A.4: Execution of CoopnTransaction

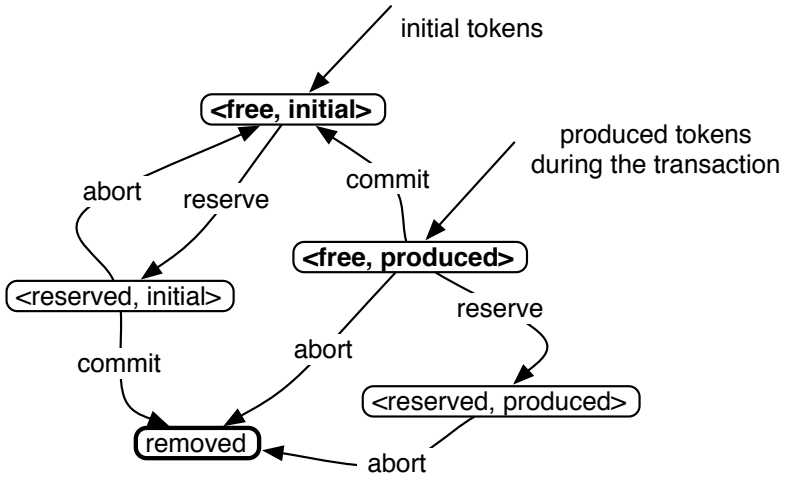


Figure A.5: CoopnToken Life-Cycle

- **produceTime**: the transaction which produces this token. *null* means this token exists before the beginning of the transaction.

Formalization Let **TN** the set of tokens in a system. A CoopnToken $tn \in TN$ is an entity whose state is the Cartesian product $TR \times TP$, where:

- T is the set of transactions in the system
- $TR \Leftrightarrow \{null\} \cap T$
- $TP \Leftrightarrow \{null\} \cap T$

$$\forall tn \in TN \text{ with state } < tr, tp >, \forall tr \in TR,\ \forall tp \in TP,\ tr \neq null,\ tp \neq null,$$

$$RootOf(tr) = RootOf(tp)$$

A.2.4 CoopnPlace

CoopnPlace maintains a set of *CoopnTokens* which have compatible type with the place. The tagging of tokens is done by *CoopnPlace*, i.e. during the insertion of token into the CoopnPlace, *CoopnTransaction*. The responsibilities of *CoopnPlace* are:

- tagging a token when it is inserted, e.g. P_.insert(token, T) ⇒ token.setProduceTime(T)

- find eligible tokens for a given transaction, return a token iterator. e.g. $P\_.pre(T)$, iterator is used for variable-binding.
- compute results in response to commit and abort of transactions: tag/untag tokens, remove tokens.

Another important work of *CoopnPlace* is to find the eligible or visible tokens during a complex transaction, especially with transactions using sequential operator "**..**". An iterator of eligible tokens is returned to the transaction which needs tokens from the CoopnPlace. The cursor on the iterator is used for the backtracking, in case of necessary. During the exploration of the transaction tree, the eligibility or visibility rules implemented by *CoopnPlace* are:

- All reserved tokens are not eligible.
- Tokens produced by the simultaneous transactions are not eligible, e.g. With, // .
- Free tokens produced by precedent sequential transactions are eligible. e.g. in *seq1..(sim1//sim2)*, tokens produced by *seq1* are eligible for transaction *sim1* and *sim2*. However, *sim1* and *sim2* have concurrent access of the tokens. In the class CoopnTransaction, this work is carried out by a function called "*isAfter(CoopnTransaction)*", e.g. for two nodes t_1 and t_2 on the same transaction tree, if $t_2.isAfter(t_1)$ returns *true*, it means tokens produced by t_1 are eligible for t_2, provided that no other transactions take the tokens before t_2.

Condition Evaluation and Backtracking Backtracking and Redo

- Backtracking is a basic search technique used to find solutions during the exploration of tree-like or graph-like solution space.
- User explicitly **redo** a transaction in order to use another (next) solution. **Redo** uses the backtracking technique.

Technically, backtracking requires a determinist search algorithm or additional spaces to store already explored nodes. There are several levels of backtracking in a CO-OPN model:

- Token. Try next eligible tokens.
- Axiom. Try next axiom if there is. Each axiom has its own pre-conditions, guards, and post-conditions.

