N° d'ordre : 04/2011-D/IN

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumedienne

Faculté d'Electronique & Informatique

THESE

Présentée pour l'obtention du grade de DOCTORAT

EN : INFORMATIQUE

Spécialité : INFORMATIQUE

Par

Mr Youcef HAMMAL

Thème

Sémantique Formelle des Modèles de Parallélisme Orientés Objets et Temps réel : Application aux Diagrammes Dynamiques du Langage UML

Soutenue publiquement le : 15/12/2011, devant le jury composé de :

Mme IOUALALENE-BOUKALA Malika	**Professeur**	**Présidente,**
Mr AHMED-NACER Mohamed	**Professeur**	**Directeur de thèse,**
Mme ALIMAZIGHI Zaia	**Professeur**	**Examinatrice,**
Mr BOUFAIDA Mahmoud	**Professeur**	**Examinateur,**
Mr MEZGHICHE Mohamed	**Professeur**	**Examinateur.**

Remerciements

Je tiens à remercier vivement Mr *Mohamed AHMED-NACER*, Professeur à l'USTHB, et lui exprimer toute ma reconnaissance pour avoir encadré mon travail de thèse, pour ses conseils, et ses encouragements pendant ces longues années.

Je remercie particulièrement Mme *Malika IOUALALENE-BOUKALA*, Professeur à l'USTHB, pour m'avoir fait l'honneur de présider le jury de ma soutenance.

Je remercie également les Professeurs *Zaia ALIMAZIGHI* de l'USTHB, *Mahmoud BOUFAIDA* de l'Université de Constantine, et *Mohamed MEZGHICHE* de l'université de Boumerdes, pour l'intérêt qu'ils portent à mon travail en acceptant d'êtres examinateurs de cette thèse.

Je tiens aussi à remercier les membres de l'équipe de recherche MOVES du laboratoire LSI de l'USTHB, pour m'avoir offert un espace approprié pour mener à bien mes travaux de recherche et discuter des diverses approches y afférentes.

Je remercie également tous mes collègues enseignants du département d'Informatique avec qui j'ai le plaisir de travailler.

Pour finir, je remercie mon conjoint *Assia* pour sa disponibilité, son aide et son soutien durant toutes ces années pour concrétiser ce travail.

University of Science and Technology Houari Boumediene

Faculty of Electronic and Computer Science
Department of Computer Science

Doctoral Dissertation

Formal Semantics of Concurrent Object-Oriented and Real-Time Models: Application to the Unified Modeling Language Dynamic Diagrams

Youcef HAMMAL

submitted in partial fulfillment of the requirements for
the doctorate degree in Computer Science

15/12/2011

Jury:
Prof. BOUKALA Malika President,
Prof. ALIMAZIGHI Zaia Member,
Prof. BOUFAIDA Mahmoud Member,
Prof. MEZGHICHE Mohamed Member.
Supervisor: Prof. AHMED-NACER Mohamed.

Abstract

This thesis deals with the formal semantics of active objects and components in reactive and safety-critical systems whose internal behaviors are depicted by means of UML state machines and interaction protocols are depicted by UML sequence diagrams.

First, we propose a formalization process throughout many steps in order to comprehensively tackle the syntactic and semantics complex features of UML state machines. This translation consists in mapping expanded state machines into modular Petri nets whose reachability graphs are gradually built up according to two run-to-completion policies and then augmented with timing constraints.

Similarly, we provide UML sequence diagrams with branching time semantics where few generalized algebraic operations are defined on yielded graphs so that formal definitions of interaction operators could be given in a compositional manner. We present as well, some algorithms to extract time annotations that adorn sequence diagrams and transform them into timing constraints in our graphs.

Such formal methods aim to alleviate the hard task of consistency checking between sequence and state diagrams depicting, respectively, specification and implementation models of reactive systems. Besides an exhaustive approach which copes well with most of interaction operators, we propose, as well, modular exploration methods of dynamic diagrams to yield partial specification and implementation graphs which shall encompass, respectively, intended and realized interactions of each component with its environment. Then, these graphs are used for incrementally checking both the compatibility between the system components, and the consistency between each component state machine and all sequence diagrams where this is involved.

Lastly, we deal with the behavioral substitutability and refinement of state machines related to active objects. We present new subtyping relations which allow us comparing behaviors of previous and new objects and components respectively within interleaving and true concurrency semantics. These relations are based on optimal variants of branching and forth-back bisimulations and aim at preserving service availability and correctness properties in highly reactive and complex systems.

Contents

1 Introduction **1**
1.1 Reactive concurrent and real-time systems . . . 2
1.2 Concurrency and the object oriented paradigm . . . 5
1.2.1 Objects and messages . . . 5
1.2.2 Inheritance and subtyping . . . 6
1.2.3 Objects and concurrency: active objects . . . 7
1.3 Detailed problem statement: using UML for modeling reactive systems 8
1.4 Contributions of this thesis . . . 11
1.4.1 Formal semantics of UML state machines . . . 11
1.4.2 Formal semantics of UML sequence diagrams . . . 13
1.4.3 Consistency checking of dynamic diagrams . . . 13
1.4.4 State machine refinement and substitutability . . . 15
1.5 Outline of this thesis . . . 16

2 A Road-map for Formalizing UML State Machines **18**
2.1 UML state machines . . . 18
2.2 Current work in formalization of StateCharts . . . 19
2.2.1 StateCharts semantics . . . 19
2.2.2 Exploration of reachability graphs . . . 22
2.3 Towards a behavioral model for UML state machines . . . 22
2.3.1 Some early expansion rules . . . 22
2.3.2 Discussion on the informal semantics of UML state machines . 27
2.3.3 Preliminary decisions for an efficient dynamic semantics . . . 30

3 Formal Semantics of UML State Machines **34**
3.1 Implementation model of state machines . . . 34
3.1.1 TPN-Component syntax . . . 35
3.1.2 TPN-Component semantics . . . 38
3.2 Mapping state machines into modular time Petri nets . . . 38
3.2.1 Translation patterns . . . 38
3.2.2 Translation algorithms . . . 48
3.3 Simulation phase . . . 55
3.3.1 Why using asynchronous transition systems? . . . 55
3.3.2 Modular generation of marking graph . . . 57
3.4 Adding time constraints in the marking graph . . . 67

3.4.1 Time annotations on UML state machines 67
3.4.2 Mapping time annotations into time constraints 69
3.5 Conclusion 72

4 Formal Semantics of UML Sequence Diagrams 73
4.1 Introduction 73
4.2 Motivations and related work 74
4.2.1 Fragment-based specifications versus monolithic model specifications 75
4.2.2 Branching time semantics versus trace semantics for sequence diagrams 75
4.2.3 Related work in formalization of UML Interactions 76
4.3 Interactions and sequences diagrams 78
4.3.1 Features of sequence diagrams 78
4.3.2 Outline of the translation approach 80
4.4 Formal model for an Interaction behavior 82
4.4.1 Choice operation 83
4.4.2 Parameterized cartesian product 84
4.4.3 Star Operation 84
4.4.4 Handling of synchronous messages 86
4.5 Formal semantics of Interaction fragments and operators 87
4.5.1 Lifeline of a participant 87
4.5.2 Basic Interaction fragment 87
4.5.3 Choice operator *alt* 89
4.5.4 Operator *break* 89
4.5.5 Operator *opt* 89
4.5.6 Operator *par* 89
4.5.7 Operator of strict sequencing seq_s 90
4.5.8 Operator of weak sequencing seq_w 90
4.5.9 Operator *loop* 91
4.5.10 Operators *neg* and *assert* 91
4.6 Extraction of timing information 93
4.6.1 Enhancing graphs with timing constrains 93
4.6.2 Extracting time constrains from sequence diagrams 94
4.7 Conclusion 96

5 Consistency Checking of Dynamic Views 98
5.1 Introduction 98
5.2 Modular versus exhaustive consistency checking 101
5.3 Exhaustive consistency checking of dynamic diagrams 103
5.3.1 Formalization of sequence and state diagrams 103
5.3.2 Exhaustive consistency checking of dynamic diagrams 105
5.4 Modular consistency checking 111
5.4.1 Partial reachability graph 112
5.4.2 Composition and compatibility 117
5.4.3 Partial interaction model 121

5.4.4 Comparison criterion 121
5.4.5 Fulfillment relation within interleaving semantics 125
5.4.6 Fulfillment relation within true concurrency semantics 132
5.5 Conclusion 134

6 State Machine Refinement & Substitutability 135
6.1 Introduction 135
6.1.1 Liskov behavioral substitution principle 135
6.1.2 Motivation and related work 137
6.2 State machine refinement in UML specifications 139
6.3 Preliminary concepts: strong branching bisimulation 140
6.4 Behavioral subtyping relations within interleaving semantics 144
6.4.1 Case 1: The new component adds new provided/required services without removing any required services 144
6.4.2 Case 2: The new component adds new provided/required services with removing some previous required services 148
6.5 Compatibility checking 152
6.6 Substitutability relations in true concurrency context 156
6.6.1 Preliminary concepts 156
6.6.2 Substitutability relations 157
6.7 Conclusion 158

7 Conclusions 160
7.1 Summary 160
7.2 Future research directions 162
7.2.1 Consistency checking of dynamic diagrams 162
7.2.2 State machine refinement and substitutability 164
7.2.3 Compatibility and composability checking of components 164
7.2.4 Runtime verification and model based testing 164

Chapter 1

Introduction

Nowadays, the unified modeling language has become the defacto object oriented language for documenting and designing software systems. Among these systems, we notice the presence of reactive and safety-critical systems including various hardware and software components which operate concurrently by means of different synchronization and communication mechanisms. Unlike “common” transformational software systems[1], reactive systems are characterized by continuous and intensive interactions with their environments and their reactions to external stimuli must be correctly achieved within strict time intervals. Otherwise, the violation of timed constraints produces critical and irreversible consequences. Therefore, checking their reliability and safety becomes as important as review of their functional requirements and so, to enable such analysis capabilities, reactive systems need to be specified with proper modeling notations.

In this setting, the Unified Modeling Language (UML) [103] is a multi-paradigm language which copes well with modeling and design of the aforementioned systems by offering many visual and flexible notations for description of their various aspects including time constraints. Also, UML includes high-quality design concepts such as abstraction, encapsulation and modularity which help manage the complex structures of reactive systems.

In this thesis, we develop an UML-based object computing model integrating both active objects [2] as concurrent and potentially distributed computational entities, and components as both fine-and coarse grained abstractions over the computational space. The object model is based on a formal semantics we propose for the UML, mainly its state and sequence diagrams, and mapped into abstract transition systems.

[1] Transformational systems get inputs, perform related operations on them, display outputs, and then stop.

[2] An active object is a mutable shared component [88] which progresses within its own thread.

As we are promoting the use of UML as specification and design language of reactive concurrent and real-time systems, we need to ascribe UML and particularly its dynamic diagrams with formal semantics in order to enable the rigorous and automatic analysis of such mission and/or safety-critical systems. We focus on the formalization and checking process of state and sequence diagrams regarding the fact that these are the common models used for describing, respectively, the individual behavior of active objects and their external interactions.

The aim of this chapter is to give a brief outline of the background and some of the terminology used in this thesis. It gives details on the systems this thesis is focused on, and an overview of the methods used to correctly specify and develop them, and explains why they are necessary but often insufficient to avoid system failures. To this end, the current chapter is organized as follows: Section 1.1 introduces the notion of a concurrent reactive and real-time system, and explains the necessity for the use of formal methods in their development process. Section 1.2 explains the object oriented paradigm used for modeling and programming these systems. Section 1.3 presents the detailed problem statement, that is, the use of UML as modeling language of reactive systems and in Section 1.4, we present the contributions of our work. Lastly, Section 1.5 outlines the structure of this thesis.

1.1 Reactive concurrent and real-time systems

David Harel and Amir Pnueli coined the term reactive systems in [69] to describe complex computing systems that continuously interact with their environment by exchanging information with it. In fact, these systems react to stimuli from their environment by possibly changing their state or mode of computation and, in turn, influence their environment by sending back some signals to it or initiating some operations that affect the computing environment.

Reactive systems are inherently parallel systems and a key role in their behavior is played by communication and interaction with their environment. A standard computing system can also be viewed as a reactive system in which interaction with the environment takes place only at the beginning of the computation (when inputs are fed to the computing device) and at the end (when the output is received). However, reactive systems maintain a continuous interaction with their environment, and we may think of the computing system and its environment as parallel processes that communicate with each other. In addition, non-termination is a desirable feature of some reactive systems. In contrast to the setting of standard computing systems, we certainly do not expect the operating systems running on our computers or the control program monitoring a nuclear reactor to terminate.

Moreover, the correctness of many reactive systems not only depends on their generated outputs, but also on the time at which they become available. Such systems whose specifications encompass temporal aspects of their behaviors are referred to as real-time systems .

