Petri Nets Based Formalization of Textual Use Cases

Stéphane S. Somé
School of Information Technology and Engineering (SITE) University of Ottawa
800 King Edward, P.O. Box 450, Stn. A Ottawa, Ontario, K1N 6N5, Canada
ssome@site.uottawa.ca

Abstract

A Use Case is a specification of interactions involving a system and external actors of that system. The intuitive, user centered nature of textual use cases is one of the reasons for the success of the use case approach. A certain level of formalization is however needed to automate use case based system development, including tasks such as model synthesis, verification and validation. In this paper, a formalization of textual use cases is proposed. At the syntactic-level, an UML metamodel and a restricted-form of natural language are defined for use case description. Use cases execution semantics are proposed as a set of Mapping Rules from well-formed use cases to Basic Petri nets. The semantics consider use cases sequencing constraints defined at the syntactic-level. The proposed formalization serves as a basis for state-model synthesis from use cases. UML activity diagrams are generated to capture use cases sequencing and UML StateCharts to capture event flows within use cases.

Keywords: Use Cases, Petri nets, StateCharts, Activity diagrams, Synthesis.

1 Introduction

A use case is defined in the Unified Modeling Language (UML) specification as "the specification of a sequence of actions, including variants that a system (or a subsystem) can perform, interacting with actors of the system" [21]. Since their introduction by Jacobson [11], use cases are used to drive the development process from the early stages of business modeling to acceptance testing [10]. The UML defines use cases as abstract specification of behaviors. The concrete behavior corresponding to a use case is specified using various behavior description approaches such as interactions, activities, state machines, pre/post-conditions or natural language text.

For practical reasons, and in order to allow for an easy communication with stakeholders, natural language text is usually used for use cases at the early stages of development. While behavior modeling techniques such as interactions, activities and state machines have been subject to intensive formalization efforts, informal representations are essentially used for use cases in textual form. Various *templates* and *guidelines* [2, 3] can be considered as providing some degree of formalization to textual use cases. However, the emphasis is mainly on the structure of use case documents in term of sections, and on directions on the form of natural language. Very little work has been done on the formal definition of textual use cases execution semantics. There are several potential benefits to a more formal definition of textual use cases. As requirements artifacts, it is important to ensure use cases effectively express properties that are agreed up on, that can be verified and that are consistent. Formalization is also required for the automation of tasks such as test derivation or requirements simulation.

The main contribution of this paper is a definition of formal semantics for a textual representation of use cases. We assume the UML definition of use cases including $<<$ *include* $>>$ and $<<$ *extend* $>>$ relations. We also take use case sequencing into consideration. According to the UML, use cases should be "useful on their own". Each use case should specify a unit of useful functionality that a system provides to its users [21]. The functionality provided by a use case is initiated by an actor and must be completed for the use case to complete. It is not always possible to structure use cases such that they are completely independent one from the other. Complex applications often involve several tasks with a high degree of independence, but that need to be synchronized in different ways. As an example, we present an "Online Broker System" in Section 3. Intuitively, an *order* needs to be submitted before *suppliers* can bid. Different policies can then be envisioned for handling bids. For instance, the application requirements might ask for all bids to be received first before one is selected, or a selection may be made as bids are received. It is not considered a good practice to use UML relationships to structure use cases in a way that these types of sequencing constraints are obtained. The resulting use case models suffer from the functional decomposition problem [7] that makes use cases difficult to maintain and pushes implementation toward non-object-oriented paradigms.

We chose Basic Petri nets [22] to express use case execution semantics. This choice is motivated by the possibility to use the same formalism for the execution semantics of both single and sequentially related use cases. Petri nets allow modeling of state based concurrent systems and are well suited to the specification of synchronization issues. They are supported by a variety of analysis and simulation tools. The Petri nets semantics are used as a basis for state model synthesis from use cases. One of our goals in doing so is to validate the defined semantics. Another objective is to provide a translation from textual use cases to an UML-based graphical representation. The generated state models capture single use case behavior as UML *StateCharts*, and use cases sequential relations as UML *Activity Diagrams* [21]. A state model obtained from related use cases integrates behaviors defined separately in a single graphical model that can be simulated and analyzed for consistency and completeness.

The remainder of this paper is organized as follow. We discuss some related works in the next section. In section 3, we introduce use case models and present an abstract and a concrete syntax for textual use cases. Use case syntax is based on our previous works in [25, 27]. The Petri nets semantics of use cases are presented as a set of Mapping Rules in section 4. These semantics are used in section 5 for the synthesis of state models from use cases. Finally we conclude the paper in section 6.

2 Related works

Different works related to use cases are based on a variety of use case notations with formal semantics. In [23, 16, 24] use cases are defined in relation to formal pre/postconditions. The formalization allows derivation of conceptual models in [16], and test generation in [24]. Interaction models including UML Sequence Diagrams [21], Message Sequence Charts (MSCs) [9] and Live Sequence Charts (LSCs) [4] are frequently used formalisms for use cases description. Each use case is considered as a set of related scenarios, and these scenarios are elaborated using interactions. There are well defined semantics for the common notations for interactions such as UML Sequence Diagrams, MSCs and LSCs. The main problem tackled by interaction-based approaches concerns scenarios relation within and between use cases. Most of the scenario-based approaches use formal state models as a representation of the integrated behavior defined by related scenarios. A recent survey listed 21 such approaches [17]. Proposed solutions for relating scenarios include: annotation of scenarios with state labels [14, 13, 28], the use of a high-level representation of sequencing relations between scenarios [8, 15, 28, 33, 32], and operation predicates [26, 31].

In our work, we represent use cases using a restricted form of the natural language. Other automated approaches based on restricted natural language include [18, 19]. In [18], use cases adhering to defined language structures are parsed to automatically extract information for UML class and interaction diagrams generation. Specific syntactic forms are also assumed for use cases in [19]. A statistical parser is used to identify the different types of actions making up use case steps, and an executable regular expression representation of the use case is produced. There no explicit mention of operational semantics in [18, 19]. It can be deduced from the synthesized models that control flow semantics are assumed. The basic operational semantics considered for use cases in this paper is similar. Our goal in this paper is to explicitly state these semantics. In addition, unlike [18, 19], we consider UML use case inclusion and extension relationships as well as use case sequencing constraints.

This paper builds from previous works of ours where we defined an abstract syntax and a concrete syntax for use cases [25], and proposed use cases sequencing constructs [27]. The abstract syntax is presented as a meta-model and the concrete syntax is a restricted form of natural language. These previous results are summarized in section 3 in part in order to ensure this paper is self-contained, but also to account for updates to the use case syntax. In [25], we presented an algorithm for StateChart generation from use cases. Differently to this paper, StateChart generation in [25] is based on a formal description of operations using *added/withdrawn* predicates. In the present work our focus is on control flows semantics of use cases. State model synthesis is performed without the need for operations specification.

This paper discusses StateChart generation from Petri nets. A work related to this matter is presented in [5], where a structure preserving algorithm that translates Petri nets to StateCharts is introduced. We do not use this algorithm in part because it does not consider compound StateChart transitions. However, we show that Petri nets obtained from "consistent" use cases satisfy the required properties listed in [5] for the obtainment of equivalent StateCharts.

3 Use Cases Modeling

There are two levels in use cases description according to the UML. The use case model level concerns use cases abstracted from internal details, and shows use cases relations within the system's environment and each with the other. The second level concerns the description of use cases internal interactions.

3.1 Use Cases model

A UML use case model consists of use cases, actors and relationships. We formally define a use case model as a tuple [*Act*,*Uc*, *Rel*, *InitialUc*] with: *Act* a set of actors, *Uc* a set of use cases, $Rel = Rel_{actuc} \cup Rel_{inc} \cup Rel_{ext}$ a set of relations and $InitialUc \subset Uc$ a set of *initial* use cases. Actors are entities in the domain model that interact with the system. Use cases are descriptions of these interactions. Each use case is initiated by an actor and is related to the achievement of a goal of interest to this actor or other actors in the system. The set of relations *Rel* includes relationships between actors and use cases (Rel_{actuc}) and relationships between use cases. The former are defined on domain *Act* and range *Uc*. Relationships between use cases include <<*include*>> relationships (Rel_{inc}) and <<*extend*>> relationships (Rel_{ext}). An <<*include*>> relationship $uc_{base} \times uc_{inc}$ denotes the inclusion of use case uc_{inc} as a sub-process of use case uc_{base} (the *base use case*). An *extend* relationship $uc_{ext} \times econd \times epoints \times uc_{base}$ denotes an extension of a use case uc_{base} as addition of "chunks" of behaviors defined in an *extension use case* uc_{ext}. These chunks of behaviors are included at specific places in the base use case called *extension points* (*epoints*). Each extension is realized under a specific condition (*econd*).

Figure 1 shows an example of use case diagram in the UML notation. The system under consideration is an *Online Broker System*. The goal of the system is to allow customers to find the best supplier for a given order. A customer fills up an online order form and after submission, the system broadcast it to suppliers. We assume three suppliers in this example, "SupplierA", "SupplierB" and "SupplierC". Each supplier after examining the order may decide to decline or submit a bid. Submitted bids are sent back to the broker to be shown to the customer, who eventually asks the system to proceed with a bid. The use case model corresponds to a tuple [*Act*,*Uc*,*Rel*, *InitialUc*] with:

- *Act* = {"Customer","SupplierA", "SupplierB", "SupplierC", "Payment System"}.
- *Uc* = {"Submit order", "Process Bids", "Register Customer", "Handle Payment", "SupplierA bid for order", "SupplierB bid for order", "SupplierC bid for order", "Cancel Transaction"}.

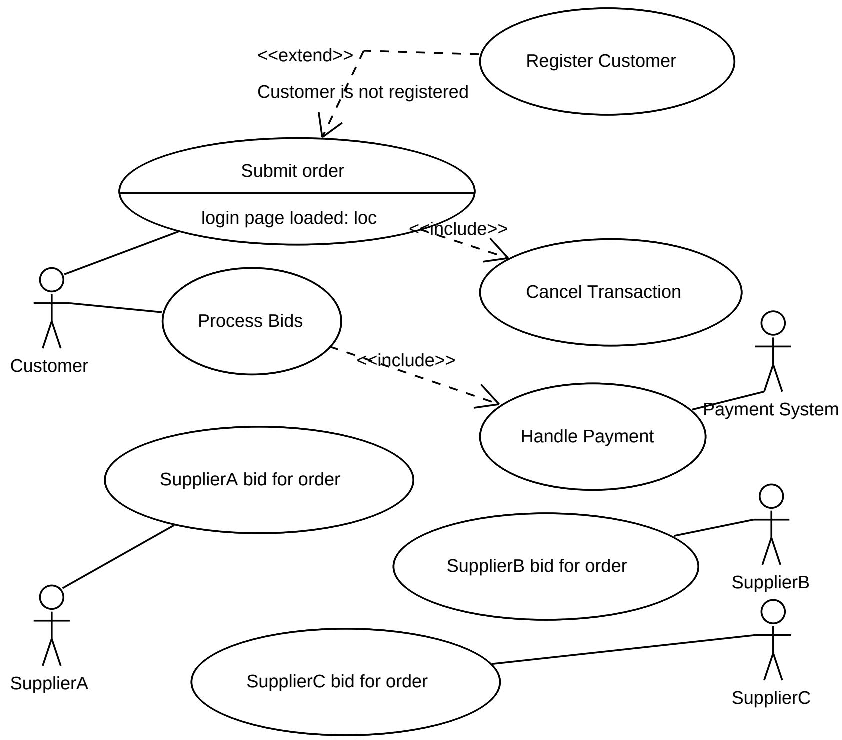


Figure 1: Example of UML Use Case diagram for an Online Broker System.

- A set of relations $Rel = Rel_{actuc} \cup Rel_{inc} \cup Rel_{ext}$.

  Relationships between actors and use cases $Rel_{actuc} = \{$"Customer" $\times$ "Submit order", "Customer" $\times$ "Process Bids", "SupplierA" $\times$ "SupplierA bid for order", "SupplierB" $\times$ "SupplierB bid for order", "SupplierC" $\times$ "SupplierC bid for order", "Payment System" $\times$ "Handle Payment"$\}$.

  The set of include relationships Rel_{inc} is $\{$"Process Bids" $\times$ "Handle Payment", "Submit order" $\times$ "Cancel Transaction"$\}$.

  The set of extend relationships Rel_{ext} is $\{$"Register Customer" $\times$ not($<$*Customer*, *registered*$>$) $\times$ $\{$*ep1*$\}$ $\times$ "Submit order"$\}$. not($<$*Customer*, *registered*$>$) represents the condition under which the extension takes place (we define conditions in section 3.2.3). *ep1* is an extension point defined in reference to use case *Submit order*. We define extension points in section 3.2.1.

- *InitialUc* $= \{$"Submit order"$\}$, assuming use case "Submit order" is enabled by default.

We define *main use cases* as the subset of use cases in a use case model that are not including use cases nor extension use cases.

Definition 1 *Given a use case model* $\mathcal{M}$ = [Act,Uc,Rel = $Rel_{actuc} \cup Rel_{inc} \cup Rel_{ext}$, InitialUc], UC_{main} *the set of* main use cases *of* $\mathcal{M}$ *is such that*

$UC_{main} = \{uc \in Uc |\ \nexists r_i = uc_x \times uc \in Rel_{inc}\ and\ \nexists r_j = uc \times econd \times epoints \times uc_y \in Rel_{ext}\}.$

All initial use cases must be *main* use cases for consistency.

Consistency Rule 1 *Any use case model* $\mathcal{M}$ = *[*Act,Uc,Rel, InitialUc*] with a set of* main use cases UC_{main} *must be such that* $InitialUc \subset UC_{main}$.

Use case diagrams are abstract high-level view of functionality. They do not describe interactions. In the next section, we discuss use cases description. We defined an abstract syntax for use case interactions inspired from Cockburn's template [3]. We also developed a *restricted* form of natural language for concrete representation of use cases.

3.2 Use cases description

Figure 2 shows a UML metamodel for use case description. This metamodel defines an abstract syntax for use cases description as an extension to the UML framework. We specialized class *UseCase* (defined in the UML Specification UseCase package [21]) into *NormalUseCase* and *ExtendUseCase* to account for their differences in description. A *normal use case* defines complete behaviors. Its execution results on either the fulfillment of a goal or an error situation. An *extend use case* specifies a set of behavior *chunks* intended to extend the behavior defined by other use cases. The distinction between normal and extend use cases also allows a more rigorous definition of the <<*include*>> relation by enforcing that *UseCaseInclusion* can only refer to a *NormalUseCase*.

3.2.1 Normal use cases

According to our metamodel, a normal use case description is a tuple [*UCTitle*, *UCPrec*, *UCFoll*, *UCSt*, *UCAlt*, *UCPost*][1] with:

- *UCTitle* a label that uniquely identifies the use case, *UCPrec* a *Constraint* that must be true before an instance of the use case can be executed (the term *constraint* is used in the UML specification to refer to conditions),
- *UCFoll* a possibly null *follow list* specifying which use cases must precede the described use case and how these use cases are synchronized [27],
- *UCSt* a sequence of *use case steps* (*main steps sequence*),
- *UCAlt* a set of *global alternatives* that apply to all the steps in the use case, and
- *UCPost* a condition that must be true at the end of the use case *main scenario* (*success* postcondition).