- Transaction or method. e.g. in transaction m *With m1..m2*, the evaluation order is $m \rightarrow m1 \rightarrow m2$, when *m2* is not firable, we need backtrack to *m1* or m to find a solution which satisfy all the three transactions.

A.2.5 Runtime Variables and Stacks

For each object, two variables and three stacks are maintained transparently during the execution of transactions. The two variables are:

- **Last transaction**, *_lastTransaction*: The last transaction on the transaction tree during the exploration. Initialized to **null**, its value is assigned until the first complete execution of the transaction, afterwards its value is used by the **redo** process.
- **The redone transaction**, *_redone*: The transaction which has been redone successfully. **null** means nothing has been redone.

The three stacks are:

- **Token Iterator Stack**: The iterator of tokens for each variable.
- **Transaction Stack**: The actual transaction.
- **Axiom Index Stack**: The actually used axiom.

A.2.6 The Process of Generating Java Codes

Figure A.6 shows the process of Java code generation.

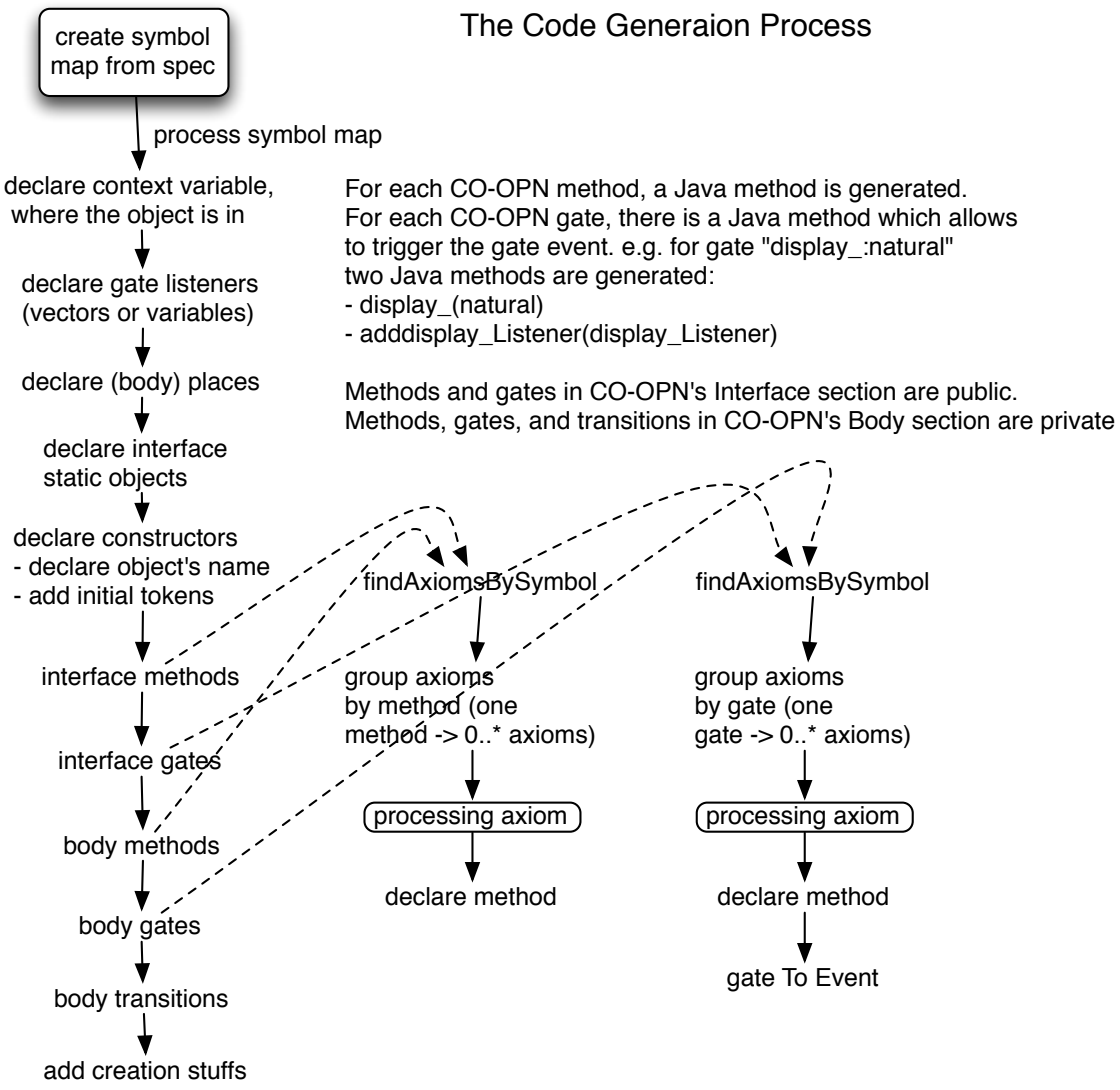


Figure A.6: The Process of Java Prototypes Generation

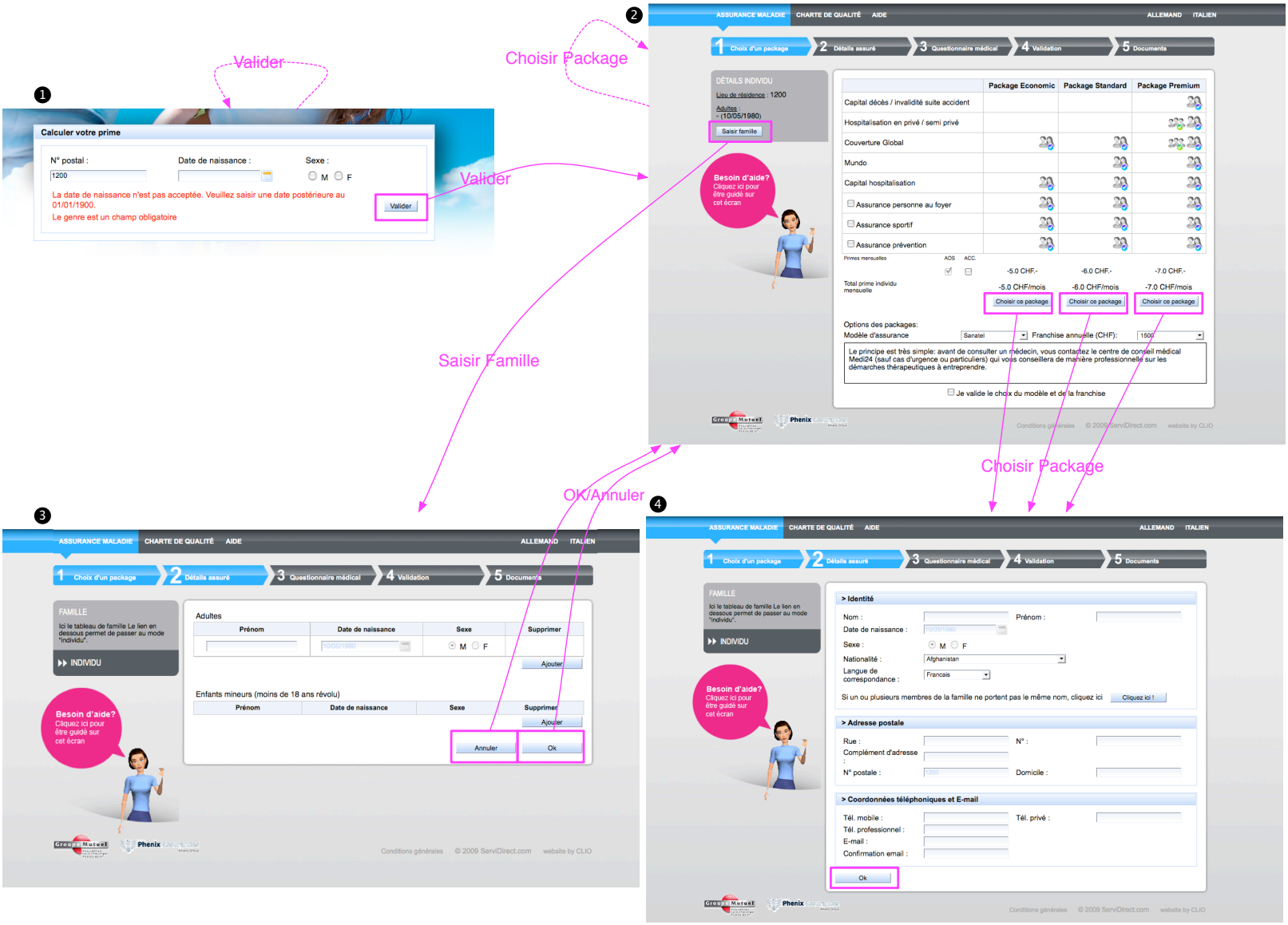
Appendix B

Example of ID-net in PNML-alike Format and Code Generation

```
<pnml xmlns="http://smv.unige.ch">
    <net id="fibnum" type="idnet" resources="res">
        <name>
            <text>A very simple ID-net with inscriptions</text>
        </name>
        <place id="p1" type="int">
            <name>
                <text>p1</text>
            </name>
            <initialMarking>
                <text>0</text>
            </initialMarking>
        </place>
        <place id="p2" type="int">
            <name>
                <text>p2</text>
            </name>
            <initialMarking>
                <text>5</text>
            </initialMarking>
        </place>

        <transition id="t1">
            <name>
                <text>t1</text>