Many contemporary computer systems qualify as being real-time sensitive, ranging from complex business information systems to telecommunication routers, and even highly specialized controllers to be found in modern vehicles such as cars, passenger aircraft, or space shuttles. The degree of sensitivity or responsiveness determines whether the system operates under hard or soft real-time constraints. The constraint for hard real-time systems is that no deadlines must be missed during execution, whereas for soft real-time it is acceptable to miss some deadlines, occasionally. For instance, the timing requirements of a business information system are inherently different to those of an airbag control system. A common characteristic of real-time systems, however, is that they may, and increasingly do, consist of many components operating in parallel and reacting to every stimulus from the environment; they are then known as concurrent (real-time) systems as well as reactive systems [105].

In the past, reactive systems were mostly centralized systems (or single-processor systems) consisting of a single CPU, its memory, peripherals, terminals or other physical inputs such as sensor devices. However, the development of increasingly compact and powerful microprocessors, as well as advances in network technology, have led to a more decentralized architecture of many such systems, now consisting of a collection of independent computers or CPUs, each connected by a shared network. Such systems are called distributed systems and are conceived by the user as being one single system. For example, an anti-lock brake system (ABS) is a physically distributed real-time system that the driver perceives as a single functionality in the car. In reality, several CPUs, sensors, and actuators are involved in order to activate the system in a timely precise manner. The overall correctness of the ABS therefore depends on the correctness of every subsystem. However, the ability to show correctness of a distributed real-time system is usually indirectly proportional to the number of subsystems, interactions, and general dependencies, physical or logical. Capturing the entire state space of such systems is, therefore, generally very hard. That is why, they must be specified and verified in terms of their (abstract) behaviors and if possible, in a modular way.

A recent and rapidly evolving development complicates reasoning over distributed reactive systems even further: distributed reactive and real-time systems are increasingly embedded; that is, they are integrated into physical devices other than computers, such as cars, aircraft, mobile phones, or even washing machines, which impose more or less strict deadlines on the controlling software, and are often part of safety-

critical applications in that a failure of these systems can have catastrophic impact on the environment, or cause injury and even death to human beings. Examples of safety-critical systems are airbag or brake controllers in modern cars as well as control systems of nuclear power plants.

It is a primary goal of software and systems engineering to develop systems which are correct with respect to a precise specification describing the systems to be constructed in an unambiguous manner. Therefore, formal methods such as static verification by means of model checking [68], or deductive reasoning by means of theorem proving [39] get employed in the design and development process, if at all, in order to prove that a system satisfies a set of predetermined and desired properties.

The term formal method refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. Generally, specifications used in formal methods are defined by well-formed statements in a abstract modeling language and the formal verifications are rigorous deductions in that formalism, i. e., each step follows from an inference rule and can thus be checked by a mechanical process. Naturally, many such techniques are nowadays tool-supported and even fully automated, e. g., model checking finite state spaces. Common examples of formal methods often target the early design stages of the system development process, such as checking of design documents and models, or facilitating their creation in the first place.

Now that we have an idea of what reactive systems are, and of the key aspects of their behavior, we can begin to consider what an appropriate abstract model for this class of systems should offer.

Actually, there is not one but different concurrency formal models that have been proposed making it thus difficult to compare them or set up syntactic and semantics mappings between them, though these could be apparently classified into the following categories of abstract models: Petri nets [107, 89], Temporal Logics [105, 68], Process Algebras [91, 67], Formal Description Languages [14], StateCharts [58, 62, 63, 126, 29], Message Sequence Charts [76, 64, 92], Communicating Automata [26, 27], Parallel Logic Programming Languages [39], and Actor Languages [2, 6].

Indeed, the difficulty of defining a formal comparison or unification method of concurrency models arises from the disparity of their underlying syntactic and semantics concepts, some of which are given below :

- The basic entity of parallelism may change from a model to another: it may be a process, Horn clause, object, function, actor.
- The manner of parallelism expression and sequential composition may be either implicit or explicit.

- The interaction mode can be achieved either by variables sharing or by messages transmission.
- Interaction topology between basic entities may be either static or dynamic.
- The semantics of a model can be defined with respect either to true parallelism or interleaving approach.
- External conditions may be imposed on program execution such as: equity, bounded execution time.
- The calculus distribution among physical processors of a machine may be either implicit or explicit.
- Other concepts make differences between languages such as the semantics kinds used to define them. Considering Time and probabilistic aspects matters too.

In this setting, we have explored in [46] some of the few existing comparison methods of concurrent languages and then we have proposed a separation schema (instead of a unification method) that is based on the algebraic structure complexities. Any language is viewed as an algebraic structure, which consists of a set of partial operations and some equivalence relation. Moreover, this approach uses the concept of language embedding to find out whether a mapping exists or not from a category of languages to another one such that it preserves both their syntactic and semantic structures (at different levels). The separation schema shows that it's not possible to get together two different languages L and L' in a same category K if a language property P preserved by K-embeddings, is satisfied by L' but not satisfied by any sub-language of L [46].

1.2 Concurrency and the object oriented paradigm

The object oriented programming knows at present tremendous growth and has applications in many fields. It features a programming style that includes (encapsulates) data and operations upon them as a single entity called object [42, 118, 71, 90, 77].

1.2.1 Objects and messages

In every object oriented methodology a system is conceived, by analogy with the real life, as a collection of interacting objects. The objects are abstractions of their real life counterparts. They exhibit only those properties which are meaningful for the system. However, the static view of objects is augmented with functional and dynamic aspects. An object has thus a state (i.e., attributes), a behavior (i.e., functions) and an identity.

The data in the objects is stored in variables not accessible to other objects: they are strictly private. By encapsulating attributes and functions together, the functional view is not anymore applied to the whole system, but localized in particular objects. The only way objects can interact is by sending messages to each other. Such a message is in fact a request from the sender for the receiver to execute a procedure. Such procedures, which are executed in response to messages, are called methods. The receiver decides whether and when it executes such a method and in some cases it even depends on the receiver which method is executed.

Computation or information processing is represented as a sequence of message passing among objects. Hence, interaction among objects respects the abstraction / encapsulation principles. The objects alone are responsible to maintain their local data in a consistent state.

Objects have a dynamic nature. Indeed, Object is a run time notion; every object is an instance of a certain class, created at execution time and made of a number of fields.

As we have already pointed out, objects are semantic entities. On the syntactic level, the corresponding notion is that of a class. In fact, a class is a description of one or more objects with a uniform set of attributes (similar properties) and services (common behavior), including a description of how to create new objects in the class.

1.2.2 Inheritance and subtyping

From the programmer's (or implementer's) point of view, it is very convenient when defining a new class, to start with all the ingredients (attributes and methods) of an existing class, and to add some more and/or possibly redefine some in order to get the desired new class. The new class is said to inherit the attributes and the methods from the old one. This can be repeated several times, and one can even allow a class to inherit from more than one class (multiple inheritance). In this way a complete inheritance hierarchy arises. By sharing code among classes in this way, the total amount of code in the system can be significantly reduced and its maintenance simplified.

However, it is worthy to note that subtyping and inheritance (i.e., subclassing) are related, but quite different concepts. In some cases they may be even contradictory. Both concepts are very useful and should be present in every object oriented method, but they reflect different concerns. The separation between subtyping and subclassing is analogous to the separation between an interface and its implementations [88, 134, 98].

Indeed, subtyping is concerned with the externally observable behavior of objects

i.e. with what the objects are expected to do and not with how they do it. If objects from a more specialized class (i.e., subtype) have all the observable properties (or behavior) of the objects in a more general class (i.e., supertype) and possibly some more, we can safely use these objects in a context requiring objects from the more general class. The externally observable behavior of objects in a class is given by their class interface. The class is said to implement this interface and the interface is a property of the class. On the other hand, inheritance is concerned with the internal structure of objects, their attributes and the code they execute for their methods. In other words, inheritance is concerned with how the objects do what they do. Inheritance is just a mechanism for implementing objects of one class by reusing the implementation of another.

1.2.3 Objects and concurrency: active objects

The localized vision of information, independence between objects, and the terminology of message passing create a natural need for distribution and parallelism. However, most existing object oriented languages are sequential in nature (Smalltalk [42], C++ [118], CLOS [71], Eiffel [90]). Their formal foundation could be understood type-theoretically, using a correspondence between object concepts with those of typed lambda calculus [16].

Such a lack for concurrent objet oriented languages could be explained by the fact that a pure one shall propose its proper syntactic constructs for introducing parallelism and synchronization mechanisms. Obviously, this would need new programming paradigms instead of simply extending sequential languages with additional libraries that exploit the operating system multi-threading functionalities.

As there are many ways to introduce parallelism in an object oriented language, different models have been proposed for embedding concurrency within object oriented systems: the Actor languages [2, 6], Parallel Object-Oriented language (POOL) [3, 4, 5], $\pi o \beta \lambda$ [78], Abacus [100, 99], Maude [21], Concurrent Object-Oriented language (CO-OPN) [81, 9], CoOperative Objects (COO) [113, 12], Object Petri Nets [80], and UML model of Active Objects [104, 103].

However, there is little agreement about their formal foundations and underlying concurrency concepts.This is in particular a consequence of the various synchronization mechanisms which can be used to explain message passing. For example POOL, $\pi o \beta \lambda$, CO-OPN and Abacus use synchronous communication while Actor and Cooperative Objects languages use asynchronous communication. Furthermore, not all the concurrent object oriented languages agree on the parallelism granularity. Hence, some languages consider all objects as active with a local activity of their own whereas

in other languages only some of objects have their own threads of control and can evolve in a parallel way. They are called active objects whereas the others are said passive objects since they only do react to stimulus from active objects. Moreover, concurrency in some languages (CO-OPN, Actors) could be realized both among and within objects, thereby enabling parallel activities to be achieved inside objects, too.

Considering the formal foundations of the aforementioned languages, these have been assigned various semantics. For instance, while CO-OPN and COO use Petri nets for describing control structures of objects and their encompassing systems, POOL and Abacus objects' behaviors are described using process algebras. On the other hand, Maude formal semantics is based on equational and rewriting logics and Actors were given many formal semantics based on Petri nets and transition systems.

Recently, the UML model for active objects [104, 103] has been proposed as a unifying model which allows intra and inter objects concurrency and enables the following synchronization policies:

- Asynchronous message passing based communication between objects (as in Actor and COO languages) where messages are queued and handled by receiving objects in order of their arrival.

- Synchronization pattern (as in POOL and CO-OPN) that is based on a message passing in a way like the ADA rendez-vous; the sender and the receiver wait for each other until both are ready to exchange the message, then the receiver executes the specified method, it returns the result to the sender, and after that both pursue their own activities in parallel again.

- Synchronous communication pattern similar to the Java synchronous method call message passing [77] between objects (some of which are passive objets); whenever an active object sends a method call to a passive object it becomes blocked until the latter returns a result to the former. To manage concurrent accesses by active objects to a shared passive object (and avoid thus race conditions), this one has to be embedded within monitors.

1.3 Detailed problem statement: using UML for modeling reactive systems

Reactive (real time and concurrent) systems have extraordinarily high safety and reliability requirements. Instead of being developed in assembly language without a formal design approach, many of these systems are now being designed using structured and object-oriented approaches. There are many reasons for this change. On

one hand, it is being driven by the growing complexity of the current generation of the reactive systems. On the other hand, an important driver is the decrease in the available time-to-market for reactive components. Not only must developers contend with vastly more complex requirement sets, and combine these individually complex components together into more complex interacting systems, they must do it in less time than ever before.

The Unified Modeling Language (UML) which is a high level and multi-paradigm modeling language has proven useful, just as well, to design and document reactive systems. Indeed, this OMG standard language [101, 103] offers a collection of visual, user-friendly and flexible notations for expressing artifacts representing various structural and dynamic aspects of reactive systems. UML is also based on an object-orientation involving design concepts suited to deal with the inherent complexity of these systems such as abstraction, encapsulation, and decomposition of systems into objects. Furthermore, its extensibility facilities make it easier to improve UML notations to address timeliness issues of safety-critical systems as pointed out in [102]. However, in spite of the precise syntactic aspects of UML notations, their semantics are informally defined and have been left deliberately too vague in order to allow various possible interpretations. The rationale behind such semantics variations is to allow sketching, in early development stages, design artifacts which would be refined later into implementations.

Unfortunately, such imprecise semantics could be considered as drawbacks for the rigorous design and review of reactive systems whose highly constrained behaviors should be safe and deadlock-free. Imprecise specifications are, in fact, a major source of risky inconsistencies and invalid choices at late design phases. Moreover, they prevent the use of rigorous analysis methods [10, 32, 36]. Therefore, to promote a thorough design of these systems by means of UML, it is important to provide this language (mainly its notations for the dynamic view) with an adequate formal semantics to meet their critical requirements and enable use of analysis and consistency checking tools.

To this end, formal methods have been proposed for mapping UML diagrams into rigorous formalisms such as Petri nets [107] and Formal Description Techniques [14] whose analysis techniques and tools could thus be exploited.

We focus particularly on UML state machines because they are fit for modeling behaviors of active objects and components. Furthermore, UML state machines [103] are an object-based variant of Harel's StateCharts [58] which have been largely used for specification of reactive systems [58, 60, 62, 63, 66].