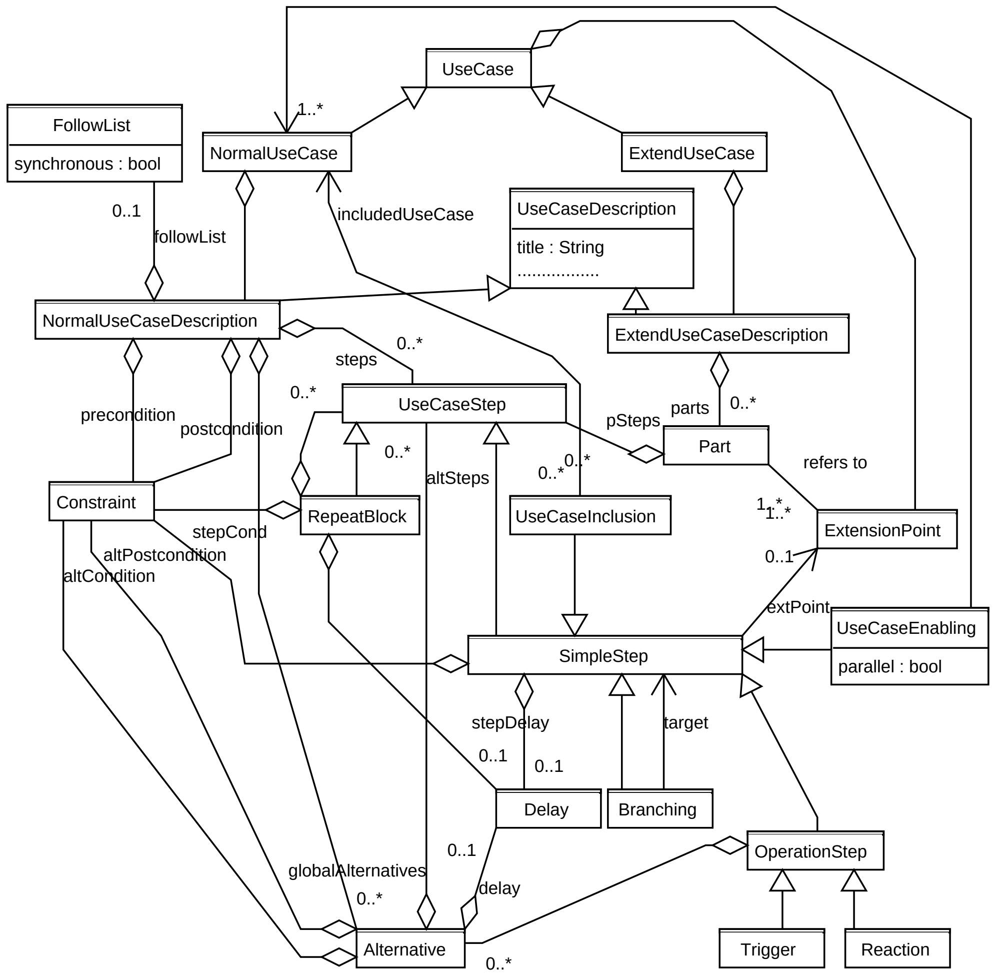


Figure 2: UML metamodel for use cases description.

Figures 3 shows a description of use case "Submit order". The remaining normal use cases in the "Online Broker System" example are shown the Appendix.

A non null *follow list* lists one or more use cases that are "followed" by the described use case. For instance, after a Customer has submitted an order, the Broker System broadcasts it to the suppliers (*SupplierA*, *SupplierB* and *SupplierA* in this case). Therefore use cases *SupplierA Bid*, *SupplierB Bid* and *SupplierC Bid* each follow use case *Submit order*. The *follow list* of these use cases is used to capture these sequencing requirements. When multiple use cases are listed in a *follow list*, these use cases are specified as an expression reflecting how they should be synchronized in relation to the described use case. We use the two

[1]Note that other elements are included in the use case template. However, only the elements above are relevant to this paper.

Title: Submit order
System Under Design: Broker System
Precondition: The Broker System is online and the Broker System welcome page is being displayed
Success Postcondition: An Order has been broadcasted
STEPS

1. The Customer loads the login page
2. The Broker System asks for the Customer's login information
3. The Customer enters her login information
4. The Broker System checks the provided login information
5. IF The Customer login information is accurate THEN The Broker System displays an order page
6. The Customer creates a new Order
7. Repeat while the Customer has more items to add to the Order
   7.1. The Customer selects an item
   7.2. The Broker System adds the selected item to the order
8. The Customer submits the Order
9. The Broker System broadcast the Order to the Suppliers
10. enable in parallel use cases SupplierA bid for order, SupplierB bid for order, SupplierC bid for order

ALTERNATIVES

*1.

*1a. The Customer selects cancel operation

*1b. The Broker System displays the welcome page

2a. after 60 seconds

2a1. The Broker System displays a login timeout page

4a. The Customer login information is not accurate

4a1. GOTO Step 2.

8a. The Order is empty

8a1. The Broker System displays an error page

EXTENSION POINTS

STEP 1. login page loaded

Figure 3: Description of use case "Submit order" in the Online Broker System.

operators AND and OR with the following meaning. Operator AND expresses synchronization (*synchronized follow list*) while OR captures asynchronism (*unsynchronized follow list*). More formally, given use cases uc_0, uc_1, uc_2, $\cdots$, uc_n, the following interpretation is given.

- If the *follow list* of uc_0 is specified as "$uc_1\, AND\, uc_2\, AND\, \cdots uc_n$", all of uc_1, uc_2, $\cdots$, uc_n must reach a point from which use case uc_0 is *enabled* before use case uc_0 (synchronism).
- If the *follow list* of uc_0 is specified as "$uc_1\, OR\, uc_2\, OR\, \cdots uc_n$", use case uc_0 may be executed as soon as any of use cases uc_1, $\cdots$, uc_n reaches a point from which use case uc_0 is *enabled* (asynchronism).

As an example, the *follow list* of use case "Process Bids" described in Figure 22 is *unsynchronized*. A Customer may proceed with a bid as soon as a supplier has replied without waiting for other replies.

3.2.2 Extension use cases

An extension use case includes one or more *parts*. These parts are inserted at specific *extension points* in a *base use case* as realization of an *extend* relationship. An extension use case is formally a tuple [*UCTitle*, *Parts*] with: *UCTitle* as previously defined and *Parts* a set of parts. Each part is a tuple [*ExtPoint*, *Steps*]. *ExtPoint* is a reference to an extension point and *Steps* a sequence of steps (a *part steps sequence*). As an example, Figure 4 shows an extension use case titled *Register Customer* with one part. The use case diagram in Figure

Title: Register Customer
EXTENSION USE CASE PARTS
PART 1. At Extension Point login page loaded

1.1. Customer selects registration operation

1.2. Broker System asks for Customer name, date of birth and address

1.3. Customer enters registration information

1.4. Broker System validates Customer information

1.5. Broker System generate login information for Customer

ALTERNATIVES

1.4.a. Customer registration information is not valid

1.4.a.1. Broker System displays registration failure page

Figure 4: Extension use case "Register Customer".

1 defines an extend relationship between *Register Customer* and the base use case *Submit*

order. The meaning of the relationship is that use case *Register Customer* extends use case *Submit order* when condition "*Customer is not registered*" holds.

3.2.3 Conditions

Each condition is formally a *predicate*, the negation of a predicate, conjunction of predicates or disjunction of predicates. A predicate is a pair $<E, V>$ where E is an *entity* and V a *value*. Entities refer to *concepts* (actors or the system under consideration) or *attributes* of *concepts*. For instance use case *Submit order* precondition is formally the conjunction of predicates <"Broker System","online"> and <"Broker System"."welcome page", "displayed">. In our concrete syntax [25], predicates are represented in the form
"*name of entity*" "*verb*" "*possible value of entity*" with "*verb*" a conjugated form of a limited number of verbs including to be and to have. A *domain model* that enumerates all the entities in the application and their possible values is needed for parsing. A substantial part of this model is obtained by pre-processing use cases [20].

3.2.4 Use case steps

Use case steps include *simple steps* and *repeat blocks*. A repeat block denotes an iterative execution of a sequence of use case steps (*repeat steps sequence*) according to a condition and/or a time delay. Repeat blocks are introduced with keywords repeat, while and until. For instance, step *7* of use case *Submit order* is a repeat block with two steps that are iterated according to a condition. More formally, we define a repeat block as a tuple [*RGuard*, *RDelay*, *RSteps*] with *RGuard* a possibly null *guard* condition, *RDelay* a possibly null *delay* and *RSteps* a sequence of steps.

We distinguish the following types of simple steps: *operation steps*, *branching statements*, *use case inclusion directives* and *use case enabling directives*. A simple step may be constrained by *explicit* and *implicit guards* and/or a *delay*. A *guard* is a condition that must hold for the step to be possible. We introduce explicit guards using keywords if .. then. As an example, step *5* in use case *Submit order* is constrained by an explicit guard. In addition to explicitly specified *guards*, a step st_i is also constrained by implicit guards that include the negation of the conjunction of all the conditions of the step st_{i-1} alternatives (if any). Step st_{i-1} being a step immediately preceding st_i in the same steps sequence. For instance, although step *9* in use case "Submit order" does not include an explicit guard, it includes condition "the order is NOT empty" as implicit guard (the negation of alternative *8a* condition). A delay specifies a minimum time amount that must pass before a step is possible. Delays are counted from the completion moment of the previous step or from when the use case became enabled when applied to the first step of a use case. We introduce delays using keyword after.

An *operation step* denotes the execution of an operation by an actor in the environment of the system (a *trigger*) or the system itself (a *reaction*). Steps *1* and *3*, *6*, *7.1*, and *8* in use case *Submit order* correspond to triggers while steps *2*, *4* and *5*, *7.2* and *9* correspond to reactions. Triggers and reactions are distinguished based on information in the domain model. Our concrete syntax [25], also assume that operations are declared in the domain model according to the format "*action_verb* [*action_object*]". Where the *action_verb* is a verb in infinitive and the *action_object* refers to a concept or an attribute of a concept affected by the action. As an example, "load login page" is an operation name where the action verb is "load" and the action object is "login page". Given this naming convention, an *operation step* has the following form:

"*name of concept*" "*action_specification*" ["*preposition*" "*action_participant*"][2]

The "*action_specification*" has the form

"*conjugated_action_verb*" ["*action_object*"]

The "*conjugated_action_verb*" is the "*action_verb*" used in the concept operation declaration in the present tense.

An operation step may be associated with an *extension point*; a label that references a particular point in a use case where interactions defined in extension use cases may be inserted. An extension point corresponds to the point of execution reached after a successful completion of the operation. Step *1* of use case "Submit order" is associated with an extension point labelled "*login page loaded*". An operation step may also be associated with one or more *alternatives*. An alternative specifies a possible continuation of a use case after a step. Alternatives are used to describe exceptions, error situations or less common courses of events. Formally an alternative is a tuple [*Acond*, *Adelay*, *Asteps*, *Apost*] with *Acond* a constraint that must be true for the alternative to be possible, *Aftd* a delay, *Asteps* a sequence of use case steps (*alternative steps sequence*) and *Apost* an alternative postcondition. Our use case notation allows *global alternatives*. For instance, use case "Submit order" includes a global alternative labelled "*1". A global alternative *galt* is equivalent to the inclusion of *galt* to each operation step where *galt* is possible. In use case "Submit order", alternative **1* effectively only applies to steps corresponding to *triggers*, because its condition (*Acond*) and delay (*Adelay*) are both null.

A *branching statement* includes a reference to a step st_i such that the flow of use case events continues from st_i whenever the *branching statement* is interpreted. We refer to st_i as the branching statement *target*. Step *4a1* in use case "Submit order" branches to step *2*. Execution of that step would result in the flow of events continuation from step *2*.

A *use case inclusion directive* is a realization of an *include* relationship between the use case and an included use case referred in the directive. As an example, step *3* in use case "Process Bids" is an inclusion directive referring to use case "Handle Payment" shown in Figure 23.

A *use case enabling directives* allows to explicitly state control flow between use cases. The directive can refer to one or more use cases. In the latter case, the directive may specify whether the enabled use cases execute concurrently (with keyword in parallel or alternatively. For instance, step *10* in use case "Submit order" is an enabling directive for use cases "SupplierA bid", "SupplierB bid" and "SupplierC bid". These three use cases may

[2] Elements between "[]" are optional.

execute concurrently. Only one use case may execute in the non-parallel variant of the use case enabling directive. As soon as a use case start executing, all other use cases referred to in the directive are considered disabled. For consistency we assume the first events of the enabled use cases are distinct enough that an unambiguous choice can be made among the use cases. More formally, suppose $\mathsf{firstEvent}$ is a function such that given a use case *uc* $\mathsf{firstEvent(uc)}$ is the first event of *uc*, the following Consistency Rule must be satisfied.

Consistency Rule 2 *The set $UC_{edir} = \{uc_1, \cdots, uc_n\}$ of use cases referred to in a non-parallel* `enabling` *directive* edir *must be such that* $\forall uc_i \in UC_{edir}, \nexists\ uc_j\ (i \neq j) \in UC_{edir}$ *such that* $\mathsf{firstEvent}(uc_i) = \mathsf{firstEvent}(uc_j)$*.*

A use case may enable itself and there is no restriction to where the directive may appear. Notice that *follow lists* and use case enabling directives depend each on the other. The following rule must be satisfied for consistency.

Consistency Rule 3 *When a use case uc_i refers to a use case uc_j in its* follow list*, use case uc_j must include an enabling directive referring to use case uc_i. Conversely, for each use case referred to in an enabling directive, the enabled use case must include the enabling use case in its* follow list*.*

A use case enabling directive may be followed by subsequent steps. These steps are assumed to execute concurrently with the enabled use cases.

3.2.5 Steps Sequences

A *steps sequence* is a succession of steps in a use case. A normal use case [*UCTitle*, *UCPrec*, *UCFoll*, *UCSt*, *UCAlt*, *UCPost*] includes a *main steps sequence UCSt*. The sequence of steps *Asteps* in an alternative [*Acond*, *Adelay*, *Asteps*] is an *alternative steps sequence*. We also distinguish *repeat steps sequences* (*RSteps* in a repeat block [*RGuard*, *RDelay*, *RSteps*]) and *part steps sequences* (*Steps* in an extension use case part [*ExtPoint*, *Steps*]).

A main use case must be initiated by an actor [21]. Consequently the following consistency rule needs to be satisfied.

Consistency Rule 4 *The first step in a* main steps sequence *must be a* trigger*.*

For consistency, a *branching statement* must not connect unrelated steps sequences. We define a *subordinate* relation sub between steps sequences as follow.

Definition 2 *A* steps sequence *sep' is subordinate of a* steps sequence seq *($sep'\,\mathsf{sub}\,seq$) if:*

- seq *includes a step* st *such that sep' is a steps sequence of an alternative of* st*,*
- seq *includes a repeat block* rb *such that sep' is* rb*'s* repeat steps sequence
- *there is a* steps sequence *sep'' such that ($sep'\,\mathsf{sub}\,sep''$) and ($sep''\,\mathsf{sub}\,sep$).*

Consistency Rule 5 *For any* branching statement $step_o$ *such that* $step_d$ *is* $step_o$ *target, let* seq_o *be a* steps sequence *such that* $step_o \in seq_o$ *and* seq_d *be a* steps sequence *such that* $step_d \in seq_d$*,* seq_o *and* seq_d *must be such that* $seq_d = seq_o$ *or* $seq_d \mathsf{sub}\, seq_o$*.*

A *branching statement* must also be last in a steps sequence as subsequent steps are unreachable.