            </name>
            <precondition> x &lt;= 1 </precondition>
            <postcondition>  </postcondition>
            <inscription> z:=x+y </inscription>
        </transition>
        <transition id="t2">
            <name>
                <text>t2</text>
            </name>
            <precondition> x &gt; 1 </precondition>
            <postcondition> </postcondition>
            <inscription> y:=x-1; z:=x-2 </inscription>
        </transition>

        <arc id="a1" source="p1" target="t1">
            <inscription> y </inscription>
        </arc>
        <arc id="a2" source="t1" target="p1">
            <inscription> z </inscription>
        </arc>
        <arc id="a3" source="p2" target="t1">
            <inscription> x </inscription>
        </arc>
        <arc id="a4" source="p2" target="t2">
            <inscription> x </inscription>
        </arc>
        <arc id="a5" source="t2" target="p2">
            <inscription> y,z </inscription>
        </arc>
    </net>

    <resources id="res" name="Some variables">
        <variable name="a" type="*int"/>
        <variable name="b" type="*int"/>
    </resources>
</pnml>
```

Appendix C

Example of Webpages and Test Generation in TestIndus Project


1
Valider
Calculer votre prime
N° postal :
1200
Date de naissance :
Sexe :
M F
La date de naissance n'est pas acceptée. Veuillez saisir une date postérieure au 01/01/1900.
Le genre est un champ obligatoire
Valider
Valider
Choisir Package
2
ASSURANCE MALADIE
CHARTE DE QUALITÉ
AIDE
ALLEMAND
ITALIEN
1 Choix d'un package
2 Détails assuré
3 Questionnaire médical
4 Validation
5 Documents
DÉTAILS INDIVIDU
Lieu de résidence : 1200
Adultes : (10/05/1980)
Saisir famille
Besoin d'aide?
Cliquez ici pour être guidé sur cet écran
Package Economic
Package Standard
Package Premium
Capital décès / invalidité suite accident
Hospitalisation en privé / semi privé
Couverture Global
Mundo
Capital hospitalisation
Assurance personne au foyer
Assurance sportif
Assurance prévention
Primes mensuelles