On the other hand, UML includes notations for scenarios-based specification called sequences diagrams [103]. Indeed, these notations are widely used for describing design

requirements since these allow designers to express in an intuitive and visual way how system components (seen as black or grey boxes) have to interact and communicate in order to provide system level functionalities. Hence, a sequence diagram of the system under design can be seen as its specification model (or a piece thereof) depicting an expected interaction between its components and showing the messages that may be dispatched among them.

UML uses hence each of the separate state machines (i.e, UML StateCharts) to focus on describing the internal behavior of only one object or component in such a way that the execution of these state machines altogether, correctly realize the specified interactions of the sequence diagrams. That is, the state diagram consisting of individual state machines could be considered as the implementation model of the system under design, while its sequence diagrams constitute its specification.

It should be noticed that such a step is error-prone since it is manually achieved. Consequently, the key issue to support the design of reliable software systems (particularly safety-critical ones) is to be able to check that their sequence diagrams are faithfully synthesized into state machines which have to correctly implement the expected interactions. In other terms, designers have to check the semantics consistency of UML dynamic models composed of state and sequence diagrams.

In this thesis, we aim at automating the process of checking wether the design model of a reactive system behaves according to the collection of scenarios specified by its sequence diagrams. Such a process (illustrated by Fig.1.1) shall also test the absence of deadlocks and forbidden scenarios.

However, in spite of the expressiveness and precise syntactic aspects of UML notations, their semantics remain described in natural language with sometimes OCL formulas. Accordingly, to use automated tools for analysis, simulation and verification of parts of produced UML models, UML diagrams should be given a precise and formal semantics by means of rigorous mathematical formalisms [137, 61].

In view of the fact that UML diagrams lack for formal semantics, we propose a formalization method (defined in [48]) to translate positive (resp. negative) sequence diagrams into transition systems which depict all the allowed (resp. forbidden) computations of the system and preserve the branching structure. Similarly, we give a formalization method (defined in [47]), which translates the state diagram into an enhanced modular Petri net, whose reachability graph represents the behavioral (implementation) graph of the system under design.

We propose then some checking methods using an adequate comparison criterion which copes well with the features of sequence diagrams depicting only a subset of allowable and/or forbidden scenarios. The method consists in proving that any computation of the implementation graph either is within allowable scenarios or does not

lead to erroneous statuses of the specification graph related to the sequence diagram. Indeed, the system should avoid erroneous states because these depict forbidden computations specified by interaction fragments under the scope of *neg* or *assert* operators (see Subsection 4.5 for more details).

As software systems always evolve throughout the product life cycle, we address, just as well, some optimal substitutability relations among active objects such that properties checked on old state machines by aforementioned methods remain fulfilled by properly refined state machines.

1.4 Contributions of this thesis

The contributions of our thesis are explained over the following related research axes:

1.4.1 Formal semantics of UML state machines

We aim at providing behavioral diagrams of UML (namely, state machines [101]) with formal semantics in terms of a modular and enhanced variant of time Petri nets [89] because of their inherent concurrency and asynchronous communication features in addition to their formal nature.

Our target formalism is a component-based formalism derived from Petri nets [107, 89], which is enhanced with time intervals on transitions to represent timing information about event dispatching delays. Note that describing real systems using Petri nets, usually produces very complex models lacking in modularity and readability capabilities. That is why Petri nets are not used much in design activities whereas StateCharts have been adopted by software engineers thanks to their diagrammatic nature, intrinsic modularity, and too vague semantics which allow more flexibility to model design ideas. Conversely, the language of Petri nets is a mathematical formalism with rigorous semantics and analysis capabilities. Accordingly, we propose a formalization approach which defines a method for a comprehensive and progressive translation of state machines into modular Petri nets [47] and then gives two simulation algorithms of resulting Petri nets in order to yield intended behaviors in accordance with various dynamic semantics including that of the UML standard.

Unlike many other approaches [8, 13, 36, 41, 65, 79, 122, 127, 136] based on subsets of UML state machines, our translation method aims at supporting major StateCharts features such as hierarchy, orthogonality and support for history states. Indeed, these methods consider only fragments of flattened UML state machines discarding complex syntactic constructs and neglect the run-to-completion rules. Such limitations which simplify the translation process are due to the fact that such approaches are

almost always interested in business applications and transformational systems rather than reactive and safety-critical systems, the modeling of which needs an extensive use of complex syntactic constructs (e.g., orthogonal and hierarchical structures encompassing history states and boundary-crossing arcs). Moreover, the simplified dynamic semantics that these approaches adopt are not appropriate for safety-critical systems whose behaviors are event-driven and time-constrained and have thus to be carefully tackled during the translation and simulation processes.

Consequently, we propose to overcome the aforementioned difficulties with a multi-step formalization process which effectively copes with the large syntactic and semantics gaps between the high-level modeling language of UML state machines and the basic behavioral and rigorous language of Petri nets as follows:

1. The first step is an expansion process which transforms some complex shorthand notations into equivalent primitive syntactic elements of state machines in order to obtain simpler models that would be manageable in the next steps.

2. The second step is a gradual and comprehensive syntactic translation of expanded state machines into modular Petri nets. This constitutes a first major part of our method which thoroughly revises the approach given in [47]. Indeed, the target model is here a modular variant of Petri nets enhanced with time constraints on transitions and provided with new semantics mechanisms. Time annotations on state machines arcs are considered, as well, and various run-to-completion policies are accurately captured through the state space exploration of modular time Petri nets.

3. The last step is the simulation process (i.e., state exploration) in respect of two widespread dynamic semantics of StateCharts: The first one related to classical Harel StateCharts is based on the run-to-completion and priority rules of the STATEMATE tool [66] whereas the second semantics matches the UML standard whose rules were inspired from that of the RHAPSODY tool [62]. We thus select in the latter case among the semantics variations of UML state machines those that fit the features of component-based reactive systems in such a way that the adopted dynamic semantics conforms to the UML standard [103, 104]. For instance, passive objects are grouped under the control of active objects into components, each of which possesses a compound state machine having one common event pool and evolving in accordance with its own run-to-completion pace. Events generated in one step by contained objects are considered now as internal and thus handled within this same step. Such design decision helps overcome drawbacks of the UML semantics in respect of the time constraints of

reactive systems (cf. Subsection 2.3.3).

It is worth noting that the state exploration algorithms we give in the third step constitute a main contribution of our approach which aims to faithfully implement the dynamic semantics of UML StateCharts. In fact, all previous work based on Petri nets rely on their standard semantics and use, as is, their classical algorithm for the state exploration which does not match the particular run-to-completion policies of StateCharts. Furthermore, the component style adopted throughout our approach steps alleviates the state explosion problem that is inherent to the analysis of safety-critical models. Indeed, once we build the whole Petri net of a system from Petri net modules of its components, we explore its state space in a modular way by exploring and then combining partial reachability graphs of its modules.

1.4.2 Formal semantics of UML sequence diagrams

We present a new formal approach to defining branching time semantics for UML Sequences Diagrams in a denotational style [48]. Our approach deals with the partial order of events and yields lattice-like graphs that specify faithfully the intended behaviors by recording both traces of all interacting components together with branching bifurcations. We provide our mathematical structure with few generalized algebraic operations making it easy to give formal definitions of interaction operators in a compositional manner. Furthermore, we extend our formalism with logical clocks and time formulas over these clocks to express timing constraints of complex systems. Some given algorithms show how to extract time annotations of sequence diagrams and transform them into timing constraints in our timed graphs.

Obviously, this approach considerably alleviates the hard task of consistency checking of UML diagrams, specifically interaction diagrams with regards to state diagrams. Timeliness and performance analysis of timed graphs related to sequence diagrams could take advantage of works on model checking of timed automata [1].

1.4.3 Consistency checking of dynamic diagrams

We propose first an exhaustive checking method using an adequate comparison criterion which copes well with the features of sequence diagrams depicting subsets of allowable and/or forbidden scenarios. The method consists in proving that any computation of the implementation graph either is within allowable scenarios or does not lead to erroneous statuses of the specification graph related to the sequence diagram. Indeed, the system should avoid erroneous states because these depict forbidden computations as specified by interaction fragments under the scope of *neg* or *assert* operators (see Subsection 4.5 for more details).

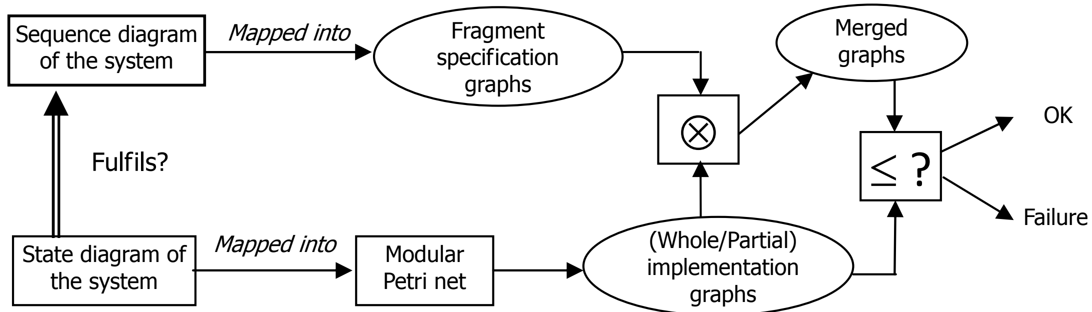


Figure 1.1: The approach for consistency checking of UML dynamic diagrams.

However, such a global checking process suffers from the state explosion problem when tackling medium and large systems. Moreover, one may need only to check that some new component could be safely substituted for the previous one without doing (from the scratch) the consistency checking of all the system that has been previously proven.

We therefore propose in a next part to thoroughly revise and extend our component based approach given in [49, 51, 52, 54] to address the aforementioned issues. Herein, we do not require that UML state machines implement all scenarios since these are not mandatory (as this was considered in [49, 54]). Furthermore, we deal with sequence diagrams which may contain interaction operators *neg* and *assert*. Hence, the checking process uses now an appropriate comparison relation over the behavioral models enhanced with the concept of erroneous statuses.

The other contributions of our work are explained throughout the steps of our modular checking process summarized as follows (see Fig.1.1):

- Given a component we would like to check, we make use of the formalization method of [47] in a modular fashion to extract its specification graph by considering only sequence diagrams including our component as participant. Besides interaction operators considered in [47], we deal in this part with *neg* and *assert* operators thanks to the concept of erroneous states.

- Similarly, we give a modular method to extract from the whole state diagram the partial behavioral graph in relation to the component.

- The two previous steps yield, thus, partial specification and implementation graphs which shall encompass, respectively, intended and realized interactions of this component with its environment. Afterwards, we give a fitting method which uses these behavioral graphs to incrementally check both the compatibility between the system components, and the consistency between each component

state machine and all sequence diagrams where this component is involved.

Our method for consistency checking copes well with sequence diagrams that can only specify partial allowable and/or forbidden scenarios (and not mandatory ones as in LSC [28]). Indeed, we use a fitting observational comparison criterion based on synchronization and hide operators over these graphs in order to check that each computation in the implementation graph, does either belong to the set of allowable sequences of the specification graph (reaching thus a final status) or does not lead the system to a deadlock or erroneous status according to this specification graph.

Such a component based checking approach can be gradually used to check the consistency checking of the whole state diagram of the system with its comprehensive interaction diagrams.

1.4.4 State machine refinement and substitutability

The formal semantics of UML state machine has to be provided with suitable observational relation to rigorously handle refinement and inheritance of active objects. We first discuss the shortcomings of informal refinement policies of UML StateCharts defined in terms of syntactical rules within UML specification documents. Instead, we formally deal with substitutability at the semantics level of StateCharts rather than the syntactical aspects.

Hence, we address the issue of the behavioral substitutability [88, 134, 98] of active objects and components whose underlying models are transition systems. We discuss the various observational criteria that were used in order to define subtyping relations which we analyze and criticize in connection to their capacity to preserve availability, in refined state machines, of services' sequences that were previously provided. Our main concern is that any updated component still fulfills properties which the oldest component used to fulfill. We therefore propose new strong observational equivalences which we use to define optimal subtyping relations between active components in reactive and safety-critical systems where involved components have to continuously react to sporadic and periodic stimuli of their environment. Such relations are defined in such a way that the high reacting capabilities (i.e., service availability) of the old components could be preserved and deadlines of their handling actions be met in their subtypes [50].

That is, even if only qualitative aspects of temporal ordering are considered, we propose strong observational criteria for comparing behaviors of components in such a way that old actions which were previously enabled at some state of the old component execution would continually be offered at equivalent states in new components even if these are interleaved with new actions. In other terms, performance of old services that

are claimed by the environment would not be rejected (or even postponed) because of executing new services. Hence, any of these services which should meet some deadlines could be enabled at all intermediate states reached by execution of new services. Of course, if there are other enabled actions in conflict, a non deterministic choice is taken[3].