Consistency Rule 6 *For any* steps sequence $seq = step_0 \cdots step_n$*, if* $\exists i$ *such that* $step_i$ *is a* branching statement*, then* i *must be equal to* n*.*

A normal use case may include a precondition and a set of postconditions (a success postcondition and alternative postconditions). Preconditions and postconditions are implicit specifications of use case sequencing constraints. A use case precondition is a *condition* that needs to hold before the use case, while a use case postcondition is a *condition* that is guaranteed to hold at the end of the use case. Assuming functions pre and post such that: $\mathsf{pre(uc)}$ is the precondition of a use case *uc* and given *seq* a *main* or *alternative* steps sequence in *uc*, $\mathsf{post(seq,\ uc)}$ is the postcondition associated with *seq*, use case uc_1 enables use case uc_2 (and uc_2 follows uc_1) if there is a steps sequence seq_i in uc_1 such that $\mathsf{post}(seq_i, uc_1) \Rightarrow \mathsf{pre}(uc_2)$. For instance, by analyzing the pre and postconditions of use cases in Figure 3 - 21, we can infer that use case “Submit order” could be followed by use cases “SupplierA Bid”, “SupplierB Bid” and “SupplierC Bid”. Preconditions and postconditions allows to specify which use case must precede a given use case and which use cases are enabled after a use case. However, they do not offer a possibility to specify how use cases are synchronized or whether several enabled use cases execute concurrently or alternatively [27]. Additionally, since postconditions are only checked at the end of a steps sequence, situations where enabling directives are not the last elements of steps sequences do not have an equivalent. Nevertheless, explicit constraints specified by use case sequencing constructs and implicit constraints derived from pre/postconditions must comply with the following consistency rules. We assume functions pre and post as above.

Consistency Rule 7 *Given use cases* uc_i *and* uc_j*, if* uc_i *is included in the* follow list *of* uc_j*, there must exist a steps sequences* seq_k *in* uc_i *such that* $\mathsf{post}(seq_k, uc_i) \Rightarrow \mathsf{pre}(uc_j)$ *(notice that in accordance with rule 3,* seq_k *must include an enabling directive referring to use case* uc_j*).*

Consistency Rule 8 *Given use cases* uc_i *and* uc_j*, if there is a steps sequence* seq_k *in* uc_i *such that* $\mathsf{post}(seq_k, uc_i) \Rightarrow \mathsf{pre}(uc_j)$*, use case* uc_i *must be included in the* follow list *of* uc_j *and there must be a use case enabling directive referring to* uc_j *in the steps sequence* sc_i*.*

4 Formal interpretation of use cases

A use case captures a set of event sequences. We represent these event sequences using the Basic Petri nets formalism.

4.1 Basic Petri nets

A Basic Petri net is formally defined as follow.

Definition 3 *A Basic Petri net (P/T net) is a triple [P, T, F] with:*

- P *a finite set of places,*
- T *a finite set of transitions,*
- $F \subseteq (P \times T) \cup (T \times P)$ *a flow relation such that for each transition* $t \in T$*, the* input places *of* t *(denoted* $\bullet$t*) are all places* p_{in} *such that* $p_{in} \times t \in F$*, and the* output places *of* t *(denoted* t$\bullet$*) are all places* p_{out} *such that* $t \times p_{out} \in F$*.*

The set of use cases in a use case model $\mathcal{M}$ corresponds to a P/T net Pn = [P, T, F], and each main use case in $\mathcal{M}$ corresponds to a *subnet* of Pn defined as follow.

Definition 4 *A* subnet *of a P/T net* Pn = *[P, T, F] is a P/T net* $Pn' = [P', T', F']$ *such that* $P' \subset P, T' \subset T, F' \subset F$ *and* $\forall t \in T', \bullet t \subset P', t\bullet \subset P'$*.*

We use the terminology *use case P/T net* to refer to a subnet corresponding to a use case. The *initial place* of a use case P/T net as an input place to the first transition corresponding to the first step of the use case. Transitions correspond to events depicted by use cases and places represent states reached between these events. A step in a use case is typically mapped to a sequence of transitions. Figure 5 is a P/T net representation of the set of event sequences corresponding to use case "*Submit order*" extended with use case "*Register Customer*". We use the corresponding step numbers as labels for transitions. c1 is condition <*Customer,registered*>, c2 is condition <*Customer.registration,valid*>, c3 is condition <*Customer.'login information',accurate*>, c4 is condition <*Customer,'more items to add'*> and c5 is condition <*Order,empty*>.

The token game semantics governs P/T nets animation. Places may contain *tokens* and a transition t is said to be *enabled* if all $\bullet t$ contain *tokens*. An enabled transition t may *fire* by removing a token from each of its input place and adding a token to each of its output place. A *marking* M is the distribution of tokens over places in a P/T net. Each marking represents a state of the described system. The *initial marking* of a P/T net corresponding to a use case model $\mathcal{M}$ = [*Act,Uc,Rel, InitialUc*] is such that there is a token in all the initial places of the use cases in *InitialUc* and every other place is empty. More formally, M_i a marking of a use case model P/T net [P, T, F] is a function $M_i : P \rightarrow \mathbb{N}$ such that $\forall p \in P M_i(p)$ is the number of tokens in place p. M_0 the *initial marking* of a use case model P/T net is defined as follow. We suppose initialp is a function such that initialp(uc) is the initial place of use case uc P/T net.

$$M_0(p) = \begin{cases} 1 & if\ (\exists uc \in \mathit{InitialUc} | \mathsf{initialp}(uc) = p) \\ 0 & \text{otherwise} \end{cases}$$

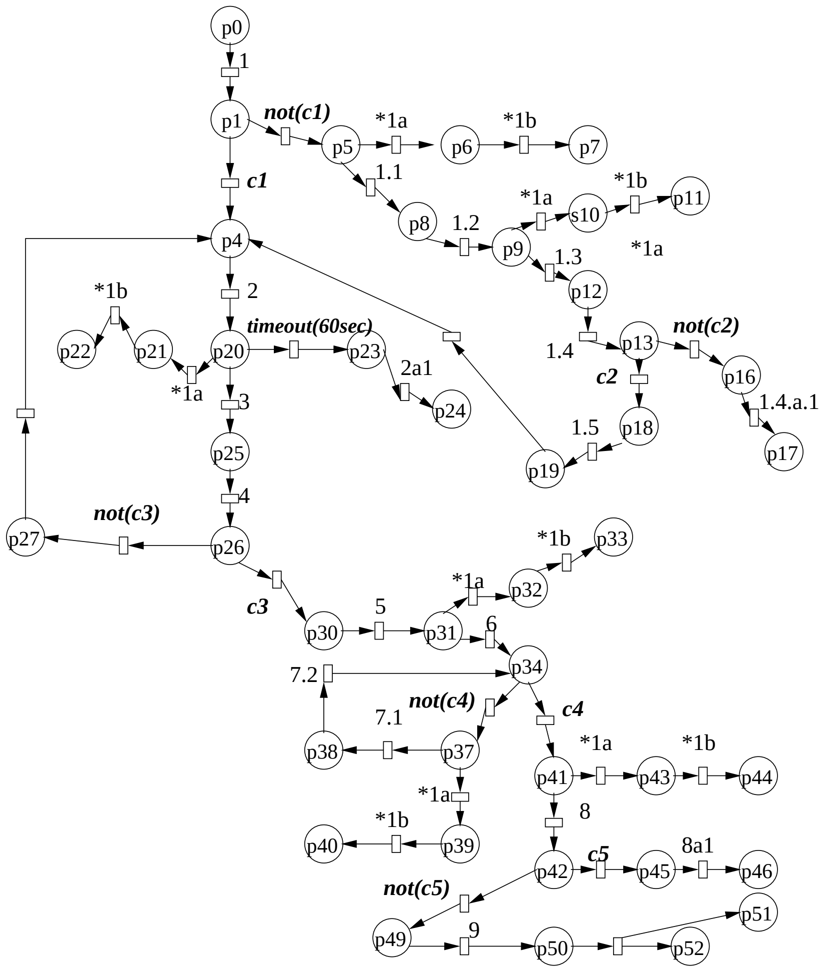


Figure 5: Petri net representation of use case "Submit order" set of possible event sequences.

Each subsequent marking M_i $(i > 0)$ is derived from its preceding marking M_{i-1} after the firing of an enabled transition t as follow:

$$M_i(p) = \begin{cases} M_{i-1}(p) - 1 & if\ (p \in \bullet t\ and\ p \notin t\bullet) \\ M_{i-1}(p) + 1 & if\ (p \in t\bullet\ \ and\ p \notin \bullet t) \\ M_{i-1}(p) & \text{otherwise} \end{cases}$$

Each marking (M_i) is a possible state of the system. An execution trace is a sequence of events obtained from the sequence of transitions firing starting with a P/T net initial marking.

4.2 Events

Transitions in a use case P/T net correspond to the occurrence of events. We distinguish *trigger events* (corresponding to *trigger transitions*), *reaction events* (corresponding to *re-*

action transitions), *decision events* (corresponding to *decision transitions*), *timeout events* (corresponding to *timeout transitions*) and a *null event* (corresponding to *null transitions*). The set of events depicted by a use case is partially ordered. Events derived from a same steps sequence are totally ordered.

A *trigger event* denotes the execution of an operation by an actor in the environment, while a *reaction event* denotes the execution of an operation by the system under consideration. Although execution of an operation takes certain duration, we consider that an operation event is discrete and correspond to the moment of completion of the operation. A *decision event* occurs whenever one of different associated *conditions* evaluates to true. For instance, in Figure 5, decision events for conditions $<$*Customer*, *registered*$>$ and not($<$*Customer*, *registered*$>$) are possible after operation event *loads login page*. Suppose predicate $<$*Customer*, *registered*$>$ evaluates to true, the corresponding decision event would be produced and the behavior would progress along the transition labelled by that event. We assume that each place is associated with a *timer* that is started whenever a transition enters that place and stopped whenever a transition leaves the place. A *timeout event* $\mathsf{timeout(d)}$ is produced when a delay d elapses after a *timer* has started and hasn't been stopped.

We suppose reactive semantics similar to [6] for use case P/T nets. There is no control over when actors operations (triggers) are performed. System operations (reactions), the null event as well as decision events however, happen as soon as possible. Timeout events are delayed by a delay value such that other events may occur before they are fired. As a consequence, triggers and timeouts are ignored from places when there is at least one outgoing transition corresponding to a reaction, decision or null event. More formally, let $\mathsf{systTrans}$, $\mathsf{trigTrans}$, $\mathsf{timeTrans}$, $\mathsf{decTrans}$ and $\mathsf{nullTrans}$ be functions such that given a P/T net [P, T, F] and a place $p \in P$, $\mathsf{systTrans}(p)$ is the set of transitions t_r such that $p \times t_r \in F$ with t_r a reaction transition, $\mathsf{trigTrans}(p)$ is the set of transitions t_t such that $p \times t_t \in F$ with t_t a trigger transition, $\mathsf{decTrans}(p)$ is the set of transitions t_c such that $p \times t_c \in F$ with t_c a decision, and, $\mathsf{nullTrans}(p)$ is the set of transitions t_n such that $p \times t_n \in F$ with t_n a null transition.

Definition 5 *Given a P/T net [P, T, F], an event* ev *is* ignored *if* ev *corresponds to a transition* t *from a place* p *with $t \in$ trigTrans(*p*) or $t \in$ timeTrans(*p*), and $\exists t'$ such that $t' \in$ decTrans(*p*) or $t' \in$ nullTrans(*p*).*

Additionally, a timeout event with delay d is ignored from a place in presence of another timeout event with a delay less than d.

Definition 6 *Given a P/T net [P, T, F], a timeout event* ev *is* ignored *if* ev *corresponds to a transition $t_{ev} \in$ timeTrans(*p*) with delay* d, *and $\exists t'_{ev} \in$ timeTrans(*p*) with delay d' such that $d' < d$.*

Another consequence of the reactive semantics is that behavior depicted in a use case P/T net is *non-deterministic* when (1) more than one reaction or null transition start from a place, (2) the set of transitions starting from a place includes decision transitions mix with reaction/null transitions, (3) more than one transition corresponding to a same trigger starts from a place, or (4) more than one timeout transition starts from a place. More formally,

Definition 7 *The behavior depicted by a P/T net [P, T, F] is* non-deterministic *if* $\exists p \in P$*:*

(1) $|\textsf{\textit{systTrans}}(\mathrm{p}) \cup \textsf{\textit{nullTrans}}(\mathrm{p})| > 1$*, or,*

(2) $|\textsf{\textit{decTrans}}(\mathrm{p})| > 0$ *and* $(|\textsf{\textit{systTrans}}(\mathrm{p}) \cup \textsf{\textit{nullTrans}}(\mathrm{p})| > 0)$*, or,*

(3) $\exists t_t \in \textsf{\textit{trigTrans}}(p), \exists t'_t \in \textsf{\textit{trigTrans}}(p)(t_t \neq t'_t)$ *such that* t_t *and* t'_t *correspond to the same operation, or,*

(4) $\exists t_t \in \textsf{\textit{timeTrans}}(p), \exists t'_t \in \textsf{\textit{timeTrans}}(p)(t_t \neq t'_t)$ *such that* t_t *and* t'_t *correspond to timeout events with the same delay.*

Notice that a P/T net such that none of (1) - (4) is satisfied may still exhibit *non-deterministic* behavior. For instance when decisions events corresponding to different transitions can hold at the same time in a place. The Mapping Rules presented in the next section produce P/T nets devoid of *non-determinism* as defined in Def. 7, when Consistency Rule 9 is satisfied (a proof to that is discussed in Section 4.4).
Let function $\textsf{alternatives}$ be such that given a step $step_i$, $\textsf{alternatives}(step_i)$ is $step_i$ set of alternatives. Let $\textsf{guard}$ and $\textsf{delay}$ be two functions such that $\textsf{guard}(step_i)$ is the conjunction of $step_i$ explicit and implicit guards (Cf. Def. 9), and $\textsf{delay}(step_i)$ is $step_i$ delay. We assume $\textsf{possible\_triggers}$ and $\textsf{applicable\_delays}$ are two functions defined as follow. Given a step $step_i$, $\textsf{possible\_triggers}(step_i)$ includes all actor operations $astep_{j0}$ such that $alt_j = [Acond_j = \textsf{null}, Adelay_j = \textsf{null}, Asteps_j = astep_{j0}, \cdots, astep_{jm}] \in \textsf{alternatives}(step_i)$ and $step_{i+1} \in \textsf{possible\_triggers}(step_i)$ if $\textsf{delay}(step_{i+1}) = \textsf{guard}(step_{i+1}) = \textsf{null}$ and $step_{i+1}$ is an actor operation. Given a step $step_i$, $\textsf{applicable\_delays}(step_i)$ includes all delays $Adelay_j$ such that $alt_j = [Acond_j = \textsf{null}, Adelay_j \neq \textsf{null}, Asteps_j] \in \textsf{alternatives}(step_i)$ and $step_{i+1} \in \textsf{applicable\_delays}(step_i)$ if $\textsf{delay}(step_{i+1}) \neq \textsf{null}$ and $\textsf{guard}(step_{i+1}) = \textsf{null}$.