AOS
ACC.
-5.0 CHF.-
-6.0 CHF.-
-7.0 CHF.-
Total prime individu mensuelle
-5.0 CHF/mois
-6.0 CHF/mois
-7.0 CHF/mois
Choisir ce package
Choisir ce package
Choisir ce package
Options des packages:
Modèle d'assurance
Sanatel
Franchise annuelle (CHF):
1500
Le principe est très simple: avant de consulter un médecin, vous contactez le centre de conseil médical Medi24 (sauf cas d'urgence ou particuliers) qui vous conseillera de manière professionnelle sur les démarches thérapeutiques à entreprendre.
Je valide le choix du modèle et de la franchise
Groupe Mutuel
Phenix
Conditions générales
© 2009 ServiDirect.com
website by CLIO
Saisir Famille
OK/Annuler
Choisir Package
3
ASSURANCE MALADIE
CHARTE DE QUALITÉ
AIDE
ALLEMAND
ITALIEN
1 Choix d'un package
2 Détails assuré
3 Questionnaire médical
4 Validation
5 Documents
FAMILLE
Ici le tableau de famille Le lien en dessous permet de passer au mode "individu".
INDIVIDU
Besoin d'aide?
Cliquez ici pour être guidé sur cet écran
Adultes
Prénom
Date de naissance
Sexe
Supprimer
M F
Ajouter
Enfants mineurs (moins de 18 ans révolu)
Prénom
Date de naissance
Sexe
Supprimer
Ajouter
Annuler
Ok
Groupe Mutuel
Phenix
Conditions générales
© 2009 ServiDirect.com
website by CLIO
4
ASSURANCE MALADIE
CHARTE DE QUALITÉ
AIDE
ALLEMAND
ITALIEN
1 Choix d'un package
2 Détails assuré
3 Questionnaire médical
4 Validation
5 Documents
FAMILLE
Ici le tableau de famille Le lien en dessous permet de passer au mode "individu".
INDIVIDU
Besoin d'aide?
Cliquez ici pour être guidé sur cet écran
> Identité
Nom :
Prénom :
Date de naissance :
Sexe :
M F
Nationalité :
Afghanistan
Langue de correspondance :
Francais
Si un ou plusieurs membres de la famille ne portent pas le même nom, cliquez ici
Cliquez ici !
> Adresse postale
Rue :
N° :
Complément d'adresse :
N° postale :
Domicile :
> Coordonnées téléphoniques et E-mail
Tél. mobile :
Tél. privé :
Tél. professionnel :
E-mail :
Confirmation email :
Ok
Groupe Mutuel
Phenix
Conditions générales
© 2009 ServiDirect.com
website by CLIO