Lastly, since our components can be provided with either interleaving semantics or true concurrency semantics, we define and discuss the substitutability relations in both these two contexts, especially, the issues about compatibility checking of our subtyping relations in connection with properties preservation expressed in terms of sequence diagrams.

1.5 Outline of this thesis

The aforementioned claims are materialized in our thesis as follows :

Chapter 2 - A Road-map for Formalizing UML State Machines. This chapter discusses the related approaches for assigning formal semantics to StateCharts and gives the motivation of our approach adopted. Then, It shows basic syntactic and semantic features of UML state machines, explains their expansion rules and discusses some design decisions so that their dynamic semantics meets the requirements of reactive systems.

Chapter 3 - Formal Semantics of UML State Machines. In this chapter, we first present our target formalism which is a component based time Petri net. Afterwards, we give key patterns sketching our translation method and finally we present the main algorithms which map UML state machines into our Petri net based formalism. We lastly, explain two methods for exploring state spaces of derived Petri nets depending on which run-to-completion policy is adopted.

Chapter 4 - Branching Time Semantics for UML 2.0 Sequence Diagrams. We show first the basic features of UML Sequences Diagrams and present our formal model along with its algebraic operations. Next, we give the semantics of sequence diagrams in a compositional style by combining the denotations of their interaction fragments using our algebraic operations. This chapter presents as well, a temporal enhancement of our graphs with logical clocks and temporal formulas and then it gives the method to extract into our timed graphs the timing constraints from time annotations on sequence diagrams.

Chapter 5 - Consistency Checking of UML Dynamic Views. This chapter presents more details about the motivation of our approach and the related work. We then present an exhaustive approach to deal with the consistency checking. Next, we

[3]Only a choice operator with priority can help more to meet deadlines [44]

present a modular approach by giving an algorithm to partially explore the Petri net related to the state diagram and yield the partial graph related to one component. Then, the compatibility issue related to composition of components, along with the modular consistency checking, are explained and discussed both in the interleaving and true concurrency contexts. Finally, a conclusion is given.

Chapter 6 - State Machine Refinement and Substitutability. In this chapter, we first argue the formal handling of substitutability at the semantics level of UML state machines rather than the syntactical aspects. We accordingly propose new subtyping relations that are based on strong branching bisimulations and which cope efficiently with service availability when dealing with refinement and inheritance of state-based objects. Next, we make use of the formal semantics of sequence diagrams given in Chapter 4 to prove the correctness of the substitutivity relations over State-Charts in relation to the sequence diagrams they fulfill. We address, as well, the issue of substitutability within the true concurrency semantics where previous subtyping relations are no longer suitable. To this end, we propose an adequate relation based on a new branching variant of forth and back bisimulation upon asynchronous transitions systems and discuss its properties.

Chapter 7 - Conclusions. Subsequently, we summarize herein the results of our work and discuss some future research directions such as the timed analysis topics of the behavioral graphs derived from UML state machines and sequence diagrams, consistency checking and substitutability within the non-interleaving semantics.

Chapter 2

A Road-map for Formalizing UML State Machines

2.1 UML state machines

As mentioned in the UML specification [101, 103], state machines are the object variant of Harel StateCharts [58]. A state machine shows “a behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions”. An event can be a signal, an operation invocation, a time passage or a condition change, whereas a state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for some event. Here, an event is an occurrence of stimulus that can trigger a state transition.

Note that a state may be either simple or composite: any state enclosed within a composite state is called a substate of that composite state and it is called a direct substate when it is not contained within any other state; otherwise it is called a transitively nested substate. If substates can execute concurrently, they are called orthogonal regions. For example, Fig.2.1 illustrates a state machine of an audio-visual device driver where the occurrence of the event “on” triggers the transition from the simple state ”STANDBY” to a concurrent composite state “ON” whose the two orthogonal regions “IMAGE” and “SOUND” are both sequential composite states.

A transition (arc) is a relationship between two states indicating that an object in the first (source) state will perform certain actions and enter the second (target) state when a specified event occurs and specified conditions are satisfied. There are also “Entry/exit” actions that are executed on entering and exiting the state, respectively.

When dealing with composite and concurrent states, the simple term “current state” can be quite confusing because more than one state can be active at a time.

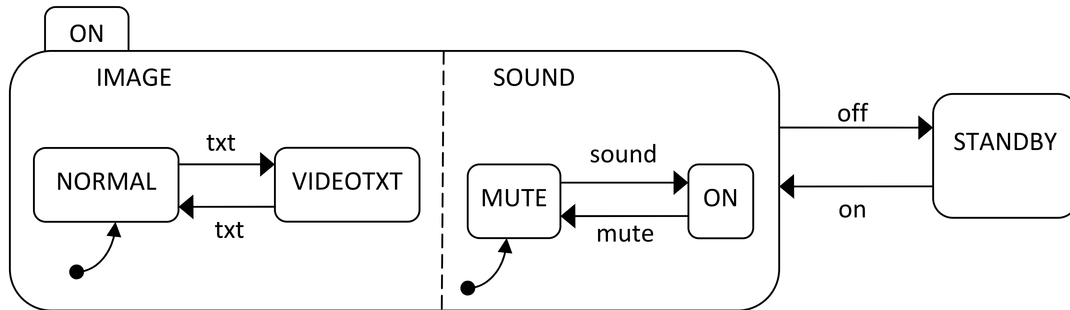


Figure 2.1: UML state machine of an audio-visual device driver.

If the control is in a simple state then all the composite states that either directly or transitively contain this simple state are also active. Furthermore, since some of the composite states in this hierarchy may be concurrent, the current active state is actually represented by a tree of states starting with the single top state at the root down to the individual simple states at the leaves. We refer to such a state tree as a state configuration or status. Also, any transition originating from the boundary of composite state is called a high-level or group transition. If triggered, it results in exiting of all the substates of that composite state executing their exit actions starting with the innermost states in the active state configuration.

2.2 Current work in formalization of StateCharts

2.2.1 StateCharts semantics

Numerous works have been proposed to overcome the several ambiguous points in the semantics of classical StateChart, as well as their object-oriented variants. Firstly, StateCharts were proposed as a structured design based modeling language [58] which supports particular concepts such as negative events and a broadcast-style of communication among StateCharts constituents.

Among approaches aiming at formalizing classical StateCharts, the main difference is whether changes that occur in a given step take effect in the current step or in the next one [95, 126]. The earliest trend based on Pnueli and Shalev's step semantics [106] emphasized a "strong" synchrony hypothesis where all events raised in a step have to be taken in the same step, thus violating the compositionality property. To overcome this drawback, authors of [121, 84] characterized that semantics using mappings from algebras of restricted StateCharts terms respectively to a domain of labeled transition systems and intuitionistic Kripke models.

In contrast, STATEMATE based semantics as pointed out in [66, 93] adopt a more

basic step policy where generated events in a step are sensed in next steps. However, authors of [29] propose a different compositional semantics for STATEMATE designs using the super-step policy favoring internal events against external ones as it has been adopted later in RHAPSODY. Thus, in late 1990s the object orientation had ultimately affected StateCharts and accordingly, authors of [60, 63, 62] customized the STATEMATE semantics to give rise to a new version referred to as RHAPSODY semantics which would eventually influence that approved for UML state machines [101, 103]. Instead of modeling an entire system with a full StateChart, (active) objects are described in RHAPSODY using (partial) StateCharts which communicate via both asynchronous signal broadcast and synchronous method call. Moreover, its run-to-completion rule compels each step to handle any method call generated within this same step. In this setting, it is worthy to mention the papers [119, 19, 20, 95, 126] which address the comparison of the variants of StateCharts including UML state machines and show the syntactic and semantics differences among them.

Concerning the description of the UML state machine semantics, UML specification documents [101, 103] use the meta-modeling level to describe UML using class diagrams and Object Constraint Language OCL [101] to capture the static relationship between modeling concepts. However, such languages are not adequate for critical-safety systems since they are too imprecise to describe the dynamic semantics of UML diagrams making it hard to avoid inconsistencies among UML models and leading to a number of problems relating to readability, use of rigorous design process and rigorous analysis [10, 36]. Therefore, much effort has so far been made to provide UML models with clear semantics in order to help designers achieve their analysis by means of verification and validation tools. Among the addressed works, we distinguish many categories of approaches:

A first family includes approaches that improve the metamodel of UML to overcome ambiguities of standard UML semantics. For instance, the authors of [33] add dynamic meta-modeling rules for the specification of UML consistency constraints and provide concepts for an automated testing environment. Another work [109] exploits the profile extension mechanism [102] to add definitions of some stereotypes and a set of UML diagrams that enable the specification of real-time systems and their properties. To specify properties of such systems, specification-classes are defined having predefined constraints presented in an extended variant of Timed Computation Tree Logic (TCTL). Likewise, authors of [15] enhance the UML state machines with time constraints for the specification of real-time behaviors in order to obtain the timed StateCharts of Kesten and Pnueli [74].

A second large family defines mappings from UML constructs into rigorous formalisms such as Petri nets and process algebras to carry out consistency checking.

This way, the approach of [36] exploits a formal notation like Z instead of OCL to formalize UML components so that the usual capabilities of Z become available for type checking and proving properties about components. In similar works, authors of [127] propose a mapping using the incremental two-way translation between UML and SDL concepts and authors of [23] define a mapping from UML models to their equivalent E-LOTOS expressions to form a single formal model. However, many drawbacks arise thereby; for instance, synchronous communication mode of process algebras compels adoption of the zero-time semantics and excludes asynchronous communication way. Also, faithfully mapping the UML semantics to a Z-like formal specification language can result in very verbose and cumbersome specifications. But, the main drawback of such translations is the use of only subsets of UML state machines and exclusion of several suitable concepts for reactive systems such as orthogonality, support of history states and boundary-crossing transitions.

In the same way, there are also graph-based transformations, like that of [41, 122] translating a subset of UML state machines into graphs via rewriting rules. Likewise, the authors of [13, 8, 65, 38, 79, 120] give dynamic semantics of fragments of UML models by means of enhanced Petri nets (respectively high-level Petri nets, object Petri nets, Colored Petri nets, and generalized stochastic Petri nets). Only draft heuristics or transformation rules are given in these papers to transform parts of the UML models into the target formalism.

Nevertheless, the aforementioned works deal only with a subset of state machines close to particular application fields and their translation rules are not well formalized. Moreover, discarded concepts, such as concurrent composite states and history vertices, are significant for modeling reactive and concurrent systems. Also, events queuing patterns of these approaches do not fit for situations where one event may trigger many transitions at the same time in various orthogonal regions.

Among relevant approaches, it is noteworthy to cite those of [97] and [108]. The first approach in [97] consists in translating UML 1.x models into communicating extended timed automata based on the RHAPSODY semantics with some restrictions (discarding asynchronous communication, a semantics based generation of the target model). The authors use also a specific (OMEGA) profile to express timing constraints and propose a lightweight extension of UML (i.e., observer classes) as a property description language instead of UML sequence diagrams. However, we deem that version 2.x of UML brings valuable syntactic and semantics improvements to state machines, as well as sequence diagrams, making it possible to exploit UML diagrams to model and specify important properties of systems [43, 48, 61]. As pointed out in [108], the authors propose a specific temporal logic and use it to formalize fundamental aspects of the semantics of UML state machines. But like for OCL, it seems hard for

modelers to deal with logic expressions which can easily become very cumbersome when used to describe complex systems.

In view of the above paragraphs, we present in this chapter a component based methodology to formalize UML state machines. We propose a new comprehensive translation method into temporal modular variant of Petri nets which are a widespread formal and concurrent language. Moreover, our approach deals step by step with all features of state machines such as hierarchical states, boundary-crossing arcs and main pseudo-states like history, though they cause many problems when jointly used.

2.2.2 Exploration of reachability graphs

Once the state machines of an UML model are mapped into linked time Petri net modules, we could proceed to the state space exploration of the composite system by building the less-complex reachability graphs of components with reference to any run-to-completion policy and then, accurately combine them to yield the whole state space. In fact, the exhaustive exploration of reachability graphs for complex systems mostly suffers from the state explosion problem. Furthermore, we frequently need just to check whether a new component can be substituted for another one without introducing bugs like deadlock or new unsuitable behaviors.

Consequently, we propose a method of partial exploration of the reachability graph with respect to one component such that we generate only partial paths of this main component interleaved with those of other components, the progress of which is necessary to unlock execution of the main one. Besides avoiding the common state explosion problem, the modular approach for exploring the state space enables to address issues related to components (like substitutability and compatibility checking) provided that such analysis techniques are based on some adequate observational equivalence over resulting graphs of components. Though such a practice is commonly used, a component based method is more suitable to deal with StateCharts than approaches of on-the-fly model checking [68] or partial exploration of reachability graphs in view of some properties one wants to check [7, 59, 94, 123, 124].