Consistency Rule 9 *For each step* $step_i$*:*

a) there must be no $alt = [Acond, Adelay, Asteps = astep_0, \cdots, astep_m] \in \textsf{\textit{alternatives}}(step_i)$ *with* Acond $=$ ***null****,* Adelay $=$ ***null*** *and* $astep_0$ *a* system operation*,*

b) $\forall trig_j \in \textsf{\textit{possible\_triggers}}(step_i)$, $\nexists trig_k \in \textsf{\textit{possible\_triggers}}(step_i)$ *with* $trig_j = trig_k$*,*

c) $|\textsf{\textit{applicable\_delays}}(step_i)| \leq 1$*.*

Part c of Consistency Rule 9 also ensures absence of ignored timeouts events according to Def. 6.

4.3 Mapping of use cases to P/T nets

The event sequences defined by a set of normal use cases in a use case model $\mathcal{M} = [Act, Uc, Rel, InitialUc]$ can be depicted as a P/T net $Pn = [P, T, F]$ according to a set of mapping rules. It is assumed that all use cases in Uc satisfy the Consistency Rules stated through this paper. In the remainder of this Section, P, T and F refer to the elements of the use case model $\mathcal{M}$ P/T net.
Rule 1 specifies that each use case P/T net includes an *initial place*.

Mapping Rule 1 *$\forall Pn_{uc} = [P_{uc}, T_{uc}, F_{uc}]$ such that Pn_{uc} is a use case* Uc *P/T net, $\exists! p_0 \in P_{uc}$ with* **initialp**(Uc) *$= p_0$.*

Each step in Uc corresponds to a sequence of transitions in Pn. We define a *step starting place* as a place from which the behavior defined by a step is considered in a use case P/T net. A *step final place* is the output place of the last transition corresponding to the step. We assume functions startp and finalp such that startp($step_i$) is step $step_i$ starting place and finalp($step_i$) is $step_i$ final place.

Definition 8 *Given a sequence of steps $seq = step_0, \cdots step_n$ from use case* Uc*, if* seq *is a* main steps sequence*, startp($step_0$) = initialp(Uc). For each $step_i$ $(i > 0)$, startp($step_i$) = finalp($step_{i-1}$).*

The starting place of the first step of *alternative steps sequences*, *repeat steps sequences* and *part steps sequences* are determined in Mapping Rules 4, 8 and 10 respectively. Steps final places are determined in Mapping Rules 3, 6, 7 and 8.

4.3.1 Mapping of simple steps

Recall from Section 3.2.1 that a *simple step* may be constrained by implicit and/or explicit guards. Given a steps sequence $seq = step_0, \cdots step_n$, the set of implicit guards of $step_i$ $(0 < i \leq n)$ is determined as follow. We assume ext_point is a function such that ext_point($step_i$) returns the *extension point* associated with $step_i$ if any or null. Given an extension point ep and a use case model $\mathcal{M} = [Act, Uc, Rel = Rel_{actuc} \cup Rel_{inc} \cup Rel_{ext}, InitialUc]$, matchingExtends($ep$, $\mathcal{M}$) returns the set of all <<*extend*>> relations that refer to ep in $\mathcal{M}$. More formally, matchingExtends(ep, $\mathcal{M}$) $= \{r_{ext} | r_{ext} = uc_{ext} \times econd \times epoints \times uc_{base} \in Rel_{ext} \wedge ep \in epoints\}$. Function ext_conds is such that ext_conds($extrels$) returns the set of all *extend conditions* referred to in a set of <<*extend*>> relations *extrels*. More formally, ext_conds($extrels$) $= \{econd | \exists uc_{ext} \times econd \times epoints \times uc_{base} \in extrels\}$.

Definition 9 *The set of* implicit guards *of $step_i$ (implicit_guards($step_i$)) is such that:*

- *If $step_{i-1}$ is a repeat block* rb=[RGuard, RDelay, RSteps]*, implicit_guards($step_i$) = $\{\neg RGuard\}$.*
- *If $step_{i-1}$ is a simple step, let the set $AltConds_{i-1} = \{Acond | \exists Alt = [Acond, Adelay, Asteps, Apost] \in$ alternatives$(step_{i-1})\}$. If $AltConds_{i-1} \neq \emptyset$, let $cc_{alt} = \bigwedge_{cond \in (AltConds_{i-1})} \neg cond$, $cc_{alt} \in$ implicit_guards($step_i$).*

If extrels $=$ *matchingExtends(ext_point($step_{i-1}$), $\mathcal{M}$)* $\neq \emptyset$,
let $cc_{ext} = \bigwedge_{cond \in \textsf{\textit{ext\_conds}}(\text{extrels})} \neg cond$, $cc_{ext} \in$ *implicit_guards*$(step_i)$.

Mapping Rule 2 *Given a step $step_i$ from a use case* Uc*, suppose* p_i = *startp($step_i$),* td *is a* timeout transition *with delay* *delay($step_i$ (if non* *null) and* cg *is a* decision transition *corresponding to* *guard($step_i$) (if non null).*

- *If delay($step_i$) is non null and guard($step_i$) is non null,* $P \supset \{p_j, p_k\}$, $T \supset \{td, cg\}$, $F \supset \{p_i \times cg \times p_j, p_j \times td \times p_k\}$.
- *If delay($step_i$) is non null and guard($step_i$) is null,* $P \supset \{p_k\}$, $T \supset \{td\}$, $F \supset \{p_i \times td \times p_k\}$.
- *If delay($step_i$) is null and guard($step_i$) is non null,* $P \supset \{p_k\}$, $T \supset \{cg\}$, $F \supset \{p_i \times cg, cg \times p_k\}$.
- *If delay($step_i$) is null and guard($step_i$) is null,* $p_k = p_i$.

Transitions corresponding to the remainder of $step_i$ are added from place p_k in Mapping Rules 3, 5 and 6.

Each operation step and branching statement correspond a transition.

Mapping Rule 3 *For each* operation step $step_i$, $P \supset \{p_l\}$, $T \supset \{op\}$ *and* $F \supset \{p_k \times op, op \times p_l\}$. *With* op *a transition corresponding to the execution of the operation referred in* $step_i$. *Place p_l is the* final place *of $step_i$ (finalp($step_i$) = p_l).*

An operation step may be associated with alternatives. Each alternative specifies how the execution may proceed after the step. As discussed in Section 4.2, *trigger* and *timeout* events are ignored when they conflict with *decision* events. Consequently some of a step alternatives may be ignored.

Definition 10 *An alternative* $alt_i = [Acond_i, Adelay_i, Asteps_i = astep_{i0}, \cdots astep_{ik}, Apost_i] \in$ *alternatives($step_i$)* is ignored *from $step_i$ if:*

(1) $Acond_i$ = *null*, $Adelay_i$ = *null*, $astep_{i0}$ *is a trigger, and there is at least one alternative* $alt_j = [Acond_j \neq \textsf{\textit{null}}, Adelay_j, Asteps_j, Apost_j] \in \textsf{\textit{alternatives}}(step_i)$ *or $step_i$ is associated with an extension point ep_i such that matchingExtends(ep_i, $\mathcal{M}$)* $\neq \emptyset$.

(2) $Acond_i$ = *null*, $Adelay_i \neq$ *null, and there is at least one alternative* $alt_j = [Acond_j \neq \textsf{\textit{null}}, Adelay_j, Asteps_j, Apost_j] \in \textsf{\textit{alternatives}}(step_i)$.

For instance, alternative **1* in use case "*Submit order*" *is ignored* from steps *4* and *8* in accordance with Def. 10-(1).

Mapping Rule 4 *For each* $alt_j = [Acond, Adelay, Asteps = step_0^{alt} \cdots step_m^{alt}, Apost] \in$ *alternatives*$(step_i)$ *such that* alt_j is not ignored *from* $step_i$*, let* p_i^{alt} *be* **finalp**$(step_i)$*:*

- *If* Adelay *is non* **null** *and* Acond *is non* **null***,* $F \supset \{p_i^{alt} \times cg \times p_j^{alt}, p_j^{alt} \times td \times p_k^{alt}\}$.
- *If* Adelay *is non* **null** *and* Acond *is* **null***,* $F \supset \{p_i^{alt} \times td \times p_k^{alt}\}$.
- *If* Adelay *is* **null** *and* Acond *is non* **null***,* $F \supset \{p_i^{alt} \times cg, cg \times p_k^{alt}\}$.
- *If* Adelay *is* **null** *and* Acond *is* **null***,* $p_k^{alt} = p_i^{alt}$.

startp$(step_0^{alt}) = p_k^{alt}$.

Mapping Rule 5 *For each* branching statement $step_i$ *with* target $step_j$*,* $F \supset \{p_k \times tn, tn \times p_j\}$*, with* tn *a* null *transition and* p_j = **initialp**$(step_j)$*.*

As an example, use case "*Submit order*" branching statement *4a1* corresponds to flows $p_{27} \times null$, $null \times p_4$ in Figure 5.

Use case inclusion directives are interpreted as follow. Let uc_{base} be a use case with an include directive *inclUC* and uc_{inc} a use case referred by *inclUC*. Suppose uc_included is a function such that uc_included(*inclUC*) returns uc_{inc}. Interpretation of directive *inclUC* resumes to rewriting of use case uc_{base} replacing *inclUC* with all the steps in uc_included(*inclUC*). We assume copy(Pn) is a copy of a P/T net Pn and copy(p, Pn') is the copy of a place p in Pn'. More formally,

Definition 11 *Given* Pn = [P, T, F]*, copy(*Pn*) is a P/T net* $Pn' = [P', T', F']$ *such that:*

- $\forall p \in P, \exists p' \in P'$*, with* p' *a copy of* p *in* Pn' *(*p' = *copy(*p,Pn'*))*
- $\forall t \in P, \exists t' \in T'$ *with* t' *a transition that corresponds to an occurrence of the same event as* t*,* t' *is a copy of* t *in* Pn' *(*t' = *copy(*t,Pn'*))*
- $\forall p \times t \in F, \exists p' \times t' \in F'$ *with* p' = *copy(*p,Pn'*) and* t' = *copy(*t,Pn'*)*
- $\forall t \times p \in F, \exists t' \times p' \in F'$ *with* p' = *copy(*p,Pn'*) and* t' = *copy(*t,Pn'*)*

A use case inclusion directive corresponds to the inclusion of places and transitions in the P/T net of the including use case according to the following.

Mapping Rule 6 *For each* inclusion directive $step_i$ *referring to an included use case* uc_{inc}*, let* $Pn_{inc} = [P_{inc}, T_{inc}, F_{inc}]$ *be use case* **uc_included**$(step_i)$ *P/T net and* Pn'_{inc} = **copy**(Pn_{inc}) $= [P'_{inc}, T'_{inc}, F'_{inc}]$.

- $P \supset P'_{inc}$*,* $T \supset T'_{inc}$*,* $F \supset F'_{inc}$
- *Let* p_0^{inc} *be* **initialp**(uc_{inc})*,* $T \supset \{tn\}$*,* $F \supset \{p_k \times tn, tn \times$ **copy**$(p_0^{inc}, Pn'_{inc})\}$ *with* tn *a null transition.*

- *Let* $main_{inc} = step_0^{inc} \cdots step_n^{inc}$ *be use case* uc_{inc} main steps sequence; $\mathsf{finalp}(step_i) = \mathsf{copy}(\mathsf{finalp}(step_n^{inc}), Pn'_{inc})$.

A use case enabling directive corresponds to the forking of concurrent execution sequences. One sequence corresponds to the steps that follow the directive if any, while the other sequences correspond to the enabled use cases (Cf. Mapping Rules 12 - 14).

Mapping Rule 7 *For each* use case enabling directive $step_i$, $P \supset \{p_f, p_e\}$, $tn \in T$, $F \supset \{\mathsf{startp}(step_i) \times tn, tn \times p_f, tn \times p_e\}$ *with* tn *a null transition.* $\mathsf{finalp}(step_i) = p_f$. *We consider a relation* $\mathsf{enable\_place}$ *such that* $step_i \times p_e \in \mathsf{enable\_place}$.

For instance, step *10* in use case "*Submit order*" is mapped to flows $p_{50} \times null$, $null \times p_{51}$, $null \times p_{52}$ in Figure 5.

4.3.2 Mapping of repeat blocks

A repeat block [*RGuard*, *RDelay*, *RSteps*] specifies a sequence of iterative steps *RSteps*. Each iteration depends on a guard condition *RGuard* and/or a time delay *RDelay*.

Mapping Rule 8 *Given a repeat block* rblock = [RGuard, RDelay, RSteps $= step_0^r \cdots step_m^r$], *let* p_i *be* startp(rblock), cg *a transition corresponding to condition* RGuard, *and* td *a transition corresponding to a timeout event with delay* RDelay.

- *If* RDelay *is non* **null** *and* RGuard *is non* **null**, $P \supset \{p_j^r, p_k^r\}$, $T \supset \{cg, td\}$, $F \supset \{p_i \times cg \times p_j^r, p_j^r \times td \times p_k^r\}$, finalp(rblock) $= p_i$.
- *If* RDelay *is non* **null** *and* RGuard *is* **null**, $P \supset \{p_k^r\}$, $T \supset \{td\}$, $F \supset \{p_i \times td \times p_k^r\}$.
- *If* RDelay *is* **null** *and* RGuard *is non* **null**, $P \supset \{p_k^r\}$, $T \supset \{cg\}$, $F \supset \{p_i \times cg, cg \times p_k^r\}$, finalp(rblock) $= p_i$.
- *If* RDelay *is* **null** *and* RGuard *is* **null**, $p_k^r = p_i$.

$\mathsf{startp}(step_0^r) = p_k^r$.

Notice that the final and initial places of a repeat block are identical when a non-null guard condition is used. The final place is undetermined when the guard condition is null. In such a case, the repeat block models an endless iterative behavior that can only be terminated with an embedded branching statement.

4.3.3 Use case extension

Use case extension is a mechanism for the expansion a *base* use case with behaviors defined in *extension use cases*. An extension use case *ExUc* is a tuple [*UCTitle*, *Parts*]. Each part in *Parts* in turn is a tuple [*ExtPoint*, *Steps*]

Mapping Rule 9 *An extension use case* Part pt = *[*ExtPoint*,* Steps*] corresponds to a part P/T net [P_p, T_p, F_p].*

Mapping Rule 10 $\forall pt = [ExtPoint, Steps = step^p_0, \cdots step^p_m]$ *a* part *such that* pt *P/T net* $= [P_p, T_p, F_p]$, $\exists p^p_0 \in P_p$ *with* $\textsf{initialp}(\text{pt}) = p^p_0$ *and* $\textsf{startp}(step^p_0) = p^p_0$.

An <<*extend*>> relation specifies a condition under which a use case extension is realized as well as the specific points (*extension points*) in the base use case where the behavior specified in parts are plugged in. Let functions copy, matchingExtends and ext_conds be as previously defined.