List of Figures

Bibliography

[1] Wikipedia.org. Business process reengineering.

[2] World Wide Web Consortium (W3C). Web services description language.

[3] Peri Tarr, Harold Ossher, William Harrison, Stanley M. Sutton, and Jr. N degrees of separation: Multi-dimensional separation of concerns. pages 107–119, 1999.

[4] Didier Buchs and Nicolas Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering*, 26(7):635–652, july 2000.

[5] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha, F. De Cindio, and G. Rozenberg, editors, *Advances in Petri Nets on Object-Orientation*, LNCS, pages 70–127. Springer-Verlag, 2001.

[6] W. M. P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, 1999.

[7] Ekkart Kindler. On the semantics of epcs: A framework for resolving the vicious circle. In *EPK*, pages 7–18, 2003.

[8] Ekkart Kindler. On the semantics of epcs: Resolving the vicious circle. *Data Knowl. Eng.*, 56(1):23–40, 2006.

[9] www.epml.de. Epc markup language (epml).

[10] M. Dumas and A. ter Hofstede. Uml activity diagrams as a workflow specification language. In *In Proc. of the International Conference on the Unified Modeling Language (UML) Toronto, Canada*, October 2001, Springer Verlag.

[11] Workflow Management Coalition (WfMC). Workflow reference model and standards.

[12] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web service v1.1, ibm, bea systems, microsoft, sap ag, siebel systems, 2003.

[13] Business Process Management Initiative (BPMI). Business process modeling notation (bpmn), 2004.

[14] Stephen A. White. Mapping bpmn to bpel example. *IBM Corp., United States*, February 2005.

[15] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *QUT Technical report, FIT-TR-2002-02*, 2002. (Also see http://www.tm.tue.nl/it/research/patterns).

[16] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow resource patterns. Technical report, BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004.

[17] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. Technical report, QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[18] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and N. Russell. Pattern-based analysis of uml activity diagrams. In *BETA Working Paper Series, WP 129, Eindhoven University of Technology, Eindhoven*, 2004.

[19] J. Mendling, G. Neumann, and M. Nuttgens. Towards workflow pattern support of event-driven process chains (epc). In *M. Nuttgens, J. Mendling, eds.: Proc. of the 2nd GI Workshop XML4BPM - XML for Business Process Management" at BTW 2005, Karlsruhe, Germany, CEUR Workshop Proceedings Volume 145, ISSN-1613-0073, pages 23-38*, March 2005.