2.3 Towards a behavioral model for UML state machines

2.3.1 Some early expansion rules

Pseudo-states (i.e., transient vertices) are shorthand notations used to ease modeling of some specific cases. As we are interested in translating state machines into Petri nets to analyze them, we aim thus at expanding pseudo-states into a reduced and manageable set of primitive constructs to make the translation process straightforward

and efficient. Such a technique has often been used while assigning formal semantics to description languages (e.g., LOTOS [14], Process Algebra [130]. . .) by means of more basic behavioral specification languages (e.g., transition systems, Petri nets. . .). Therefore, before the translation step, the expansion rules below must be applied with respect to the well-formedness rules defined in [101, 103]:

Expanding history vertices: Two subcases are discerned:

- Any state with a shallow history pseudo-state shall be labeled with the "history" tag.
- Any composite state owning a deep history pseudo-state together with all its direct and transitively nested substates, are labeled too with the "history" tag. This rule ensures that handling of the history property is also achieved in its transitively nested substates.

Expanding initial vertices: Any target state of a transition originating from an initial pseudo-state, is labeled with the "initial" tag.

Expanding fork vertices: Target states of a transition originating from a fork pseudo-state are labeled with the "initial" tag. In order to simplify the translation process, we consider the common case where each target state is exactly the initial state of one orthogonal region in the concurrent composite state. The transition arc incoming to the fork vertex shall be replaced by an incoming transition arc to the edge of the composite state.

Expanding join vertices: We label with the "final" tag all substates in distinct orthogonal regions of a composite state, the outgoing arcs of which merge into a join (terminator) pseudo-state. The transition arc outgoing from a join vertex to one target state, will be a triggerless outgoing arc from the edge of the composite state to the target state. Note that a triggerless outgoing transition is a completion one without an explicit trigger. When all actions and activities of the active state are completed, a completion event instance is generated. This event is the implicit trigger for a completion transition. Also, a completion transition emanating from a concurrent composite state will be taken automatically as soon as the concurrent regions have reached their final states. Consequently, the join vertex which links together final states of orthogonal regions is the only point from which the triggerless outgoing arc can be taken.

Expanding merge vertices: We straightforwardly apply the previous process to junction vertices but dynamic choice vertices need a specific handling because the

former are semantic-free vertices that realize static conditional branches while the latter realize dynamic conditional branches. Vertices of both the two kinds are used to construct compound transition paths between states. In other words, they are used to chain together multiple incoming transitions (that may be triggered by different events) into one or multiple outgoing transition segments with different guard conditions. Furthermore, one of the outgoing transition segments whose guard is *true* is taken in both the two cases (if multiple transitions have guards that are *true*, a transition from this set is chosen). However, in the case of junctions, the guards are evaluated before the compound transition is taken, whereas in a compound transition where multiple outgoing transitions originate from a common choice point, the outgoing transition whose guard is *true* at the time the choice point is reached, will be taken. Hence, guards of the second segments are dynamically evaluated within the same run-to-completion step, making it difficult to handle choice vertices in a pure syntactic way.

- Expanding junction vertices:

  Expanding a junction (see Fig.2.2(a)) consists in unfolding the compound transition paths into usual transitions, each of which depicts a distinct couple of two segments, respectively, one incoming to and one outgoing from the junction node. Each new transition is thus triggered by the event on its incoming segment and guarded by the conjunction of guards of its two segments. The source state of each generated transition is that of its incoming segment and its target state is that of its out-going segment. As events are dispatched, one at a time, transitions triggered by the same event would be enabled; however, only one whose guard is *true* should be taken (provided that the diagram is well-formed).

- Expanding choice vertices:

  This case requires the concept of transaction introduced to deal with the dynamic semantics of state machines. A transaction represents as usual an uninterruptible sequence of dependent transitions to be fired altogether within one same run-to-completion step. Expanding a dynamic choice point consists in translating each distinct couple of an incoming segment to and an outgoing segment from the choice node into a pair of usual transitions that are separated by an intermediate new state and which should fire within a same transaction. The first transition of any couple represents the incoming segment to the choice node and the second transition represents an outgoing segment from it. The first transition has, thus, the triggering event of its related incoming segment along with its guards, whereas the

Figure 2.2: Expanding merge pseudo-states.

second transition is triggerless (since outgoing segments from pseudo-states are triggerless) and has the guard of its related outgoing segment. The first transition originates from the source state of its related in-coming segment and goes to the new intermediate state from which originates the second transition whose target state is that of its related outgoing segment. As illustrated by Fig.2.2(b), each pair of dependent transitions is considered as a transaction to let them execute altogether. The intermediate state is added to force the guard evaluation of the second transition after the first transition has been performed because its results may affect the evaluation. Note that we have used two intermediate states in Fig.2.2(b) to make it more readable although we could use one intermediate state for this case.

Expanding internal activities (Do-Activity): As we are interested in using state machines as a specification language to represent abstract (i.e., externally observable) behaviors of reactive systems for analysis purposes, we opt to not include internal activities of states since these generally do not have an effect on observable behaviors. Nevertheless, we present in Subsection 4.1.1 how to show on the generated Petri nets the places where internal activities would be carried out.

Metamodel of expanded state machines

We give below the metamodel (see Fig.2.3) depicting the abstract syntax of the expanded state machines obtained after pruning the pseudo-vertices by means of the expansion rules such as noted above. Indeed, the expansion phase can be seen as a pre-processing process which aims at rewriting all sophisticated shorthand notations (e.g., pseudo-states) and high level concepts (e.g., regions) by means of primitive

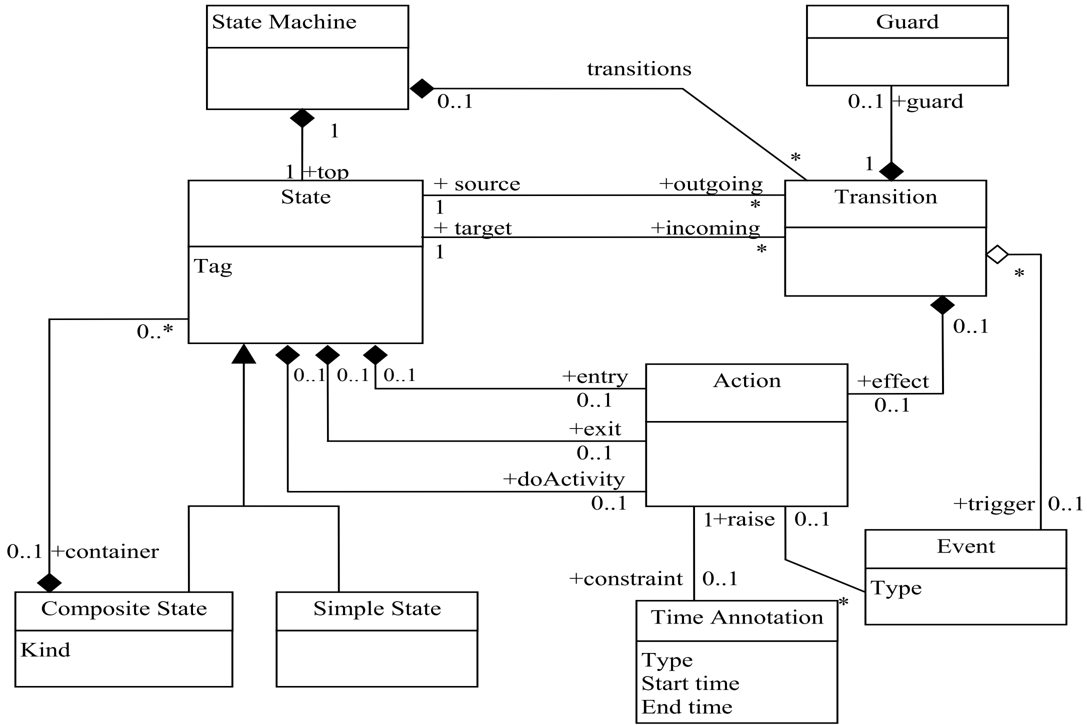


Figure 2.3: Abstract syntax of expanded state machines.

syntactic elements (states, arcs, ...). Hence, the translation and simulation phases become easier since they have now to manage few basic elements. Furthermore, once UML 2, as well as UML 1.5, state diagrams are expanded, the resulting state machines could be specified using a common simplified metamodel which encompasses their underlying syntactic elements we deal with in our formalization approach.

Mathematical structure for UML state machines

Consequently to the previous operation and before going through the translation phase, we summarize formally the aforementioned metamodel by means of a mathematical structure D modeling the expanded state machine. Such a convenient formalization leads to concise and obvious translation algorithms thanks to the use of mathematical symbols and mappings. Thus, A state machine is a tuple:

$D = \langle V, Kind, Tag, C, TA, S0, EVT, G, ACT \rangle$ where:

- V: set of states (vertices) with the topmost state S_0.
- $Kind : V \longrightarrow \{SimpleState, SequentialCompositeState, ConcurrentCompositeState\}$.

- $Tag : V \longrightarrow 2^{\{Initial,Final,History\}}$.

- $C : V \longrightarrow 2^V$ is a mapping that assigns to each state $s \in V$ its direct nested states. It can be generalized into a new mapping C^* which gives all direct and indirect nested states as follows: $C^*(s) = \bigcup_{i>0} C^i(s)$ where $C^{i>1}(s) = \bigcup\{C^{i-1}(s_j)|s_j \in C(s)\}$. We have for all $s \in V, s \notin C^*(s)$ and $C(s) = \emptyset$ if $Kind(s)$ is simple state. Note that the mapping C defines a partial order relation ($\subset$) among states we can depict as a tree.

- $TA \subseteq V \times EVT \times G \times ACT \times V$: is the set of transition arcs where EVT is the set of events, ACT is the set of actions and G is the set of guards.

- Note that ACT can be split up into two subsets respectively of synchronous and asynchronous actions as follows: $ACT = ACT_{Syn} \cup ACT_{Asyn}$ and $ACT_{Syn} \cap ACT_{Asyn} = \emptyset$.

  – Any action in ACT may raise events from the set EVT. Thus, there is a partial mapping which assigns to each action its set of raised events, $Gen : ACT \longrightarrow 2^{EVT}$.

  – Any action in ACT may have duration. Formally, there is a partial mapping $Duration : ACT \longrightarrow 2^{INT}$where: $INT = \{[d_1, d_2] \in \mathbb{R}^{>0} \times \mathbb{R}^{>0} \mid d_1 \leq d_2\}$ is the set of time intervals of positive real numbers.

  – Finally, there are three partial mappings *Entry*, *Exit*, and *doActivity* having the following same signature: $V \longrightarrow 2^{act}$.

2.3.2 Discussion on the informal semantics of UML state machines

The informal semantics [101, 103] is described in terms of the operations of a hypothetical machine whose main components are:

- an event pool which holds incoming event instances until they are dispatched,

- an event dispatcher mechanism that selects and de-queues event instances from the event pool for processing,

- and an event processor which processes dispatched event instances according to the semantics of UML state machines.

Event instances are generated as a result of some action either within the system or in the environment surrounding the system. Then, every event is conveyed to one or more targets by some means depending on the type of its raising action, its target, the properties of the communication medium, and various other factors. Thus, whilst the

event transmission is instantaneous and reliable in some cases, in others it may involve variable transmission delays or loss of events. When an event is received, it is placed on the event pool of its target. Notice that if the whole system owns a sole thread of control, all generated events are queued on the system queue whereas in a multithread system each thread will have a separate pool. Thereafter events are dispatched and delivered to the state machine for processing, one at a time. Finally, it is consumed when event processing is completed.

Moreover, no assumptions are made about time intervals between event reception, event dispatching, and consumption and the order of dequeuing is as well not defined. As expected, the semantics of event processing is based on the run-to-completion assumption which means that an event can only be dequeued and dispatched if the processing of the previous current event is fully completed. The processing of a single event is referred to as a run-to-completion step which represents a passage between two stable state configurations of the state machine. Hence, before launching a run-to-completion step, a state machine should be in a stable status with all actions (but not necessarily activities) completed. In case of active objects where each object has its own thread of execution, we are supposed to clearly distinguish the notion of run to completion from the concept of thread pre-emption. Namely, any thread can be preempted and its execution suspended in favor of another thread executing on the same processor. When the suspended thread is assigned processor time again, it resumes its event processing from the point of pre-emption and, eventually, completes its event processing.

In the presence of the concurrent states, it is possible to fire multiple transitions as a result of the same event as many as one transition in each concurrent state in the current state configuration. In such a case, the state machine selects a subset and fires them. If no transition is enabled, the event is discarded and the run-to-completion step is completed. On the other hand, when all orthogonal regions have finished executing the transition, the current event instance is fully consumed, and the run-to-completion step is completed. During a transition, a number of actions may be executed. If such an action is a synchronous operation invocation on an object(s) executing a state machine, then the transition step is not completed until the invoked object(s) complete its run-to-completion step.

In this respect, the UML 1.5 specification [101] states that: "*an event instance can arrive at a state machine that is blocked in the middle of a run-to-completion step from some other object within the same thread, in a circular fashion. This event instance can be treated by orthogonal components of the state machine that are not frozen along transitions at that time*". Conversely, this clause is removed from the UML 2 specification [103] though it copes well with situations where state machines depict

behaviors of software components enclosing many internal objects which cooperate by mutual methods invocations. We, then, adopt later in our method a tailored version of the aforesaid statement where the terms "orthogonal components of the state machine" are substituted by (or meant as) cooperating state machines within one combined StateChart related to a software component evolving under its own thread of control[1]. In this way, during a run-to-completion step we could handle all events generated within this step towards internal objects of the same active component.