Mapping Rule 11 *For each* $step_i$ *such that* $ep_i = \textsf{ext\_point}(step_i) \neq \textsf{null}$ *and* Extrels $=$ $\textsf{matchingExtends}(ep_i, \mathcal{M}) \neq \emptyset$*, let* $Extrels^2$ *be the set of* Extrels *subsets without the empty set,* NCP *be the set* $\textsf{ext\_conds}(\text{Extrels})$ *and let* p_i *be equal to* $\textsf{finalp}(step_i)$*.*
For each $Erels \in Extrels^2$*,*

- *Let* CP *be the set* $\textsf{ext\_conds}(\text{Erels})$, $p_i \times cond \in F$, $cond \in T$ *with* cond *a transition corresponding to condition* $\bigwedge_{(C \in CP)} C \bigwedge_{(NC \in NCP-CP)} \neg NC$
- *For each* rel $= uc_{ext} \times econd \times epoints \times uc_{base} \in Erels$*, with* $uc_{ext} = [$UCTitle, Parts$]$ *and* $part^r = [ep_i, Steps^r = step^r_0 \cdots step^r_m] \in Parts$*. Let* Pn_r *be a P/T net corresponding to* $part^r$ *and* $Pn'_r = [P'_r, T'_r, F'_r] = \textsf{copy}(Pn_r)$*.*
  - $P \supset P'_r$, $T \supset T'_r$, $F \supset F'_r$
  - $F \supset \{cond \times \textsf{copy}(\textsf{initialp}(part^r))\}$
  - *If* $\exists step_{i+1}$, $tn \in T$, $F \supset \{\textsf{finalp}(step^r_m) \times tn, tn \times \textsf{startp}(step_{i+1})\}$ *with* tn *a null transition.*

As an example, use case "Submit order" (Figure 3) is extended by the extension use case "Register Customer" (Figure 4) according to the <<*extend*>> relation "Register Customer" × not(<*Customer*, *registered*>) × {*ep1*} × "Submit order". The corresponding P/T net in Figure 5 shows the inclusion of behaviors defined in "Register Customer" from place p_1 (the final place of step *1*). Transition c_1 corresponds to an *implicit guard* obtained from the <<extend>> condition negation (see Section 4.3). The extension behavior corresponds to places $p_5 - p_{19}$. Place p_{19}, the *final place* of step *1.5* in "Register Customer", is linked back to place p_4 the initial place of step *2* in "Submit order". Figure 6 shows an example where more than one of extension use case contribute at an extension point. The base use case *uc1* includes an extension point *ep1* that is referred to in two extension use cases (*ucex1* and *ucex2*). Suppose the <<*extend*>> relations $ucex1 \times c1 \times \{ep1\} \times uc1$ and $ucex2 \times c2 \times \{ep1\} \times uc1$. Figure 7 shows a P/T net corresponding to use case *uc1* with the extensions. Each combination of extension conditions is considered. In the situation where both *c1* and *c2* hold, the extensions in *ucex1* and *ucex2* occur concurrently.

Title: uc1 STEPS 1 .. 2 .. 3 .. EXT POINTS STEP 2. ep1 (a) base use case	Title: ucex1 PART1. At Ext ep1 1.1 .. 1.2 .. (b) extension use case ucex1	Title: ucex2 PART1. At Ext ep1 2.1 .. 2.2 .. (c) extension use case ucex2

Figure 6: Example for illustrate mapping of use case extension.

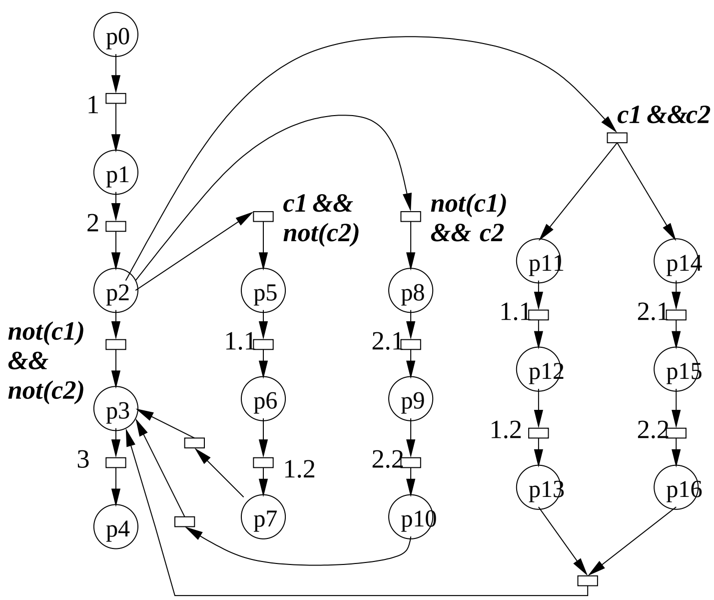


Figure 7: Petri net corresponding to use cases in Figure 6.

4.3.4 Use cases sequencing

The specification of use case sequencing is based on two complementary constructs: *enabling directives* and *follow lists*. An enabling directive specifies explicitly use cases which execution might start from the point of the directive. A *parallel enabling* directive variant allows specifying concurrent execution of the enabled use cases. In the *non-parallel enabling* directive variant, a *deferred choice* [29] is assumed among the enabled use cases and only one may execute. In accordance with Consistency Rule 2, the choice is determined from the next event.

Let isParallel be a function such that given a use case enabling directive *edir*, isParallel(*edir*) returns true if *edir* is a *parallel enabling* directive and false otherwise, let enabled_uc be a function such that enabled_uc(*edir*) is the set of use cases referred to in *edir*, let places_ucases be a relation such that p_i being a place and uc_o, uc_d being use cases $uc_o \times p_i \times uc_d \in$ places_ucases if p_i is a place corresponding to the enabling of use case uc_d by use case uc_o, and let $edir \times p_e \in$ enable_place (Cf. Mapping Rule 7).

Mapping Rule 12 *Given a use case enabling directive* edir *from use case* uc_o*,*

- *if* isParallel(edir)*,* $p_e \times tn \in F$*,* $tn \in T$ *and for each* $uc_i \in$ enabled_uc(edir)*,* $p_{uci} \in P, tn \times p_{uci} \in F, uc_o \times p_{uci} \times uc_i \in$ places_ucases *with* tn *a null transition,*
- *otherwise, for each* $uc_i \in$ enabled_uc(edir)*,* $t_{ni} \in T, p_{uci} \in P, p_e \times t_{ni} \in F$*,* $t_{ni} \times p_{uci} \in F, uc_o \times p_{uci} \times uc_i \in$ places_ucases *with each* t_{ni} *a null transition.*

A use case may include more than one enabling directive referring to the same use case. From the point of view of the enabled use case, any of the enabling directives may lead to execution. We define a relation unique_places_ucases to link enabling use cases, enabled use cases and a unique place from which a transition to the enabled use case start. Let enablePlaces be a function such that (uc_i, uc_j) being a pair of *main use cases*, enablePlaces$(uc_i,\ uc_j) = \{rel | rel = uc_i \times p_{uci} \times uc_j \in$ places_ucases$\}$.

Mapping Rule 13 *For each pair of* main use cases *(*uc_i*,* uc_j*), let* EnP *be =* enablePlaces*(*uc_i*,* uc_j*), if* $|\text{EnP}| \geq 1$*,* $p_u \in P$*, for each* $uc_i \times p_{uci} \times uc_j \in EnP$*,* $t_{ij} \in T, F \supset \{p_{uci} \times t_{ij}, t_{ij} \times p_u\}$ *with* t_{ij} *a null transition;* $uc_i \times p_u \times uc_j \in$ *unique_places_ucases.*

Follow lists specifies how enabled use cases may execute in reference to enabling use cases. In a *synchronized follow list*, all enabling use cases must have enabled the enabled use case. An *unsynchronized follow list* describes a situation where only one enabling use case is needed. Let isSynchronized and followed_ucases be functions such that given a use case $Uc =$ [*UCTitle*, *UCPrec*, *UCFoll*, *UCSt*, *UCAlt*, *UCPost*], isSynchronized(*UCFoll*) is true if *UCFoll* is a *synchronized follow list* and false otherwise, and followed_ucases(*UCFoll*) returns the list of use cases referred to in *UCFoll*.

Mapping Rule 14 *Given a* main use case Uc = *[*UCTitle*,* UCPrec*,* UCFoll*,* UCSt*,* UCAlt*,* UCPost*], let* FolUC *be* followed_ucases*(*UCFoll*),*

- *If* ***isSynchronized****(* UCFoll*), let* tn *be a null transition,* $tn \in T$*,* $tn \times$ ***initialp***(Uc) $\in F$*, for each* p_{uci} *such that* $\exists uc_i \in$ FolUC *with* $uc_i \times p_{uci} \times uc \in$ ***unique_places_ucases****,* $p_{uci} \times tn \in F$*.*

- *If* $\neg$ ***isSynchronized****(* UCFoll*), for each* p_{uci} *such that* $\exists uc_i \in$ FolUC *with* $uc_i \times p_{uci} \times uc \in$ ***unique_places_ucases****,* $F \supset \{p_{uci} \times t_{ni}, t_{ni} \times$ ***initialp***$(\mathrm{Uc}\}, t_{ni} \in T$ *with each* t_{ni} *a null transition.*

As an example, Figure 8 shows three sequentially related use cases: *uc1*, *uc2* and *uc3*, and Figure 9 shows a corresponding P/T net. Place p_2 corresponds to the enabling directive

Title: uc1 Follows: uc2 AND uc3 STEPS 1 .. 2 enable in parallel uc2, uc3 ALTERNATIVES 1.a c1 1.a.1 .. 1.a.2 enable uc2	Title: uc2 Follows: uc1 STEPS 1 .. 2 enable uc1, uc3 3 .. 4 ..	Title: uc3 Follows: uc1 OR uc2 STEPS 1 .. 2 .. 3 .. ALTERNATIVES 2.a c2 2.a.1 .. 2.a.2 enable uc1

Figure 8: Example illustrating mapping of use cases sequencing.

at step *2* of use case *uc1*. Since it is a *parallel* directive, it is mapped to flows $p_8 \times null$, $null \times p_9$, $null \times p_{10}$ denoting a forked transition. The P/T net includes p_{11} as a unique place corresponding to use case *uc2* enabling by use case *uc1*. Use case *uc2* enables *uc1* and *uc3* in a non-*parallel* way. Moreover, the enabling directive is followed by further steps. This translates to place p_{24} from which two transition sequences are forked to account for the parallel execution of steps *3*, *4* in *uc2* with the enabled use cases. Two alternative flows from p_{28} reflect the non-*parallel* nature of the directive. Use case *uc1 synchronized* follow list corresponds to a *join* transition to p_0 from enabling use cases *uc2*, *uc3*. In contrast, use case *uc3* non-*synchronized* follow list corresponds to having each enabling use case connect directly to p_{12}.

4.4 Properties of P/T nets

In this Section, we establish some properties of P/T nets obtained from our Mapping Rules. These properties are needed for state model derivation as discussed in the next Section. We consider use case behavior as *non-reentrant*. That is each execution of a use case is an instantiation of behavior completely independent from other instantiations. Notice that this assumption does not prevent several simultaneous executions of a same use case. However, each invocation starts a new *activity* with its own state information. We also consider all steps following an enabling directive as constituting a separate use case. As per Mapping

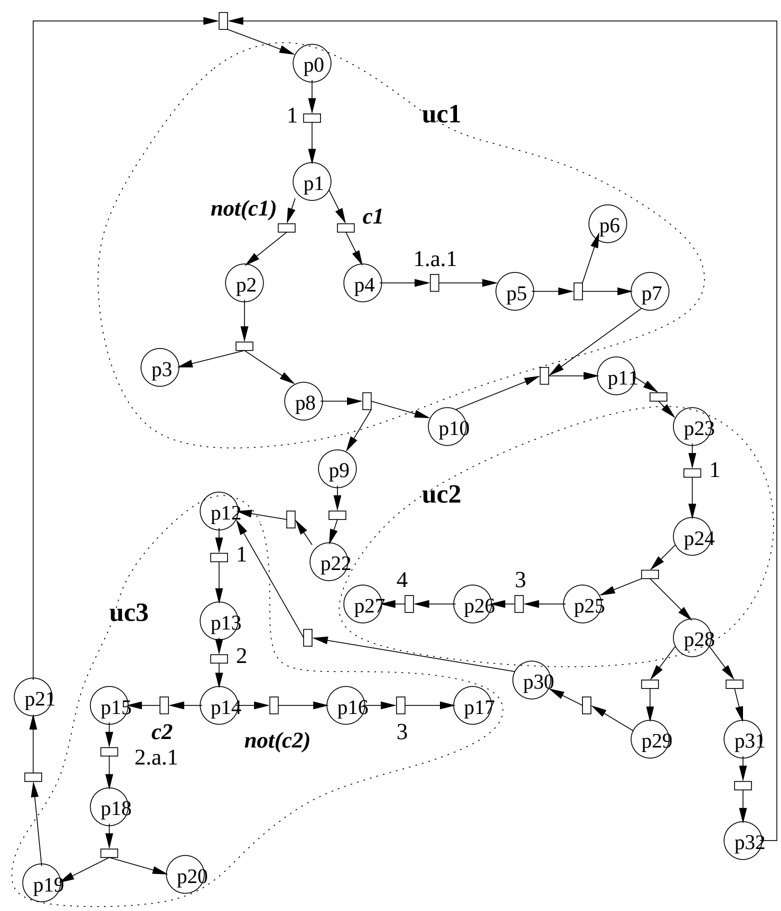


Figure 9: Petri net corresponding to use cases in Figure 8. The dotted lines delimits use case P/T nets.

Rule 7, steps after an enabling directive are concurrent with the enabled use cases. In order to adequately model such situations, a use case [*UCTitle*, *UCPrec*, *UCFoll*, *UCSt* $= step_0, \cdots, step_k, \cdots, step_n$, *UCAlt*, *UCPost*] with $step_k$ an enabled directive and $n > k$, would be considered as two use cases [*UCTitle*, *UCPrec*, *UCFoll*, *UCSt* $= step_0, \cdots, step_{k-1}$, *UCAlt*, *UCPost'*] and [*UCTitle'*, *UCPrec'*, *UCFoll'*, *UCSt'* $= step_{k+1}, \cdots, step_n$, *UCAlt*, *UCPost*]. The following theorem follows from these assumptions.

Lemma 1 *A use case P/T net obtained according to Mapping Rules 1 to 11 is* balanced*.*

Proof: A P/T net is balanced when all places/transitions sequences leaving a *forked* transition are subsequently *joined*. Only Mapping Rules 7 and 11 introduce *forked* transitions. One of the two forked transitions in rule 7 is not part of the use case P/T net as per the above assumptions. In the case of rule 11, balance is ensured as all the places/transitions sequences forked are disjointed and then merged before a transition links to a common output place. This situation is illustrated in Figure 7. □

Lemma 2 *A use case P/T net obtained according to Mapping Rules 1 to 11 is such that no parallel places/transitions sequences are connected.*

Proof: This property is a consequence of Consistency Rule 5, which mandates that there is no *branching* from unrelated steps sequences. Parallel places/transitions sequences result from only Mapping Rule 11. The only possibility for connection between parallel places/transitions sequences would necessitate a branching statement in one *part steps sequence* with a target in another *part steps sequence*. Consistency Rule 5 prevents this. □

Theorem 1 *A use case P/T net obtained according to Mapping Rules 1 to 11 is* 1-safe*.*

Proof: A P/T net *Pn* is considered *1-safe* if for all reachable marking *M*, each place in *Pn* contains at most one token. A use case P/T is a subnet restricted to a use case. According to Mapping Rule 1, there is one and only one *initial place* for a use case P/T net. Therefore, because of the *non-reentrant* assumption, only $\mathsf{initialp}(uc)$ is marked with a single token at the start of a use case execution (instantiation of a new use case P/T net). In order for a place p_k to contain more than one token, the use case P/T net needs to include a sequence of places/transitions $p_1 - t_1 \cdots - p_k$ such that (1) p_1 is an output place of more than one transitions and (2) each of these transitions is at the end of a sequence of places/transitions starting with a *fork*. This situation is impossible as use case P/T nets are balanced (Cf. Lemma 1) and parallel sequences are unconnected (Cf. Lemma 2). □

Lemma 3 *A use case P/T net [P, T, F] obtained according to Mapping Rules 1 to 11 is such that* $\forall p \in P$ *if* $\exists p \times tn \in F$ *with* tn *a* null transition*,* $\nexists t (tn \neq t)$ *such that* $p \times t \in F$.