[20] Frank Puhlmann and Mathias Weske. Using the pi-calculus for formalizing workflow patterns. In *Proceedings of 3rd International Conference on Business Process Management*, September 2005.

[21] P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Pattern-based analysis of bpel4ws. Technical report, QUT Technical report, FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002.

[22] W.M.P. van der Aalst. Patterns and xpdl: A critical evaluation of the xml process definition language. Technical report, QUT Technical report, FIT-TR-2003-06, Queensland University of Technology, Brisbane, 2003.

[23] W. M. P. van der Aalst and A. H. M. ter ter Hofstede. YAWL: Yet another workflow language. *QUT Technical report, FIT-TR-2002-06*, 2002.

[24] Daniel Moldt and Heiko Rölke. Pattern based workflow design using reference nets. In *Business Process Management*, pages 246–260, 2003.

[25] Stephen A. White. Process modeling notations and workflow patterns. *IBM Corp., United States*, 2004.

[26] W.M.P. van der Aalst N. Russell, A.H.M. ter Hofstede and N. Mulyar. Technical report.

[27] W. M. P. van der Aalst, J. Desel, and E. Kindler. On the semantics of EPCs: A vicious circle. In *Proceedings of the EPK 2002: Business Process Management using EPCs / Nèttgens, M.; Rump, F. (Eds.)*, pages 71–80, Trier, Germany, Nov. 2002. Gesellschaft fèr Informatik, Bonn.

[28] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service composition. In *In Proceedings of 18th IEEE International Conference on Automated Software Engineering, pages 152?161, Montreal, Canada, IEEE Computer Society*, October 2003.

[29] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *In Proceedings of 13th International Conference on World Wide Web, pages 621?630, New York, NY, USA, ACM Press.*, 2004.

[30] J.A. Fisteus, L.S. Fernandez, and C.D. Kloos. Formal verification of bpel4ws business collaborations. In *In Proceedings of 5th International Conference on Electronic Commerce and Web Technologies (EC-Webè4), volume 3180 of Lecture Notes in Computer Science, pages 76?85, Zaragoza, Spain, Springer-Verlag*, August 2004.

[31] A. Ferrara. Web services: a process algebra approach. In *In Proceedings of 2nd International Conference on Service Oriented Computing, pages 242?251, New York, NY, USA. ACM Press*, 2004.

[32] M. Koshkina and F. van Breugel. Verification of business processes for web services. Technical report, Technical Report CS-2003-11, York University, October 2003.

[33] R. Farahbod, U. Glasser, and M. Va jihollahi. Abstract operational semantics of the business process execution language for web services. Technical report, Technical Report SFU-CMPT-TR-2004-03, School of Computer Science, Simon Fraser University, Burnaby B.C. Canada, April 2004.

[34] S. Hinz, K. Schmidt, and C. Stahl. Transforming bpel to petri nets. In *Proceedings of 3rd International Conference on Business Process Management*, September 2005.

[35] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal semantics and analysis of control flow in ws-bpel. Technical report, BPM Center Report BPM-05-15, BPMcenter.org, 2005.

[36] Frank Puhlmann. On the suitability of the pi-calculus for business process management. pages 51–62. 2007.

[37] W. M. P. van der Aalst. Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype, 2004.

[38] Howard Smith and Peter Fingar. Workflow is just a pi process. *BPTrends*, January 2004.

[39] W. M. P. van der Aalst. Three good reasons for using a petri-net-based workflow management system. *The Kluwer International Series in Engineering and Computer Science: Information and Process Integration in Enterprises: Rethinking Documents, Chapter 10*, 428:161–182, 1998.

[40] M.H.Jansen-Vullers and H.A. Reijers. Business process redesign at a mental healthcare institute: A coloured petri net approach. In *Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Department of Computer Science, University of Aarhus, PB-576, 21-38*, October 2005.

[41] Christian W. Gunther and Wil M.P. van der Aalst. Modeling the case handling principles with colored petri nets. In *Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Department of Computer Science, University of Aarhus, PB-576, 211-230*, October 2005.

[42] Mariska Netjes, Wil M.P. van der Aalst, and Hajo Reijers. Analysis of resource-constrained processes with colored petri nets. In *Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Department of Computer Science, University of Aarhus, PB-576, 251-265*, October 2005.