In spite of the standardization effort of UML state machines [104, 103, 101], a thorough review of their dynamic semantics raises numerous questions about the adequacy of run-to-completion policy to the context of reactive systems. To sum up, any (macro) step will consist of a chain of reactions (micro-steps), which the first is triggered by an external event, thereby, raising one or more transitions. The following micro-steps are, then, triggered by internal events that have been generated by previous transitions taken within the same (macro) step.

Considering this policy, the next issues remain too vague for coping with time constraints:

- Should we also recursively handle in the same run-to-completion step any internal event be it asynchronous or synchronous, but which is generated within this step and directed to an object in the same active component? If yes, do we allow delaying for indefinite period external events even if these need to meet some time deadlines?
- How do we manage these internal events? Should we first insert them in some event pool of the related thread and thereafter dispatch them one by one but only after we have finished the handling of all transitions previously produced in the previous micro-step? Or, may all these events be dispatched in any order during the same macro step?
- In other words, is the choice of the next internal event to dispatch made in a non-deterministic or scheduled way from the pool of events?

As aforementioned, numerous approaches have been proposed to address these issues: The main one is the RHAPSODY based semantics [60, 62, 63, 97] where StateCharts are used to describe the behavior of objects and only external requests are treated one by one in a run-to-completion fashion. So, any operation call is handled immediately within the same step if it comes from inside the object but when an asynchronous event (signal) is generated, it gets queued on a system queue as described

[1]The term "Thread of control" is used here as an implementation-independent concept to characterize active components.

in [60]. Hence, the RHAPSODY based semantics matches more object-oriented programming languages where method calls are broadly used.

On the other hand, STATEMATE semantics fits more with highly reactive systems [66] because reactions to external and internal events, and changes that occur in a step, can be sensed only after completion of the step. However, it is worthy to note that events in [66] are only asynchronous and StateCharts semantics follows the structured analysis paradigm rather than the object orientation.

Finally, to let our approach support both STATEMATE and RHAPSODY hypotheses, we introduce, next, a number of improvements in our Petri net based model, which help simplify the analysis of reactive systems at early stages of their design.

2.3.3 Preliminary decisions for an efficient dynamic semantics

We explain, in this section, the design decisions we adopt to efficiently implement the UML dynamic semantics of state machines and give the motivations that lead us to explore additionally another established semantics. We appraise both of them in view of their suitability for the formal analysis and verification of reactive and safety-critical systems.

Abstract representation of events and actions

As the rationale behind our formalization process is the rigorous analysis of the resulting behavioral models, we shall consider state machines at a more abstract level and focus on their observable behaviors. In this respect, we discern two kinds of events and relate them to (inter)actions:

- Non blocking events (signal, timeout, return of method result) which are issued by asynchronous interactions; the state machine does not need cooperation from the environment to handle such events and continues its execution after producing them.

- Blocking events raised by synchronous interaction such as method invocations; the state machine stops its execution after sending such an event until receiving a response.

At an abstract level of specification, we are mainly interested in observing instantaneous actions which generate or consume all previous kinds of events. Therefore, we use the process algebra concept of communication *gate*[2] supporting these basic actions

[2]The concept of "gate" is different from the notion of "port" since the former represents abstract points of synchronization between concurrent components whereas the latter is service-oriented and denotes full interaction protocols between them.

such that we could denote them with one of the following forms (see Fig.2.4):

- $g!e$ denotes an event sending (denoted also for more simplicity by $e!$). If e is a signal or timeout then the action is not blocking else if e is a method invocation then the resulting action is a blocking one. If such an action occurs, it blocks up until a result is returned.
- $g?e$ denotes an event receipt (referred to also by $e?$) which is always a blocking action.

Moreover, gates may be either unary, binary or multi-way points of communication:

- A unary gate is visible only within its owner component. So, the gate is known and used only inside this object by its internal constituents to carry out internal operations.
- A binary gate is a rendezvous gate between exactly two components. Once these two components are combined, this gate is hidden to their environment so that no interaction with a third component may happen on it. For instance, method invocations always use this kind of gate to address a service request to a required component (server object).
- A multi-way gate is a broadcasting gate. It remains visible even after using it to handle communications between many subcomponents. As a result, every signal we broadcast on this kind of gate can be captured by other components having access to it.

Efficient handling of UML dynamic semantics

Considering the strict time constraints of reactive systems, one could find out the run-to-completion policy of the UML semantics somewhat inadequate. Indeed, several actions in the state diagram of a component have to meet given time constraints. Accordingly, it seems inappropriate to let its entire state machine block in the middle of a synchronous or long action execution whilst other urgent actions (even if external) need to be handled by this state machine. We make, therefore, two preliminary decisions that help tackle efficiently synchronous actions and actions with no zero duration whatever the adopted semantics:

First, as we are interested in use of abstract models for analysis purposes, we split these actions into two more basic, atomic, and instantaneous actions which should be respectively handled within two different steps. This assumption is closer to the synchrony hypothesis which has been claimed as a basis of most definitions of StateCharts semantics [121, 84].

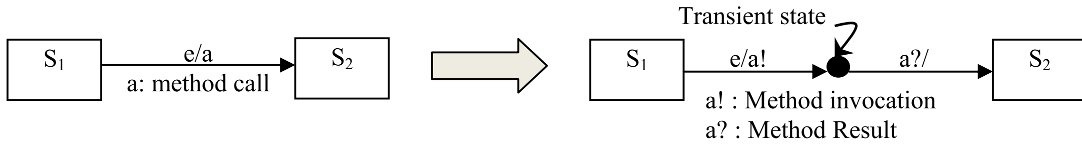


Figure 2.4: Splitting synchronous action into instantaneous basic actions.

In fact, according to this assumption, actions are assumed to be executed instantaneously or fast enough with regards to the arrival of external stimuli. Hence, the computational model of a reactive system would consist of a sequence of steps where each step contains two phases: In a first phase all parts of the state machine (individually or cooperatively) execute an arbitrarily long but finite sequence of instantaneous actions. In phase two, all parts let some amount of time synchronously progress. In fact, these practical assumptions simplify the formal semantics and help us build the behavioral models of systems whose actions may have no zero duration by splitting them into pairs of atomic and instantaneous actions (e.g., begin and finish actions, or invoke and response receipt actions as depicted in Figure 2.4.

The transient vertices introduced are differently handled depending on which semantics is adopted. For instance, they are considered as unstable statuses according to the UML semantics since the only way to exit them is to achieve the second part of the original transition which consists in the receipt of some result from either a transition triggered within the same run-to-completion step of its enclosing state machine or a transition performed within another cooperating state machine. The time it could take for the control to leave transient vertices is arbitrary without regard for the urgency of any concurrent action whose triggering external event could be available on the event pool.

Second, we propose a design decision promoting a component-based design of reactive systems to manage their complexity and time constraints: each active object will have its own thread of control and passive objects are grouped into components, each of which possesses an active object controlling its passive objects. According to the UML specification [103, 104], passive objects need to run on the thread of the calling active object. Hence, when the active object receives required services from other components (namely, their enclosed active objects) it then invokes related methods of passive objects enclosed in its component. Consequently, the system under design is seen as a collection of active components evolving in a concurrent way. Any one of them has its own event pool and progresses according to its individual pace of run-to-completion steps. However, from time to time one may require or provide services to the others by means of event instances which are conveyed and queued in event pools

of target components (independently of the statuses these components are in).

In this fashion, the component-based grouping of objects state machines allows us to adopt a trade-off rule which matches the standard semantics of UML and improves, just as well, the handling delays for synchronous and long transitions. An event instance can currently be dispatched to a state machine of any component that is blocked in the middle of the run-to-completion step provided that this event comes from an object within the same thread of control. Hence, orthogonal parts of the component state machine that are not frozen could carry out this event instance whose outcome would probably help unlock the handling of the waiting synchronous action.

Investigation of other dynamic semantics

Besides our interest for the dynamics semantics of UML [103], we propose in this thesis to implement another semantics related to classical StateCharts [66], which adopts micro run-to-completion steps handling similarly internal and external events. Although such a policy may affect the data consistency within concurrent objects, there are still some justifications in support of its adoption for reactive systems rather than business applications.

As argued previously, when thoroughly favoring internal events against external ones within the state machine of a same component, safety properties run the risk of violation in case we recursively handle internal events generated within the same step whilst some external events are still waiting in the pool although they are critical and time constrained. Moreover, performance of a transition triggered by an internal event could probably change the status of the involved component into new one within which it becomes unable to react to some important external stimuli. Such changes may prevent the system from reaching some suitable statuses, leading thus, to the violation of liveness properties.

To conclude, we propose in this thesis to translate UML state machines into modular Petri nets in a syntactic way as independent as possible of semantics considerations and then produce their behavioral graphs using two state exploration algorithms implementing respectively the following dynamic semantics:

- The semantics of classical StateCharts as implemented by STATEMATE tool.
- The semantics of UML state machines (largely inspired by that of RHAPSODY tool, the syntactic variations of which are eliminated thanks to the expansion phase).

Chapter 3

Formal Semantics of UML State Machines

3.1 Implementation model of state machines

In our approach, each state machine has to be translated into time Petri net component (TPN-component). Petri nets are high abstract formalisms which capture well synchronization and concurrency concepts [107]. Moreover, they are provided with formal semantics making them suitable enough for the verification of systems. However, as Petri nets lack modularity capabilities, we propose a component variant of time Petri nets [89] as *implementation* models for our active components designed by means of UML state machines (see Fig.3.1).

Among its internal constituents, any TPN-component possesses two kinds of interface points:

- Input points are transitions that depict the receipt over "public" gates of external events from other TPN-components requiring some services. Any transition may

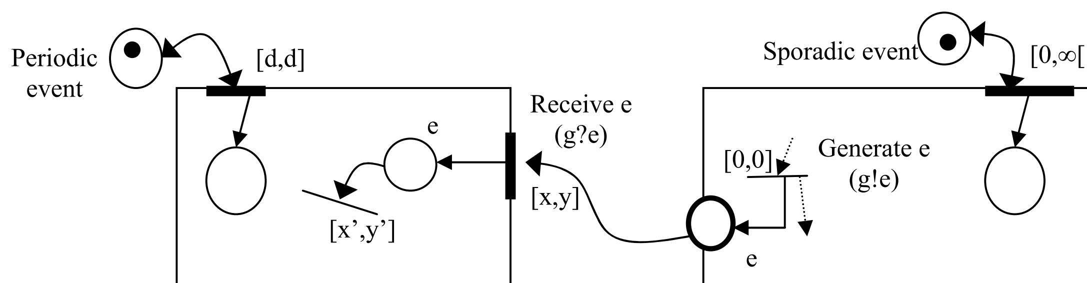


Figure 3.1: Component-based time Petri net.

be tagged with a time interval $[x, y]$ which expresses possible transmission delays of the incoming event.

- Output points are "event" places that collect events generated inside their owner TPN-component. These events are preserved for a time until they are conveyed and dispatched to other (server) TPN-components which have to provide some required services.

Besides incoming events from cooperating components, the environment can send various events to TPN-components of the system. All these events are depicted by tokens in marked free places. Whereas periodic events occur frequently at exact instants with a fixed periodicity, some events of sporadic kind occur at any moment of their time interval $[0, \infty[$. The occurrence of events is instantaneous (w.r.t. UML semantics) and this is modeled by the interval $[0, 0]$ labeling producer transitions. However, their transmission and dispatching may take some amount of time which is specified by a time value sampled from an interval $[x, y]$. Recall that any action which is synchronous or has no zero duration is divided into two instantaneous actions with a separating delay (c.f. Subsections 2.3.3 and 3.2.1).

Remark 1: Our enhanced Petri nets (TPNC) are related more to object-oriented Petri nets [125, 113, 80] rather than to modular Petri nets [22, 11]. Even if modular Petri nets, as presented in [22], allow designers to specify a system as communicating modules, these modules communicate using shared transitions or fusion places whereas TPNC do not use such shared entities. Each Petri net component is encapsulated and the only way of communication between components is the transmission of events. Object-oriented and hierarchical Petri nets [125, 113, 80] comply with these principles. However, these formalisms are intended to be used as modeling languages while we are proposing TPNC as a target (i.e., implementation) formalism for analysis purposes of StateCharts. That is why, the former Petri nets use syntactic constructs more complex than those used in our formalism.

3.1.1 TPN-Component syntax

The target model of the translation of any UML state machine is a Time Petri Net Component denoted by $TPNC$ (or TPN-Component) which is a component-oriented version of Time Petri Nets [89]. Formally, a marked $TPNC$ is a tuple: $R = \langle P, T, InOut, Pre, Post, Label_0, Label_1, Label_2, Priority, EVT, , M_I \rangle$ where:

- P is the set of places.

- T is the set of (instantaneous) transitions.