Proof: Lemma 3 is equivalent to $(|\mathsf{nullTrans}| = 1) \Rightarrow (|\mathsf{systTrans}(p) \cup \mathsf{trigTrans}(p) \cup \mathsf{timeTrans}(p) \cup \mathsf{decTrans}(p)| = 0)$. This assertion is supported by the fact that transitions corresponding to the *null event* appear in a P/T net according to Mapping Rules 5, 6, 7 and 11. Because there is no alternative or extension point associated to *branching statements*, *inclusion directives* and *enabling directives*, in each of these cases, a single transition corresponding to the *null event* is created from the departing place and this transition is the only one from that place. $\square$

Lemma 4 *A use case P/T net [P, T, F] obtained according to Mapping Rules 1 to 11 does not include* ignored events*.*

Proof: *Trigger* and *timeout* events are ignored when in conflict with *reaction*, *decision* or *null* events. We already established that transitions corresponding to the *null event* do no conflict with any other transitions (Cf. Lemma 3). Because of Consistency Rule 9 and because Mapping Rule 4 only considers alternatives that are *not ignored* (Def. 10) from the step under consideration, the following can be asserted $\forall p \in P$, $(|\mathsf{timeTrans}(p)| \leq 1) \wedge$ $(|\mathsf{trigTrans}(p) \cup\ \mathsf{timeTrans}(p)| \geq 1) \Rightarrow (|\mathsf{systTrans}(p) \cup\ \mathsf{decTrans}(p)| = 0)$. $\square$

Theorem 2 *Given a use case* Uc *that satisfies Consistency Rule 9, the P/T net [P_{uc}, T_{uc}, F_{uc}] corresponding to* Uc *according to our Mapping Rules is* **not** non-deterministic *in the sense of Def. 7.*

Proof: None of the four criteria for *non-determinism* listed in Def. 7 is satisfied by P/T nets obtained according to the Mapping Rules.

- There is no place $p \in P_{uc}$ such that $|\mathsf{systTrans}(p) \cup \mathsf{nullTrans}(p)| > 1$ because $|\mathsf{systTrans}(p)| \leq 1$, and either $|\mathsf{nullTrans}| = 0$ or $(|\mathsf{nullTrans}| = 1$ and $|\mathsf{systTrans}(p) \cup \mathsf{trigTrans}(p) \cup \mathsf{timeTrans}(p) \cup \mathsf{decTrans}(p)| = 0)$.

  $|\mathsf{systTrans}(p)| \leq 1$ follows from the fact that a transition $tr \in \mathsf{systTrans}(p)$ if and only if step $step_i$ in Mapping Rule 3 is an operation step and p is place p_k. Other transitions from place p_k would correspond to step $step_{i-1}$ alternatives and extensions. All <<*extend*>> relations include a condition, and because of Consistency Rule 9, all alternatives which first step is a *system operation* have a non-**null** guard or delay. Therefore no other transition from p_k may correspond to a *system operation*.

  $|\mathsf{nullTrans}| = 0$ or $(|\mathsf{nullTrans}| = 1$ and $|\mathsf{systTrans}(p) \cup \mathsf{trigTrans}(p) \cup \mathsf{timeTrans}(p) \cup \mathsf{decTrans}(p)| = 0)$ follows from Lemma 3.

- For all places $p \in P_{uc}$, $|\mathsf{decTrans}(p)| > 0 \Rightarrow |\mathsf{systTrans}(p) \cup \mathsf{nullTrans}(p)| = 0$. We already established that when $|\mathsf{nullTrans}(p)| > 0$, $|\mathsf{decTrans}(p)| = 0$. We can also establish that when $|\mathsf{systTrans}(p)| > 0$, $|\mathsf{decTrans}(p)| = 0$ by observing again that a transition $tr \in \mathsf{systTrans}(p)$ if and only if step $step_i$ in Mapping Rule 3 is an operation step and p is place p_k. In order for $|\mathsf{decTrans}(p)| > 0$, $step_i$ needs to have alternatives or an extension point referred by extend relations. In all these cases, according to Def. 9, $step_i$ would include at least one *implicit guard* and therefore, tr would not start from p_k.

- Because of Consistency Rule 9, $\forall t_t \in \mathsf{trigTrans}(p)$, $\nexists t'_t \in \mathsf{trigTrans}(p)(t_t \neq t'_t)$ such that t_t and t'_t correspond to the same operation, and $\forall t_t \in \mathsf{timeTrans}(p)$, $\nexists t'_t \in \mathsf{timeTrans}(p)(t_t \neq t'_t)$ such that t_t and t'_t correspond to timeout events with the same delay. $\square$

5 State model synthesis

We developed an approach for state model synthesis from use cases based on the use case formalization discussed above. Given a use case model, we generate a two-level state model description of the behavior of the system under consideration. At the use case model level, the system's behavior is seen as an UML *activity diagram* [21] with main use cases as elements. Behaviors corresponding to each main use case is detailed using a hierarchical state machine (*StateChart*). We use the term *StateChart-Chart* to refer to the graphical depiction of the two-level state model description. As discussed in the introduction, one of our motivations is to provide a visual model of complex system behavior that can be analyzed and validated. At the use case model level, it should be possible to visualize control flow between use cases without delving into details. However, the specific points from which control flows from a use case in another use case should be identifiable when needed. We present a prototype tool allowing such capabilities in Section 5.3.

5.1 StateChart Generation

5.1.1 StateChart model

We generate a subset of UML StateChart that is formally defined as a tuple [$Trig_c$, $Reac_c$, G_c, S_c, Ch_c, Fk_c, Jn_c, F_c].

- $Trig_c$ is a set of *triggers* consisting of operations from the environment and timeouts.
- $Reac_c$ is a set of *reactions* that are operations executed by the system.
- G_c is a set of *guard* conditions.
- S_c is a set of states.
- Ch_c is a set of *choice pseudostates*.
- Fk_c is a set of *fork pseudostates*.
- Jn_c is a set of *join pseudostates*.
- F_c is an edge[3] function.

$V_c = S_c \cup Ch_c \cup Fk_c \cup Jn_c$ is the StateChart set of *vertices*. We distinguish *simple* states, *composite* states and *orthogonal* states. We assume boolean functions $\mathsf{issimple}$, $\mathsf{iscomposite}$ and $\mathsf{isorthogonal}$ such that given a state s, $\mathsf{issimple}(s)$ returns true if s is a *simple* state (false otherwise), $\mathsf{iscomposite}(s)$ returns true if s is a *composite* state (false otherwise) and $\mathsf{isorthogonal}(s)$ returns true if s is an *orthogonal* state (false otherwise).

[3]We use the term "edge" for StateCharts rather than "transition" to avoid confusion with Petri nets "transitions".

- Each composite state s_c is a tuple $[S_c, S0_c, Ch_c, Fhk_c, Jhn_c]$ with S_c a set of states, $S0_c \in (S_c \cup Ch_c)$ a composite state initial vertex, Ch_c a set of choice pseudostates, Fhk_c a set of fork pseudostates and Jhn_c a set of join pseudostates. We assume functions $\mathsf{sub\text{-}vertices}$ and $\mathsf{initial\_vertex}$ defined as follow. $\mathsf{sub\text{-}vertices}(s_c)$ is a set of s_c *sub-vertices*. For each vertex $s_j \in \mathsf{sub\text{-}vertices}(s_c)$, s_c is a *sup-vertex* of s_j. $\mathsf{initial\_vertex}(s_c)$ returns the *initial sub-vertex* of s_c.

- For each $s_i \in S_c$ such that $\mathsf{isorthogonal}(s_i)$, we assume function $\mathsf{regions}$ such that $\mathsf{regions}(s_i)$ is a set of parallel *regions*. Each region r is a *composite* state.

Each state is associated with a *timer* to count elapsed time while the system is waiting in that state. We assume a function timer such that given a state s, $\mathsf{timer}(s)$ returns the timer associated to s. Implicitly $\mathsf{timer}(s)$ is *started* whenever an edge produces a state change to s, and $\mathsf{timer}(s)$ is implicitly *stopped* whenever an edge exits from state s. A *timeout* event *timeout(d)* occurs when a timer has been started and not stopped before delay d elapsed.

Each edge is a relationship $v_s \times guards \times trig \times reacs \times v_t$ with $v_s \in V_c$ a *source vertex*, $v_t \in V_c$ a *target vertex*, $reacs \subset guards$ a set of guard conditions, $trig \subset Trig_c$ a set of triggers and $reacs \subset Reac_c$ a set of reactions. The compound notation $v_s \overset{[guards]trig/reacs}{\longrightarrow} v_t$ is used for edges. The UML specification defines different constraints on edges [21]. Edges with *choice pseudostates* as sources must have *trig* empty. *Fork pseudostates* are used to split an edge into several edges. The target of each of these edges must be a state in a different region of an orthogonal state. *Join pseudostates* are used to merge several edges from different regions of an orthogonal state into a single edge. An edge to a composite state s_c is equivalent to an edge ending in the initial vertex of s_c ($\mathsf{initial\_vertex}(s_c)$). An edge to an orthogonal state s_o is equivalent to an edge ending in the initial vertices of each of s_o regions. An edge starting from a composite or orthogonal state s on a trigger t is equivalent to a set of edges from each substates of s with no outgoing edge on t.

5.1.2 StateChart generation approach

Figures 10 - 12 show an algorithm for StateChart generation from a use case, and Figure 13 shows a StateChart generated from use case "Submit order" described in Figure 3. The StateChart generation algorithm relies on the properties established in Section 4.4. It has been shown in [5] that P/T nets which are *balanced*, *1-safe* and without connection between parallel places/transitions sequences, have structure-preserving equivalent StateCharts as a one-to-one correspondence between P/T nets places and StateChart states can be made. We also established that use case P/T nets do not include ignored events (Cf. Lemma 3) and are not non-deterministic (Cf. Theorem 2). These properties allow generation of stateCharts edges in $v_s \overset{[guards]trig/reacs}{\longrightarrow} v_t$ form.

StateChart generation starts from a P/T net equivalent of a use case obtained from the application of Mapping Rules 1 to 11. Procedure $\mathsf{GeneratePTNet}$ a formulation of these Mapping Rules, returns a P/T net from a use case. We assume relation $\mathsf{PlaceVertices}$ such that p being a place and v a vertex, $p \times v \in \mathsf{PlaceVertices}$ if vertex v corresponds to place

GenerateStateChart(uc: Use Case): StateChart

A1. Pn_{uc} = GeneratePTNet(uc)

A2. Let Sc_{uc} =[$Trig_c = \emptyset$, $Reac_c = \emptyset$, $G_c = \emptyset$, $S_c = \emptyset$, $Ch_c = \emptyset$, $Fk_c = \emptyset$, $Jn_c = \emptyset$, $F_c = \emptyset$] be a StateChart

A3. Let root be a *composite* state, $S_c = S_c \cup \{$root$\}$ and sst be a *simple* state *sub-vertex* of root, initial_vertex(root) = sst, PlacesVertices = PlacesVertices $\cup$ {initialp(Uc) $\times$ $sst\}$

A4. GenerateSCFromPLace(initialp(Uc), root, sst, Pn_{uc},Sc_{uc})

A5. For each *global alternative galt* = [*Acond*, *Adelay*, *Asteps* = [$astep_0$, $\cdots$, $astep_m$], *Apost*] of *uc*

A5.1. Let $Pn_{alt} = [P_{alt}, T_{alt}, F_{alt}]$ be a P/T net corresponding to *galt*

A5.2. GenerateSCFromPLace(startp($astep_0$), root, root, Pn_{alt},Sc_{uc})

A6. return Sc_{uc}

Figure 10: StateChart generation algorithm (Part 1).

p. In procedure GenerateSCFromPlace, we assume a relation TransJoins such that t being a transition and j a join pseudostate, $t \times j \in$ TransJoins if j corresponds to t. Each use case corresponds to a *root* composite state that is supervertex of all other vertices. We consider *global alternatives* separately from the other alternatives and exploit StateCharts composite states properties. As an example, use case "Submit order" global alternative results in a transition from the root state *s0* to *s12*. This transition applies to all of *s0* sub-vertices. Procedure GenerateSCFromPlace considers a place and generates an edge in the format $v_s \stackrel{[guards]trig/reacs}{\longrightarrow} v_t$ by looking up *guards*, *triggers* and *reactions* from the place. Notice that use case P/T net properties are such that either all or none of the transitions from a place are decisions. Places from which all transitions are decisions correspond to choice pseudostates, while the other places correspond to simple states (C5). Concurrent sequences of places/transitions are mapped to *orthogonal states* (B2.1. and B2.2). Forking transitions correspond to fork pseudosates and join transitions to join pseudostates. As an example, Figure 14 shows a StateChart corresponding to use case *uc1* shown in Figure 6. We assume steps *1*, *1.2*, *2.2* are actor operations and steps *2*, *1.1*, *2.1*, *3* are system reactions. Use case *uc1* P/T net is shown in Figure 7.

5.2 Sequential integration of Use Cases

We use a variant of UML activity diagrams [21] called *StateChart-Charts*, to integrate sequentially related *main* use cases. A StateChart is not appropriate for use case integration as the properties discussed in Section 4.4 can not extend to use case model P/T nets without

GenerateSCFromPlace(p: Place, supv: Vertex, v: Vertex, Pn $= [P_{uc}, T_{uc}, F_{uc}]$**: P/T net,** $Sc_{uc} = [Trig_c, Time_c, Reac_c, G_c, S_c, Ch_c, Fk_c, Jn_c, F_c]$**: StateChart)**

B1. Mark p as visited

B2. For each transition t such that $p \times t \in F_{uc}$,

B2.1. If $|t\bullet| > 1$, let *so* be an *orthogonal* state sub-vertex of *supv*, let *sf* be a *fork pseudostate* sub-vertex of *supv*, let *cond* be the condition corresponding to t, $F_c = F_c \cup \{v \times \{cond\} \times \emptyset \times \emptyset \times sf\}$

B2.1.1. For each $t \times p_o \in F_{uc}$, Let r be a *region* in *so* and state so_i the initial state of r, $F_c = F_c \cup \{sf \times \emptyset \times \emptyset \times \emptyset \times so_i\}$, GenerateSCFromPlace($p_o$, r, so_i, Pn, Sc_{uc})

B2.2. If $|\bullet t| > 1$,

B2.2.1. If $\nexists t \times sj \in$ TransJoins, let *sj* be a *join pseudostate* sub-vertex of *supv*,

- add $t \times sj$ to TransJoins
- let $t \times p_j \in F_{uc}$, $s_j =$ GatherReactions(p_j, Reacs, supv, Pn, Sc_{uc})
- $F_c = F_c \cup \{sj \times \emptyset \times \emptyset \times Reacs \times s_j\}$

B2.2.2. Let $t \times sj \in$ TransJoins, $F_c = F_c \cup \{v \times \emptyset \times \emptyset \times \emptyset \times sj\}$

B2.3. If $|\bullet t| = |t\bullet| = 1$

B2.3.1. If t corresponds to *trig*, an *actor operation* or a *timeout*, let $t \times p_j \in F_{uc}$

- $nv =$ GatherReactions(p_j, Reacs, supv, Pn, Sc_{uc}), $F_c = F_c \cup \{v \times \emptyset \times \{trig\} \times Reacs \times nv\}$

B2.3.2. If t corresponds to *cond*, a condition let $t \times p_j \in F_{uc}$

- $nv =$ GatherReactions(p_j, Reacs, supv, Pn, Sc_{uc}), $F_c = F_c \cup \{v \times \{cond\} \times \emptyset \times Reacs \times nv\}$

B2.3.3. If t corresponds to *reac*, a *system operation*, let $t \times p_j \in F_{uc}$

- $nv =$ GatherReactions(p_j, Reacs, supv, Pn, Sc_{uc}),
- $Reacs = \{reac\} \cup Reacs$, $F_c = F_c \cup \{v \times \emptyset \times \emptyset \times Reacs \times nv\}$

Figure 11: StateChart generation algorithm (Part 2).