[43] Irene Vanderfeesten, Wil M.P. van der Aalst, and Hajo Reijers. Modelling a product based workflow system in cpn tools. In *Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Department of Computer Science, University of Aarhus, PB-576, 99-118.*, October 2005.

[44] Carnegie Mellon University. Smv homepage. http://www.cs.cmu.edu/ modelcheck/.

[45] Gerard J. Holzmann. *The SPIN Model Checker*. Addison Wesley, 2003.

[46] Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén. Sat-solving the coverability problem for petri nets. *Form. Methods Syst. Des.*, 24(1):25–43, 2004.

[47] E. Pastor and J. Cortadella. Efficient encoding schemes for symbolic analysis of petri nets. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 790–795, Washington, DC, USA, 1998. IEEE Computer Society.

[48] J.-M. Couvreur, E.Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P. Wacrenier. Data decision diagram for petri nets analysis. In *23rd international conference on application and theory of Petri Nets (ATPN 2002), jun 2002, Australia.*, volume LNCS vol 2360, 2002.

[49] V.Beaudenon and E. Encrenaz. Data decision diagrams for promela systems analysis. Technical report, LIP6 research report, 2005.

[50] Ekkart Kindler and Michael Weber. The dimensions of petri nets: The petri net cube. *Bulletin of EATCS*, 66:155–166, 1998.

[51] W.M.P. van der Aalst. Multi-dimensional petri nets. *Computing Science Notes 93/26, Eindhoven University of Technology, Eindhoven*, 1993.

[52] Charles Lakos. The object orientation of object petri nets. In *Proceedings of Workshop on Object Oriented Programming and Models of Concurrency*, pages 2–7, 1995.

[53] Charles Lakos. From coloured petri nets to object petri nets, 1995.

[54] Olaf Kummer, Universitt Hamburg, and Fachbereich Informatik. Introduction to petri nets and reference nets, 2001.

[55] Olaf Kummer and Frank Wienberg. Renew - the reference net workshop. In *Petri Net Newsletter*, pages 12–16, 2000.

[56] Fernando Rosa-Velardo and David de Frutos-Escrig. Name creation vs. replication in petri net systems. *Fundam. Inf.*, 88(3):329–356, 2008.

[57] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[58] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. Dfl: A dataflow language based on petri nets and nested relational calculus. *Inf. Syst.*, 33(3):261–284, 2008.

[59] Andreas Oberweis and Peter Sander. Information system behavior specification by high level petri nets. *ACM Trans. Inf. Syst.*, 14(4):380–420, 1996.

[60] Roberto Bruni and Ugo Montanari. Zero-safe nets: The individual token approach. In *Workshop on Algebraic Development Techniques*, pages 122–140, 1997.

[61] R. Bruni and U. Montanari. Zero-safe nets: Composing nets via transition synchronization, 1999.

[62] Roberto Bruni and Ugo Montanari. Transactions and zero-safe nets. *Lecture Notes in Computer Science*, 2128:380++, 2001.

[63] Sren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with channels for synchronous communication. In *Application and Theory of Petri Nets 1994, Proc. of 15th Intern. Conf*, pages 159–178. Springer.

[64] Ugo Montanari. True concurrency: Theory and practice. In *Proceedings of the Second International Conference on Mathematics of Program Construction*, pages 14–17, London, UK, 1993. Springer-Verlag.

[65] Lalita Jategaonkar. *Observing "true" concurrency.* PhD thesis, Cambridge, MA, USA, 1993.

[66] François Baccelli, Nathalie Furmento, and Bruno Gaujal. Parallel and distributed simulation of free choice petri nets. *SIGSIM Simul. Dig.*, 25(1):3–10, 1995.

[67] et al. D. Borrione, M. Boubekeur. An approach to the introduction of formal validation in an asynchronous circuit design flow. In *Publ. in 36th Hawai International Conference on Systems Science (HICSS'03)*, January.

[68] Luca Aceto. *Action refinement in process algebras.* Cambridge University Press, New York, NY, USA, 1992.

[69] Luca Aceto and Matthew Hennessy. Adding action refinement to a finite process algebra. *Inf. Comput.*, 115(2):179–247, 1994.