- $InOut \subseteq P \cup T$ contains the component interface (interaction points visible from outside).
- $Pre : T \longrightarrow 2^P$ is a backward incidence function giving input places (inplaces) of transitions.
- $Post : T \longrightarrow 2^P$ is a forward incidence function giving output places (outplaces) of transitions.
- $Label_0$, $Label_1$ and $Label_2$ are partial labeling functions:
  - $Label_0 : T \longrightarrow INT$, is a function that labels any $TPNC$ transition with a time interval where $INT = \{[d_1, d_2] \in \mathbb{R}^{>0} \times \mathbb{R}^{>0} \mid d_1 \leq d_2\}$ is the set of time intervals of positive real numbers.
  - $Label_1 : P \longrightarrow EVT$ where EVT is the set of events raised by the state machine transitions. The places labeled with events' names depict the event flow between the system components since these places are output places of transitions which raise their labeling events. At the same time, these are also input places of transitions triggered by their labeling events. On the other hand, any place without an event label depicts the control flow within a same component. For more precision, we draw event flow places as full grey circles whereas places depicting the control flow are drawn as white circles.
  - $Label_2 : T \longrightarrow \Sigma$ where Σ is the set of actions that are basic and instantaneous (cf. § 2.3.3). An action may be a sending or a receipt of a signal event or a method result, a method call, or an internal action coming from the split-up of any action of the related state machine (see Fig.2.4). It may also consist of a start or an end of an action (in the set ACT of the state machine actions), the completion of which takes non-zero duration.
- $Priority \subseteq T \times T$ is a priority (partial order) relationship between transitions in conflict. Hence, if $(t_1, t_2) \in Priority$ then whenever the two transitions t_1 (inner one) and t_2 (outer one) are simultaneously enabled the firing of t_1 should be preferred against that of t_2.
- $M_I : P \longrightarrow \mathbb{N}$ is the initial marking of the net (at instant 0) where $\mathbb{N}$ is the set of natural numbers. We use multi-sets as a means to represent markings to ease operations over them.
- Besides the aforementioned labeling functions, we can also tag some of the "*control flow*" places with names of internal activities similarly to the way that transi-

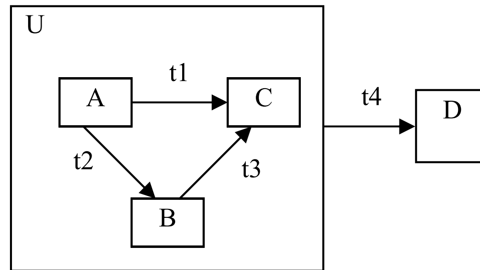


Figure 3.2: Priorities over transitions.

tions are labeled with names of actions. Indeed, since any "do-activity" is carried out while the control is in its encompassing state, then the "*control flow*" place representing this state will depict, once marked, the performance of its activity. However, it should be noted that transitions represent "instantaneous and uninterruptible" actions whereas places depict stable and durable statuses which may encompass internal activities. Once these places are exited (by removing their tokens) their activities will be simply stopped.

Before proceeding to the simulation phase (i.e., state exploration step), one would set the relationship $Priority$ according to the priority rule prescribed by the adopted semantics. Recall that priorities are needed only for handling simultaneous executions of conflicting transitions where two transitions are said in conflict if there is some common state that would be exited if any one was to be taken.

For example in Figure 3.2, t_1 and t_2 are in conflict because they would each imply exiting state A. Also, t_4 is in conflict with all of t_1, t_2 and t_3, since if t_4 is taken, the system must leave the state U and thus one of its substates. It follows that t_1 and t_4 cannot be taken in the same step.

These two kinds of conflicts are handled differently. In the first case, if the triggers of t_1 and t_2 are enabled at the beginning of a step, we get non-determinism since there is no reason to prefer one of them over the other. However, the situation is different when t_1 and t_4 are enabled in a same step. Here, according to the adopted priority policy, one of them is favored against the other one. In this context, priorities are exactly used to resolve the choice among such a class of conflicting transitions.

Accordingly, priorities in STATEMATE are determined outside-in and in RHAPSODY inside-out. Unfortunately, UML adopts RHAPSODY priority rule which transgresses some valuable concepts of design like abstraction and compositionality. For instance, at a high abstraction level, first the superstate U interface is designed, thereby asserting that outermost transitions (e.g. t_4) are taken whenever they are enabled. But, once we proceed to the detailed design of U, its nested structure may violate its

certified interface due to the RHAPSODY priority rule which favors internal transitions against external ones.

3.1.2 TPN-Component semantics

A transition "t" is enabled within a marking M if the required tokens of inplaces are available: $\forall p \in Pre(t) : M(p) \geq 1$ (because arcs incoming to transitions are all weighted with 1). Any enabled transition becomes fireable when it remains enabled for an amount of time at least equal to the minimal delay of its time interval. But, it should not exceed its maximal delay. Henceforth, the default intervals $[0, 0]$ are omitted in next figures for more readability.

When t fires, a new marking M' is instantly produced as follows: $M' = (M \setminus Pre(t)) \cup Post(t)$.

The priorities between transitions aim at setting in order simultaneous executions of several boundary-crossing arcs originating from transitively nested substates of a same superstate. According to [101, 103] the priority level decreases from innermost substates down to outermost ones such that when triggered at a time, exit actions from the former substates execute before those of the latter ones. Recall that there are two kinds of places: some ones support the control flow (white circles) whilst others support the event flow between parts of a Petri net (full grey circles).

In each Petri net unit, we distinguish *entry* and *exit* places among control flow places, which are respectively the initial place receiving control flow from a previous unit and the final place forwarding the control to a following Petri net unit. So, these particular places provide links to connect together subnets relating to substates of any composite state and depict well the control flow between them. Moreover, each action raising an event will be mapped into a transition with an event outplace which is labeled with that event name and which has to be linked somehow to any net transition triggered by this specific event.

3.2 Mapping state machines into modular time Petri nets

3.2.1 Translation patterns

We give next the key patterns about various cases of the translation. For more clarity, we use the term "transition arcs" to refer to UML transitions in contrast to Petri net (PN) transitions.

Transition arc between two simple states

Given one transition arc between two simple states S_1 and S_2 belonging to the same containing state, we distinguish four cases requiring different treatments depending on the kind of the action labeling this transition:

1. The action is asynchronous without duration; the triggered event is a signal event,
2. The action is asynchronous with duration; the triggered event is a signal event,
3. The action is synchronous; the triggered event is a method call event,
4. The special case of an action triggered by a time event.

Case 1: asynchronous action without duration. The state machine arc is mapped into a net component as follows (see Fig.3.3(a)): The source state S_1 is translated into an inplace of the transition *act* depicting the incoming control flow and the target state S_2 is related to an outplace depicting the outgoing control flow. For the trigger event of the transition *act*, we generate another event inplace that depicts the event flow. Itself is an outplace of another transition modeling the receipt of the triggering event e_1 which would have been raised elsewhere in the system.

The action *act* is mapped into a contingent sequence consisting of three successive Petri net transitions respectively labeled by: $exitS_1, act, entryS_2$. These three transitions should fire both as one indivisible unit since *act* is an asynchronous action which does not require any external cooperation. In the sequel of the chapter, such an uninterruptible sequence of transitions will be referred to as a transaction[1] (and illustrated in figures by dashed rectangles). It is obvious that the action *act*, itself, may generate some event e_2 (i.e., $e_2 = Gen(act)$) used by other transitions somewhere else. We hence should link later the event e_2 outplace to all event inplaces labeled with e_2 via receipt transitions.

Furthermore, the *abort* transition models the possibility to discard the event e_1 if the state machine is not yet in S_1. Since any transition must fire whenever it becomes enabled and no other conflicting transitions (except *abort* ones) are simultaneously enabled, we must make the priority of the transition *act* greater than that of the abort transition.

Moreover, we can label any PN-transition with a time interval. For instance, an interval labeling the receiving transition ($e_1?$) models the minimal and maximal time delays between the generation and dispatching instants of the event e_1. The variable

[1]The atomicity of a transaction is related to the run-to-completion steps and not to the thread preemption.

transmission delays depend on properties of the communication medium and other factors. For instance, the possibility of event loss can be depicted by an interval $[x, \infty[$ where x denotes the least delay to deliver that event for processing. Alternatively, we model untimed actions by non delayed transitions using the interval $[0, 0]$ so that once their trigger events are received, they will fire immediately.

Remark2: The appropriate weight N depends on the nature of the event instance that the transition generates. In other words, it depends on the number of its potential consumers. Hence if the raised event is a method call, then N should be equal to 1. However, if the event is a signal event, the number of consumers may be greater than 1. For instance, the weight N of the TPN arc joining the transition *act* with event e_2 outplace may be ω ($\omega \notin \mathbb{N}$ is a great number such that $\omega - 1 = \omega$) so that it is possible to broadcast the signal e_2 to each component using it. Though this raises a problem when this event is received more than once by some consumer component containing a loop of transitions, a simple solution may consist in adding to the receiver transition a new input place which is initially marked with a number N' of tokens equal to the times the consumer component is supposed to receive this event (in other words, the number of concurrent transitions receiving this event). It should be noted that, even if the component contains many concurrent transitions receiving the same event, all of them are dependent and should fire as a unique transaction (i.e., uninterruptible sequence of transitions) because the component actually receives this event once.

Case 2: asynchronous action with duration. The state machine arc is mapped into a net component as follows (see Fig.3.3(b)): The previous scheme is also applied to this case. However, the asynchronous action with duration should be split up into two actions (*start* and *end*) whose time annotations make them fire at two different instants separated with the given duration. So, we have two transactions: the first one consists of exiting the previous state S_1 and starting the action and the second one consists of finishing the action and entering the target state S_2. The time interval $[d_1, d_1]$ related to the first action shows the start instant and the interval $[d_2, d_2]$ of the second one represents execution duration of *act* (refer to Section 3.4 for more explanation).

Case 3: synchronous action. The state machine arc is mapped into a net component in a same way as in case 2 (see Fig.3.3(c)). The *start* action is substituted by the method call and the *end* action is the one receiving the result of the invoked method. However, the second transition here should have a triggered event which is the method response.

We will no longer show exit and entry transitions unless (this is necessary) to show resulting Petri nets in readable figures and focus on key constructs for the handled case.

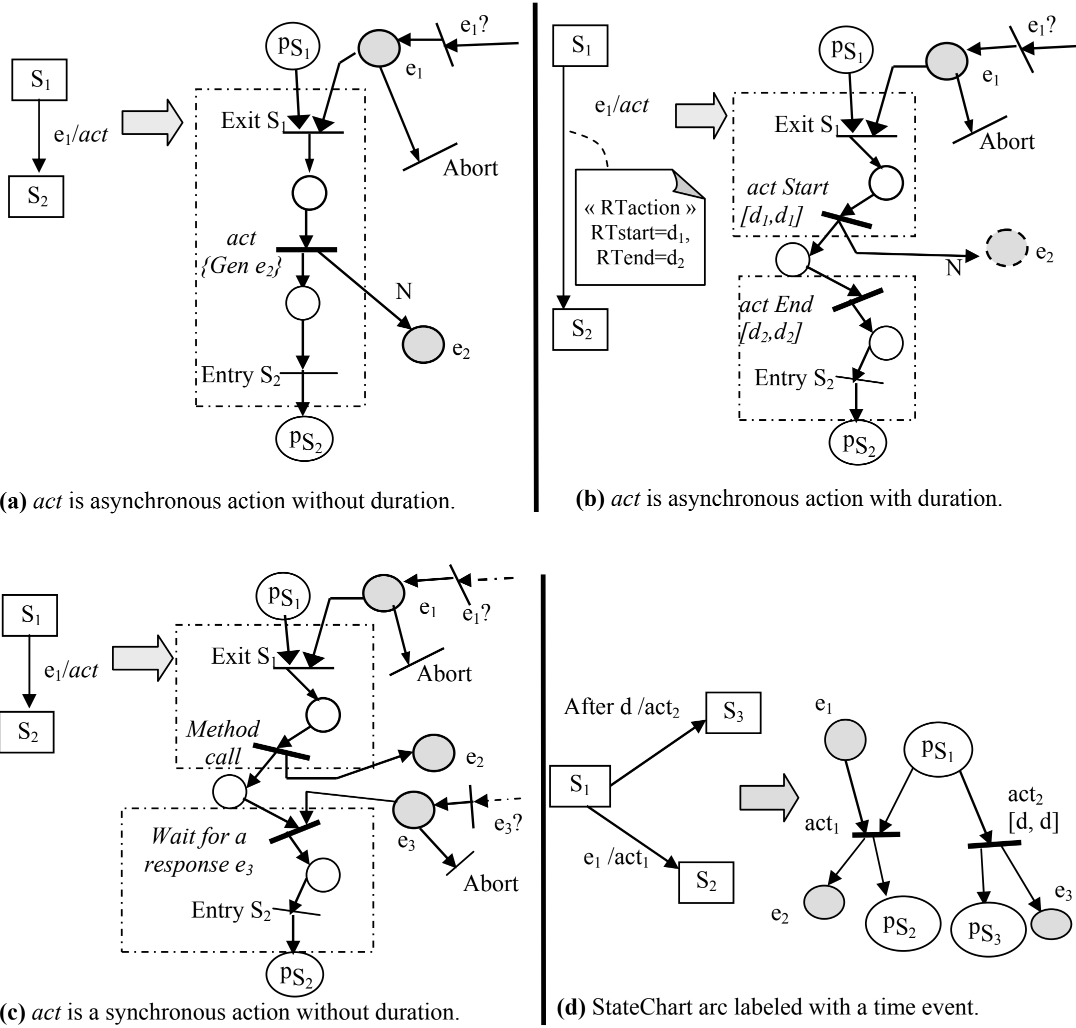


Figure 3.3: Transition arc between two simple states.