GatherReactions(p: Place,Reacs: Set, supv: Vertex, Pn = [P_{uc}, T_{uc}, F_{uc}]: P/T net, Sc_{uc} = [$Trig_c$, $Time_c$, $Reac_c$, G_c, S_c, Ch_c, Fk_c, Jn_c, F_c]: StateChart): Vertex

C.1. Let t be such that $p \times t \in F$

C.2. If t corresponds to the *null event*, let $t \times p_j \in F_{uc}$, $p = p_j$, GOTO C.1.

C.3. If t corresponds to *reac*, a *system operation*, $Reacs = Reacs \cup \{reac\}$, let $t \times p_j \in F_{uc}$, $p = p_j$, GOTO C.1.

C.4. If p is marked as visited, let $p \times sv \in$ PlacesVertices, return sv

C.5. Let sst be:

- a *choice* pseudostate *sub-vertex* of supv, if all transitions from initialp are *decisions*, or
- a *simple* state *sub-vertex* of supv, otherwise

PlacesVertices = PlacesVertices $\cup \{p \times sst\}$

C.6. GenerateSCFromPLace(p, supv, sst, Pn_{uc},Sc_{uc})

C.7. return sst

Figure 12: StateChart generation algorithm (Part 3).

imposing unpractical constraints on use case sequencing. For instance, P/T nets *balanceness* and hence *1-safety* would require every set of concurrent use cases to be subsequently synchronized. The "Online Broker System" example as presented in Section 3, would be in violation of such constraint. UML activity diagrams allows modeling of use case sequencing constraints with the formal interpretation provided by Mapping Rules 12 to 14.

5.2.1 StateChart-Charts

A StateChart-Chart is a subset of an activity diagram where *action nodes* are StateCharts. Formally a StateChart-Chart is a tuple [*SChs*, *CNds*, *CFls*] with: *SChs* a set of StateCharts, *CNds* a set of Flow Nodes, and *CFls* a StateChart-Chart flow relation. Elements in *SChs* are StateCharts generated from use cases. Flow Nodes include activity diagram *initial* nodes, *decision* nodes, *join* nodes, *fork* nodes and *merge* nodes [21]. The StateChart-Chart flow relation *CFls* is a relation defined in domain ($SEn_c \cup CNds$) and range ($SRoot_c \cup CNds$) with SEn_c the union of all the StateCharts in *SChs use case enabling* vertices, and $SRoot_c$ the union of all the StateCharts in *SChs root* nodes. A use case enabling vertex corresponds to a use case enabling directive in a StateChart. More precisely, given a step sequence $step_0, \cdots, step_k, step_l$ with $step_l$ a use case enabling directive, the *target vertex* of the edge corresponding to step $step_k$ in *Schar* is a use case enabling vertex for $step_l$. An element $nd_s \times nd_a \in CFls$ represents *control flow* from nd_s to nd_a. Control flows between nodes in

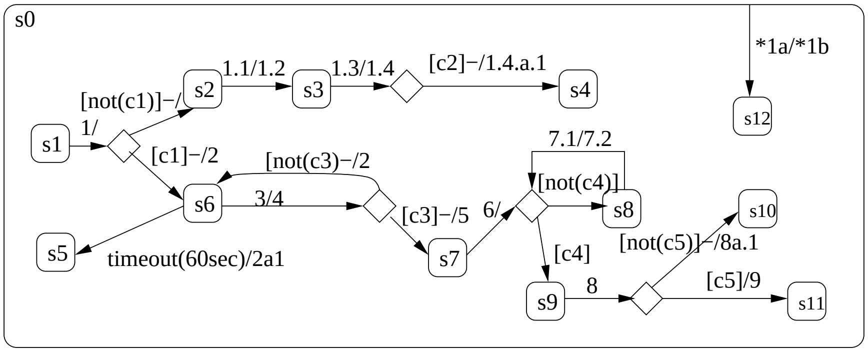


Figure 13: StateChart generated from use case "Submit order". The event labels are the same as in Figure 5.

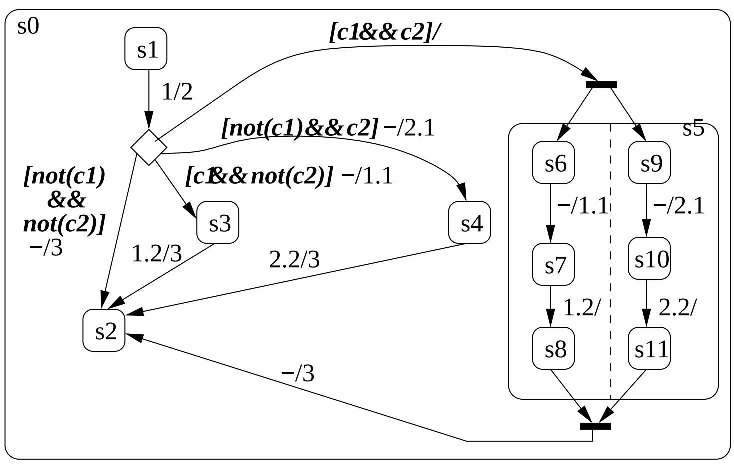


Figure 14: StateChart corresponding to use case *uc1* shown in Figure 6.

CNds obey to UML activity diagram semantics. A flow from an enabling vertex captures the enabling of use cases by the use case from which the flow originates, while a flow ending at a use case root node signifies that the target use case is enabled and may start its execution. We consider that a new instance of activity (with a corresponding StateChart) is started for each execution of a use case.

Mappings between use cases sequencing constructs, P/T nets and StateChart-Charts are described in Figure 15. There is a straightforward mapping between use case sequencing concepts and UML activity diagram as we formally defined these sequencing concepts in term of P/T nets on one hand, and the semantics of UML activity diagrams are expressed in term of the Petri nets token game [21] on the other hand. States labelled *Se* in the StateChart-Chart fragments correspond to use case enabling directives. An enabling directive that refers to a single use case (with its matching *follow list*) corresponds to a simple flow from one use case to the other (Figure 15-a). Situations where a *follow list* refers to more than one use case correspond to a join when operator AND is used (Figure 15-b), or a merge when operator OR is used (Figure 15-c). Figures 15-d shows that a non-parallel use case enabling directive corresponds to a decision modeling a deferred choice based on the next event occurrence, while 15-e shows that a parallel use case enabling directive corresponds to a fork. Finally in a situation where there are steps after an enabling directive, Figure 15-f shows that in the corresponding StateChart-Chart, the steps after the directive are separated in a composite state. A fork node is used to model the concurrent execution of these steps with the enabled use cases.

5.2.2 Use Cases integration algorithm

Figure 16 shows an algorithm for StateChart-Charts generation. We assume the following functions and relations. Given a use case model $\mathcal{M}$, $\mathsf{stateCharts}(\mathcal{M})$ is a set of StateCharts obtained from the *main* use cases in $\mathcal{M}$ according to algorithm GenerateStateChart. Given a use case uc, $\mathsf{stateChart}(uc)$ is a StateChart corresponding to uc. Function enable_vertex is such that $\mathsf{enable\_vertex}(edir, Schar)$ is a use case enabling vertex corresponding to $edir$ in a StateChart *Schar*. Given the set of all use case enabling vertices *Ests*, relation nodes_ucases is such that $uc_o \times n_i \times uc_d \in \mathsf{nodes\_ucases}$ if $n_i \in (Ests \cup CNds)$ is a node corresponding to the enabling of use case uc_d by use case uc_o. Function enableNodes is such that given (uc_i, uc_j) a pair of use cases, $\mathsf{enableNodes}(uc_i,\ uc_j) = \{rel | rel = uc_i \times n_{uci} \times uc_j \in \mathsf{nodes\_ucases}\}$. Function unique_nodes_ucases is similar to nodes_ucases except that every pair of use case corresponds to at most one node.

The StateChart-Charts algorithm is a formulation of Mapping Rules 12 to 14 based on the relation between P/T nets and StateChart-Charts depicted in Figure 15. Step *2.1* deals with the situation where an enabling directive is followed by further steps (Figure 15-f). Steps *2.2*, *2.3* and *2.4* correspond to Mapping Rule 12. The parallel form of enabling directives are dealt with in step *2.2* (Figure 15-e), the non-parallel form in step *2.3* (Figure 15-d) and the situation where only one use case is enabled in step *2.4* (Figure 15-a). Step *3* of the algorithm corresponds to Mapping Rule 13, and consists of merging all enabling flows from an enabling use case targeted toward a use case. Mapping Rule 14 corresponds to step *4*.

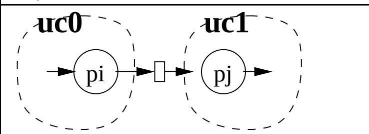
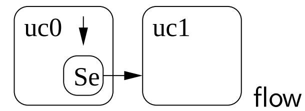
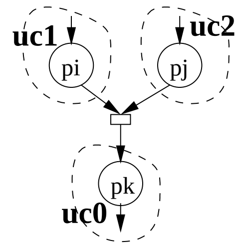
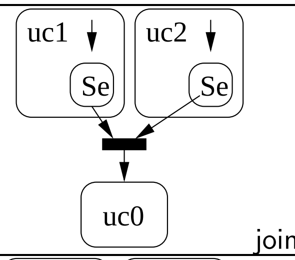
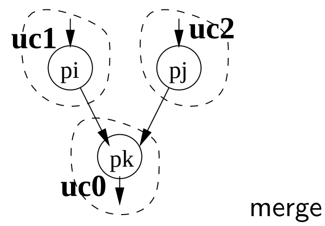
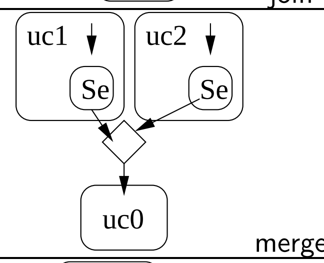
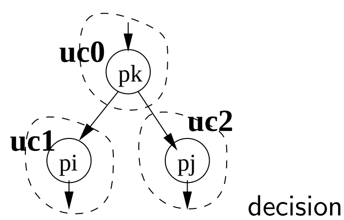
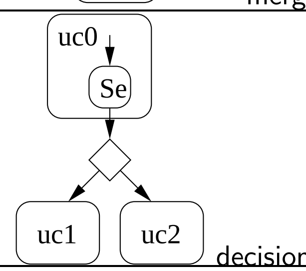
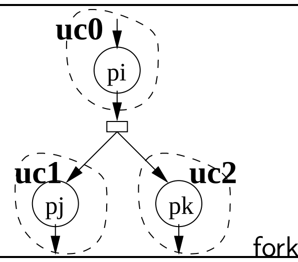
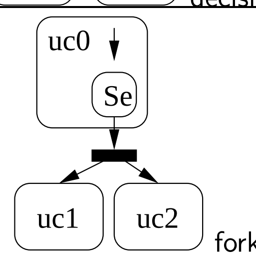
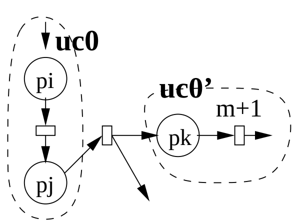
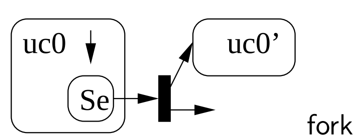
Use Case Construct	P/T net fragment	StateChart-Chart fragment
Title: uc0 1. ... ... n. enable: uc1 Title: uc1 ... Follows: uc0 (a)		
Title: uc0 ... Follows: uc1 AND uc2 (b)		
Title: uc0 ... Follows: uc1 OR uc2 (c)		
Title: uc0 1. ... ... n. enable: uc1, uc2 (d)		
Title: uc0 1. ... ... n. enable in parallel: uc1, uc2 (e)		
Title: uc0 1. ... ... m. enable ... m+1... ... n (f)		

Figure 15: Mappings between use case sequencing constructs, P/T nets and StateChart-Charts.

GenerateStateChartChart($\mathcal{M}$ = [*Act*, *Uc*, *Rel*, *InitialUc*]: Use Case Model): StateChart-Chart

1. Let Scc = [SChs = $\mathsf{stateCharts}(\mathcal{M})$, $CNds = \emptyset$, $CFls = \emptyset$] be a StateChart-Chart.

2. For each $uc \in$M, let Sch_{uc} be $\mathsf{stateChart}(uc)$.
   For each $edir \in \mathsf{enable\_dirs}(uc)$, let s_e be $\mathsf{enable\_vertex}(edir, Sch_{uc})$

   2.1. If there are steps after *edir*, let Sch_n be the StateChart corresponding to the steps after *edir*, $CNds = CNds \cup \{En\}$ with En a *fork* node,

   - $CFls = CFls \cup \{s_e \times En, En \times \mathsf{root}(Sch_n)\}$, let s_{en} = En

   Else let $s_{en} = s_e$

   2.2. If $\mathsf{isParallel}(edir)$, $CNds = CNds \cup \{Nd_e\}$ with Nd_e a *fork* node,

   - $CFls = CFls \cup \{s_{en} \times Nd_e\}$, $\forall uc_i \in \mathsf{enabled\_uc}(edir), \mathsf{nodes\_ucases} = \mathsf{nodes\_ucases} \cup \{uc \times Nd_e \times uc_i\}$

   2.3. If $\neg$ $\mathsf{isParallel}$ and $|\mathsf{enabled\_uc}(edir)| > 1$ $CNds = CNds \cup \{Nd_e\}$ with Nd_e a *decision* node,

   - $CFls = CFls \cup \{s_{en} \times Nd_e\}$, $\forall uc_i \in \mathsf{enabled\_uc}(edir), \mathsf{nodes\_ucases} = \mathsf{nodes\_ucases} \cup \{uc \times Nd_e \times uc_i\}$

   2.4. If $|\mathsf{enabled\_uc}(edir)| = 1$, let $\mathsf{enabled\_uc}(edir) = \{uc_i\}$, $\mathsf{nodes\_ucases} = \mathsf{nodes\_ucases} \cup \{uc \times Nd_e \times uc_i\}$

3. For each pair of *main use cases* (uc_i, uc_j), let *EnN* be = $\mathsf{enableNodes}(uc_i, uc_j)$, if $|EnN| \geq 1$, let $Nm_u \in CNds$ be a *merge* node, for each $uc_i \times n_{uci} \times uc_j \in EnN$, $CFls = CFls \cup \{n_{uci} \times Nm_u\}$ $uc_i \times Nm_u \times uc_j \in \mathsf{unique\_nodes\_ucases}$.