[70] André Arnold. *Finite transition systems: semantics of communicating systems.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994. Translator-Plaice, John.

[71] G. D. Plotkin. A structural approach to operational semantics, 1981.

[72] B. Kiepuszewski W.M.P. van der Aalst, A.H.M. ter Hofstede and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases, 14(3)*, July 2003.

[73] Julie Vachon, Nicolas Guelfi, and Alexander Romanovsky. Using coala for the development of a distributed object-based application. In *2nd International Symposium on Distributed Objects & Applications(DAO'00)*, Antwerp, Belgium, 2000.

[74] J. Vachon, D. Buchs, M. Buffo, G. Di Marzo Serugendo, B. Randell, A. Romanovsky, R.J. Stroud, and J. Xu. Coala - a formal language for coordinated atomic actions. Technical report, In 3rd Year Report, ESPRIT Long Term Research Project 20072 on Design for Validation. LAAS, France, November 1998.

[75] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD International Conference on Management of Data*, 1987.

[76] Panos K. Chrysanthis and Krithi Ramamritham. Acta: The saga continues. *Database Transaction Models for Advanced Applications*, 1992.

[77] H. Waechter and A. Reuter. The contract model. *Database Transaction Models for Advanced Applications*, 1991.

[78] IBM DeveloperWorks. Web services transactions specifications.

[79] OASIS. Business transaction protocol.

[80] SUN Microsystems. Web services transaction management specification (ws-txm) v1.0.

[81] W.M.P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. *In Business Process Management: Models, Techniques, and Empirical Studies, volume 1806 of LNCS, pp. 161-183.*, July 2000.

[82] Ali Al-Shabibi, Didier Buchs, Mathieu Buffo, Stanislav Chachkov, Ang Chen, and David Hurzeler. Prototyping object oriented specifications. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003), LNCS*, volume 2679, pages 473–482. Springer-Verlag, June 2003.

[83] Stanislav Chachkov and Didier Buchs. From an abstract object-oriented model to a ready-to-use embedded system controller. In *Rapid System Prototyping, Monterey, CA*, pages 142 – 148. IEEE Computer Society Press, June 2001.

[84] Stanislav Chachkov and Didier Buchs. Interfacing software libraries from non-deterministic prototypes. In *Rapid System Prototyping, Darmstadt, DE*, pages 92 – 98. IEEE Computer Society Press, July 2002.

[85] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach, 2000.

[86] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[87] AspectJ. Eclipse.

[88] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[89] Robin Milner. The polyadic -calculus: A tutorial. *Logic and Algebra of Specification*, 1993.

[90] C. A. R. Hoare. *Communicating sequential processes (Prentice-Hall International series in computer science)*. Prentice/Hall International, April 1985.

[91] Wolfgang Reisig. *Petri Nets. An Introduction*, volume 4. Springer-Verlag GmbH, 1985.

[92] Jörg Desel and Wolfgang Reisig. Place/transition petri nets. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 122–173. Springer, Berlin / Heidelberg, June 1998.

[93] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*.

[94] E. Best, editor. *Nonsequential processes*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[95] Stanislav Chachkov and Didier Buchs. From formal specifications to ready-to-use software components: The concurrent object-oriented petri net approach. In *International Conference on Application of Concurrency to System Design, Newcastle*, pages 99 – 110. IEEE Computer Society Press, June 2001.

[96] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. *SIGPLAN Not.*, 28(10):411–428, 1993.

[97] Open Service Oriented Architecture (OSOA). Service component architecture (sca). http://www.osoa.org/display/Main/Service+Component+Architecture+Home.

[98] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[99] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Inf. Softw. Technol.*, 50(12):1281–1294, 2008.

[100] Ivo Raedts, Marija Petković, Yaroslav S. Usenko, Jan M. van der Werf, Jan F. Groote, and Lou Somers. Transformation of bpmn models for behaviour analysis. In Juan C. Augusto, Joseph Barjis, and Ulrich U. Nitsche, editors, *MSVVEIS*, pages 126–137. INSTICC press, 2007.

[101] Seleniumhq.org. Selenium.

[102] Didier Buchs, Levi Lucio, and Ang Chen. Model checking techniques for test generation from business process models. In *Ada-Europe*, pages 59–74, 2009.