4. For each *main* use case uc = [*UCTitle*, *UCPrec*, *UCFoll*, *UCSt*, *UCAlt*, *UCPost*], let *FolUC* be $\mathsf{followed\_ucases}(UCFoll)$,

   4.1. If $\mathsf{isSynchronized}(UCFoll)$, let $Nf \in CNds$ be a *join* node, $CFls = CFls \cup \{Nf \times \mathsf{root}(uc)\}$.

   4.2. If $\neg$ $\mathsf{isSynchronized}(UCFoll)$ and $|FolUC| > 1$, let $Nf \in CNds$ be a *merge* node, $CFls = CFls \cup \{Nf \times \mathsf{root}(uc)\}$.

   4.3. If $|FolUC| = 1$ let *Nf* = root(uc).

   For each n_{uci} such that $\exists uc_i \in FolUC$ with $uc_i \times n_{uci} \times uc \in \mathsf{unique\_places\_ucases}$, $CFls = CFls \cup \{n_{uci} \times Nf\}$

5. $CNds = CNds \cup \{In\}$ with *In* an *initial* node. For each $uc_i \in InitialUc$, $CFls = CFls \cup \{In \times \mathsf{root}(\mathsf{stateChart}(uc_i))\}$

6. return Scc

Figure 16: StateChart-Chart generation algorithm.

Step *4.2* handles synchronized follow lists (Figure 15-b), step *4.1* handles unsynchronized follow lists (Figure 15-c) and step *4.3* handles situations where only one use case is referred to in a follow list.

Figure 17 shows a StateChart-Chart corresponding to the use cases in Figure 8. A

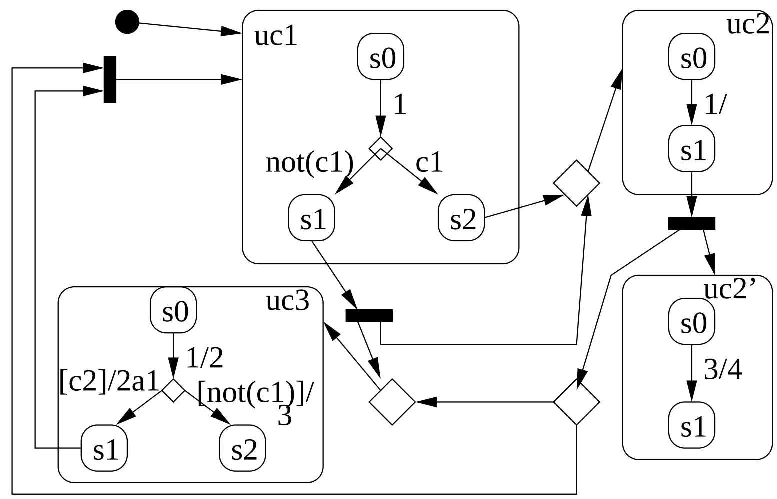


Figure 17: StateChart-Chart corresponding to use cases in Figure 8.

corresponding P/T net is presented in Figure 9. The example illustrates a situation where a use case is split by an enabling statement, with use case *uc2* corresponding to composite states *uc2* and *uc2'*. We assume use case *uc1* is the only initial use case among the three use cases.

5.3 Implementation and initial evaluation

The StateChart-Chart synthesis algorithm is implemented in a tool called Use Case Editor (UCEd) [1]. The tool accepts use cases in the concrete syntax outlined in Section 3.2, checks for the consistency rules enumerated in Sections 3.2 and 4, and generates StateChart-Charts from consistent use cases. UCEd provides a visualization mechanism for generated StateChart-Charts at two levels: the use case model level with StateCharts details hidden and use case StateCharts level. Figure 18 shows a StateChart-Chart generated from the “Online Broker System” example at the use case model level. Detailed StateChart for each use case can be seen by expanding use case nodes. Generated StateChart-Charts can be animated as prototypes. UCEd includes a Simulator that provides a graphical user interface that allows “playing” generated StateChart-Charts. This gives an opportunity to validate use cases and their sequencing constraints.

We used UCEd to carry different case studies aimed at validating our approach. The tool is being used to teach use case modeling in an academic setting. Moreover, UCEd is released as an open source project for further feedback. The projects we experimented

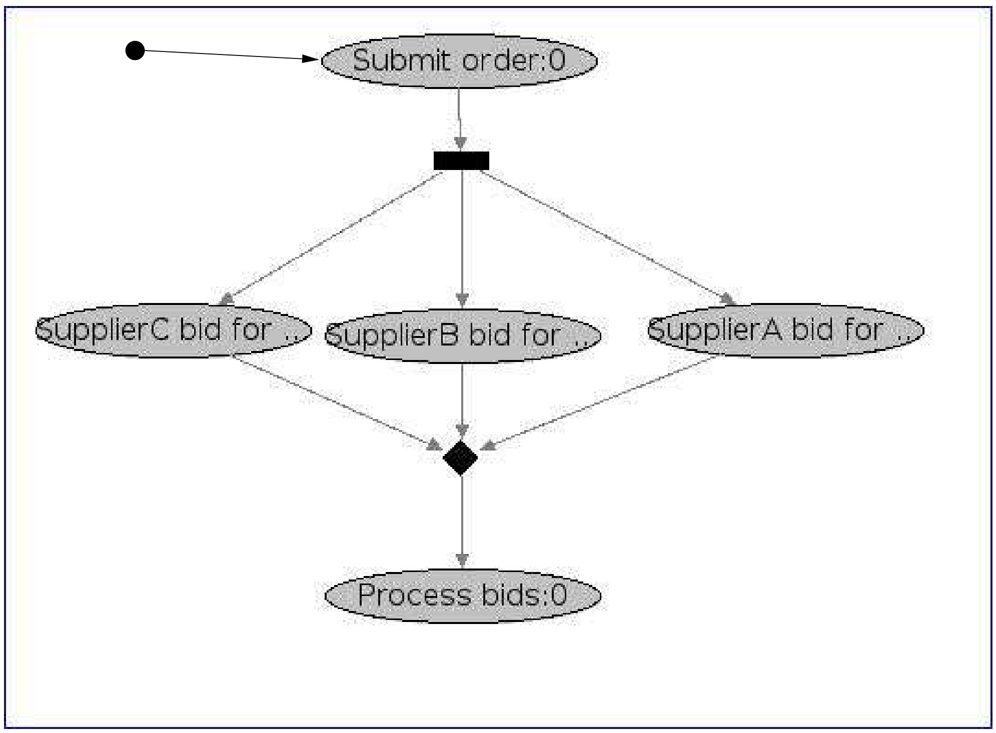


Figure 18: Use case model view of a StateChart-Chart generated from the “Online Broker System”.

with involve up to 20 use cases of varying degree of complexity. Because use cases are requirements artifacts, the number of use cases in project is typically limited. In any case, scaling to much larger projects should not be an issue because of the complexity of the state model construction algorithm. It is note-worthy that the complexity of StateChart and StateChart-Chart generation algorithms are both polynomial. Only Mapping Rule 11 induces an exponential complexity in term of the number of extension parts contributing to an extension point. However, this number is generally very limited in realistic examples.

6 Conclusions

This paper has proposed a formalization of textual use cases. We started from a formal definition of use case syntax; a UML metamodel as abstract syntax and a restricted natural language as concrete syntax. The main contribution of the paper is a definition of formal control-flow based semantics for use cases. We chose to express these semantics using the Basic Petri nets formalism. The choice of Basic Petri nets is sufficient for the assumed event model. Different assumptions or the consideration of control flow issues may ask for other forms of Petri nets such as Timed [30] or Coloreds Petri nets [12].

An equivalent P/T net may be constructed from a use case model granted that Consistency Rules specified in this paper are satisfied. These Consistency Rules should therefore be considered as part of a guideline for authoring use cases. We also developed algorithms for the synthesis of a two-level state model from a set of sequentially related use cases. These algorithms are implemented in a prototype tool and serve to validate the formal semantics.

We only consider UML use case << *include* >> and << *extend* >> relations in our formalization. We do not consider use cases *generalization* mainly because of uncertainties about the impact of this relation on textual use cases. We expect it would be possible to support use cases *generalization* in the future by extending use cases to Petri nets Mapping Rules. Our future works also includes system acceptance test generation from use cases based on the formal semantics.

A Description of the Online Broker System use cases

Title: SupplierA Bid
System Under Design: Broker System
`Follows: Submit order`
Precondition: An Order has been broadcasted
Follows Use Cases: Submit order
Success Postcondition: SupplierA has submitted a bid
STEPS

1. SupplierA receives the Order and examines it
2. SupplierA submits a Bid for the Order
3. The Broker System sends the Bid to the Customer
4. enable use case Process bids

ALTERNATIVES

- 1.a. SupplierA can not satisfy the Order
    - 1.a.1. SupplierA passes on the Order

Figure 19: Description of use case “SupplierA Bid” in the Online Broker System.

Title: SupplierB Bid
System Under Design: Broker System
`Follows: Submit order`
Precondition: An Order has been broadcasted
Follows Use Cases: Submit order
Success Postcondition: SupplierB has submitted a bid
STEPS

1. SupplierB receives the Order and examines it
2. SupplierB submits a Bid for the Order
3. The Broker System sends the Bid to the Customer
4. enable use case Process bids

ALTERNATIVES

1.a. SupplierB can not satisfy the Order

1.a.1. SupplierB passes on the Order

Figure 20: Description of use case “SupplierB” in the Online Broker System.

Title: SupplierC Bid
System Under Design: Broker System
`Follows: Submit order`
Precondition: An Order has been broadcasted
Follows Use Cases: Submit order
Success Postcondition: SupplierC has submitted a bid
STEPS

1. SupplierC receives the Order and examines it
2. SupplierC submits a Bid for the Order
3. The Broker System sends the Bid to the Customer
4. enable use case Process bids

ALTERNATIVES

1.a. SupplierC can not satisfy the Order

1.a.1. SupplierC passes on the Order

Figure 21: Description of use case “SupplierC Bid” in the Online Broker System.

Title: Process Bids
System Under Design: Broker System
Follows Use Cases: `SupplierA Bid` OR `SupplierB Bid` OR `SupplierC Bid`
Precondition: SupplierA has bidded or SupplierB has
bidded or SupplierC has bidded
STEPS

1. Customer examines the bid
2. Customer signals the system to proceed with bid
3. include Handle Payment
4. System put an order with the selected bidder

Figure 22: Description of use case "Process bids" in the Online Broker System.

Title: Handle Payment
System Under Design: Broker System STEPS

1. The Broker System asks the Customer for Credit Card information
2. The Customer provides her Credit Card information
3. The Broker System asks a Payment System to process the Customer's Payment
4. The Broker System displays an acknowledgement message to the Customer

ALTERNATIVES

3.a. The Customer Payment is denied

3.a.1. The Broker System displays a payment denied page

Figure 23: Description of use case "Process bids" in the Online Broker System.

References

[1] Use Case Editor (UCEd) toolset.
http://www.site.uottawa.ca/~ssome/Use_Case_Editor_UCEd.html.

[2] C. Ben Achour and C. Souveyet. Bridging the gap between users and requirements engineering the scenario-based approach. Technical Report 99-07, Cooperative Requirements Engineering With Scenarios (CREWS), 1999.

[3] A. Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2001.

[4] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001.

[5] Rik Eshuis. Statecharting petri nets. Beta Working Paper WP-143, Eindhoven University of Technology, 2005.

[6] Rik Eshuis and Juliane Dehnert. Reactive Petri nets for Workflow Modeling. In Springer-Verlag, editor, *Application and Theory of Petri Nets 2003, volume 2679 of Lecture Notes in Computer Science*, pages 295–314, 2003.

[7] M. Fowler. Use and Abuse Cases. www.DistributedComputing.com, 1998.

[8] M. Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In *Software Engineering - ESEC'95. Proceedings of the 5th European Software Engineering Conference*, pages 254–271. Springler LNCS 989, 1995.

[9] ITU-T. Message Sequence Charts (MSC'2000). Recommendation Z.120, 2000.

[10] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1998.

[11] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering,* A Use Case Driven Approach. Addison-Wesley, ACM Press, 2 edition, 1993.

[12] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[13] Ismail Khriss, Mohammed Elkoutbi, and Rudolf K. Keller. Automating the synthesis of UML statechart diagrams from multiple collaboration diagrams. In *UML*, pages 132–147, 1998.

[14] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In *DIPES*, pages 61–72, 1998.

[15] Stefan Leue, L. Mehrmann, and Mohammad Rezai. Synthesizing software architecture descriptions from message sequence chart specifications. In *ASE*, pages 192–195, 1998.

[16] Xiaoshan Li, Zhiming Liu, and Jifeng He. Formal and use-case driven requirement analysis in UML. Research Report 230, UNU/IIST, Macau, March 2001.

[17] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A Comparative Survey of Scenario-based to State-based Model Synthesis Approaches. In *SCESM '06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 5–12. ACM Press, May 2006.

[18] Dong Liu, Kalaivani Subramaniam, Armin Eberlein, and Behrouz H. Far. Natural Language Requirements Analysis and Class Model Generation Using UCDA. In *IEA/AIE 2004: 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 295–304. Springer, May 2004.

[19] Vladimir Mencl. D eriving Behavior Specification From Textual Use Cases. In *Proceedings of Workshop on Intelligent Technologies for Software Engineering (WITSE04)*, pages 331–341, September 2004.

[20] S. Nayanamana and S. Somé. Generating a Domain Model from a Use Case Model. In *14th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2005)*, july 2005.

[21] OMG. UML 2.0 Superstructure, 2003. Object Management Group.

[22] James L. Peterson. *Theory and the Modeling of Systems*. Prentice-Hall, N.J., 1981.

[23] Shane Sendall and Alfred Strohmeier. *From Use Cases to System Operation Specifications*, volume 1939 of *Lecture Notes inComputer Science*, pages 1–15. Springer, 2000.

[24] Wuwei Shen, Mohsen Guizani, Zijiang Yang, Kevin J. Compton, and James Huggins. Execution of A Requirement Model in Software Development. In *Proceedings of the ISCA 13th International Conference on Intelligent and Adaptive Systems and Software Engineering*, pages 203–208. ISCA, July 2004.

[25] S. Somé. Supporting Use Cases based Requirements Engineering. *Information and Software Technology*, 48(1):43–58, 2006.

[26] S. Somé, R. Dssouli, and J. Vaucher. From Scenarios to Timed Automata: Building Specifications from Users Requirements. In *Proceedings of the 2nd Asia Pacific Software Engineering Conference (APSEC'95)*. IEEE, dec 1995.

[27] Stéphane S. Somé. Specifying Use Case Sequencing Constraints Using Description Elements. In *Sixth International Workshop on Scenarios and State Machines (SCESM'07: ICSE Workshops)*, 2007.

[28] S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavorial Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2), February 2003.

[29] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[30] Jiacun Wang. *Timed Petri Nets, Theory and Application*. Kluwer Academic Publishers, 1998.

[31] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, jun 2000.

[32] Jon Whittle and Praveen K. Jayaraman. Generating hierarchical state machines from use case charts. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, 2006.

[33] T. Ziadi, L. Helouet, and J. M. Jezequel. Revisiting statechart synthesis with an algebraic approach. In *Proceedings of the 26th IEEE International Conference on Software Engineering (ICSE 2004)*, pages 242–251, 2004.