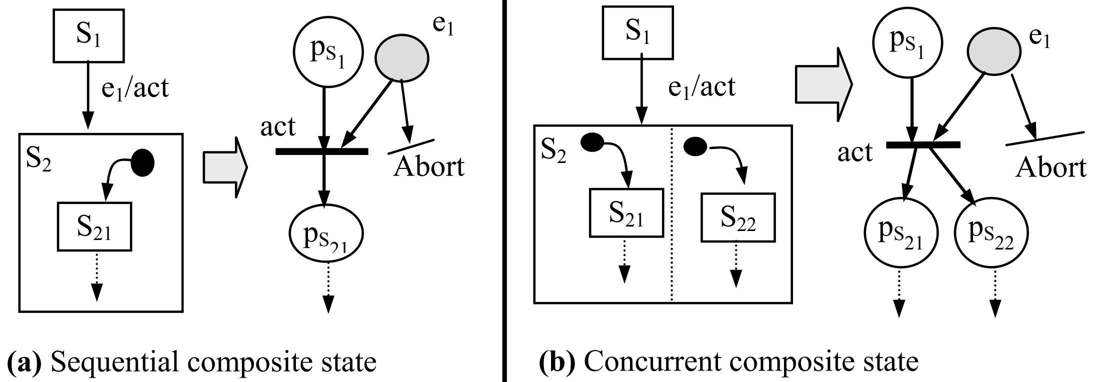


Figure 3.4: Transition arc from a simple state to a composite state without a history tag.

Case 4: action triggered by a time event. A time event "*after d*" models the expiration of a given deadline that may be absolute or relative to the time of entry into the source state of the transition triggered by the time event. The translation of a time event arc into a net transition consists in linking the control flow inplace p_{S_1} to the transition labeled with the exception action act_2 (see Fig.3.3(d)). This transition is labeled with the interval $[d, d]$ which means that if act_1 does not fire within this delay, a "timeout" occurs, firing hence the transition act_2.

Transition arc from a simple state to a composite state without a history tag

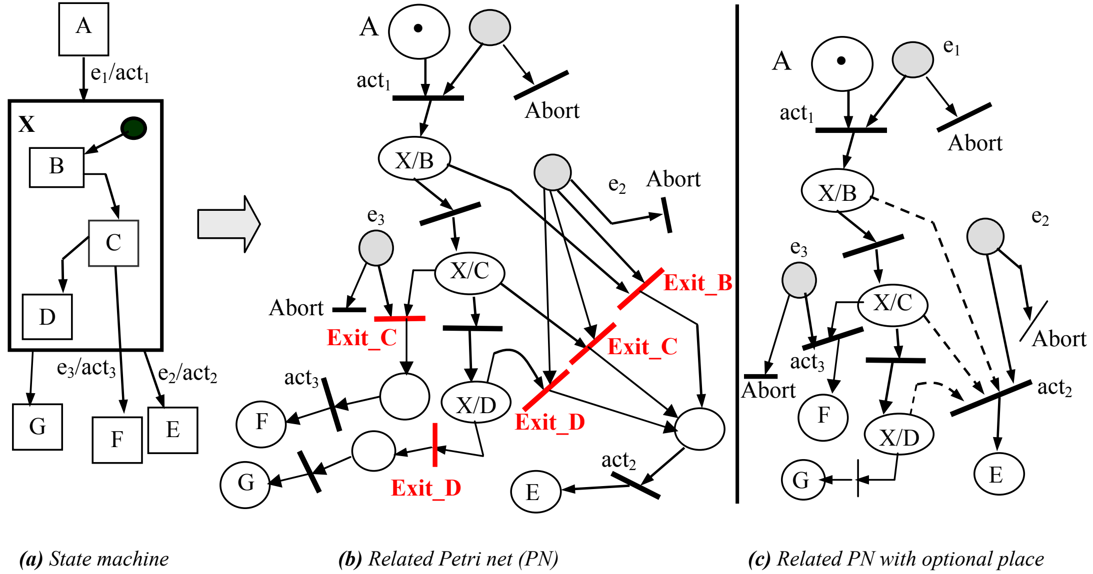
Once the trigger event e_1 is dispatched (see Fig.3.4(a)), the state machine processes the transition action *act* as soon as possible if the control is in the source state S_1. Otherwise, this event e_1 is discarded. As the arc tagged with e_1 goes to the edge of the composite state S_2, the outplace of the *act* transition is the entry place of the subnet related to S_2 if this one is sequential (Fig.3.4(a)).

But if S_2 is a concurrent state (see Fig.3.4(b)), all entry places relating to respectively initial states of concurrent regions of S_2 will be outplaces of the *act* transition. Hence, when firing this transition, a token is produced in the entry place of each region of the state S_2, activating thus all those concurrent regions at the same time.

Transition arc from a composite state without history tag

In Figure 3.5(a) there are three kinds of transition arcs exiting from the state X:

- The simple one is a triggerless transition outgoing to the state G. In this case,



(a) State machine *(b) Related Petri net (PN)* *(c) Related PN with optional place*

Figure 3.5: Source state of an arc is a sequential composite state.

we connect the (exit places of) subnet relating to the final substate D of X with the (entry places of) subnet relating to the target state G (the subnets of simple states are mostly single places).

- The second kind is a high-level transition outgoing from the edge of X to the state E. In this case, we have to connect each subnet relating to one direct substate of X with the entry place of E via a net transition "act_2" triggered by the same event e_2 (see Fig.3.5(b)). For more convenience, the transition arc "e_2/act_2" can be mapped into just one net transition for all X substates without exit actions $\{X/B, X/C, X/D\}$ whose related places are taken as optional inplaces of this net transition (see Fig.3.5(c)). This feature is illustrated by dashed lines joining the optional inplaces $\{B, C, D\}$ to the transition act_2 which becomes enabled if at least one of these inplaces is marked (for more details on optional places, see the last case handled in Subsection 3.2.1).

- The third transition is an exit arc labeled with act_3 and outgoing from only one direct substate C to the new target state F. So, we have to connect only the subnet of C with the subnet relating to F. Note that when exiting one region to somewhere outside its concurrent composite state, the other orthogonal regions of X should also be disabled (see Subsection 3.2.2 for more details).

Transition arc from a simple state to a composite state with a history tag

In Figure 3.6(a), the state S_2 is a sequential composite state with history tag. If this state is exited and later, reentered, the system must return to its most recent active configuration; that is the state configuration that was active when the composite state was last exited. Therefore, we use a "*thread*" place p_H as entry place to model control passing between substates of S_2.

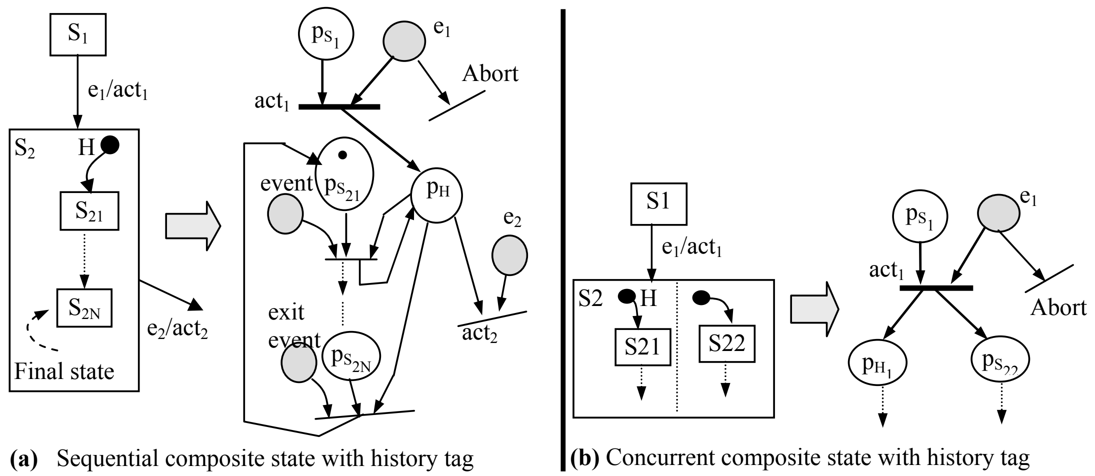
Notice now that the entry place p_H is not marked at the start. However, during the net execution it may receive a token from a previous transition so that it activates S_2. On the other hand, the initial place $p_{S_{21}}$ relating to the first substate S_{21} will be initially marked. When firing some internal transitions in S_2, the token of $p_{S_{21}}$ moves to the next places relating to substates of S_2 making it possible to find out every time the current marked place, and as a result, the related active substate.

Now, if the event e_2 occurs then the system will leave S_2 (removing definitely p_H token) whilst a token is still in an intermediate place. If the system returns to S_2 (renewing a token in p_H), the recent active configuration will be restored because the midway place remains marked until the control of S_2 is resumed. Conversely, when the (triggerless) exit transition fires from the final state, the p_H token is consumed and a new token is produced in the initial place $p_{S_{21}}$. In Figure 3.6(b), S_2 is a concurrent composite state whose one of the two orthogonal regions is labeled with the history tag. Here, we cope with each region according to the suitable method among those we have presented above. Accordingly, the firing of act_1 transition produces a token in the exact entry place of each region; whenever the region is history tagged we consider its thread place p_H as entry place. Otherwise, the initial place related to the first substate of that region is taken as an entry place.

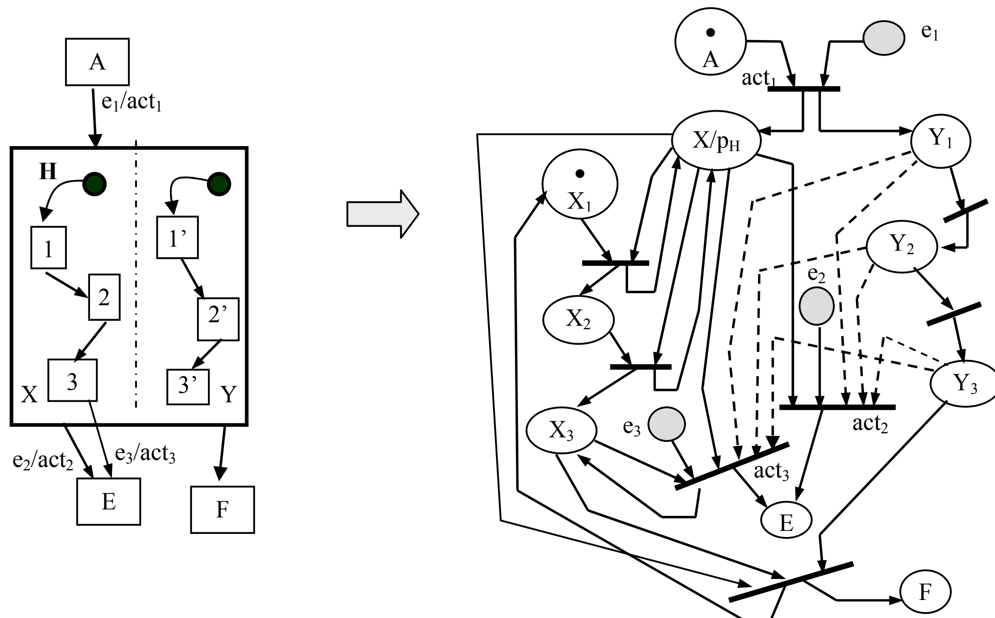
Transition arc from a composite state with a history tag

To shorten the ensuing Petri net in Figure 3.6(c), we do not add abort transitions modeling event removals. The dashed lines mean that the transitions act_2 and act_3 have to be replicated, once for each optional inplace Y_1, Y_2 and Y_3. Recall that exit transitions of orthogonal regions must synchronize. So, the exit arc which is event triggerless requires joining the final place of each region to that transition. In addition we join X/p_H to the exit transition, the firing of which renews a token only in the initial place X_1 of the history tagged region.

If the exit transition is a compound one (i.e. e_2/act_2) outgoing from the edge of a composite state, then we use, as classical inplace, the thread place of the history tagged region and, as optional inplaces, all the places of the other orthogonal region without history tag.



(a) Sequential composite state with history tag **(b)** Concurrent composite state with history tag



(c) Concurrent composite state with history tag

Figure 3.6: Composite state with a history tag.

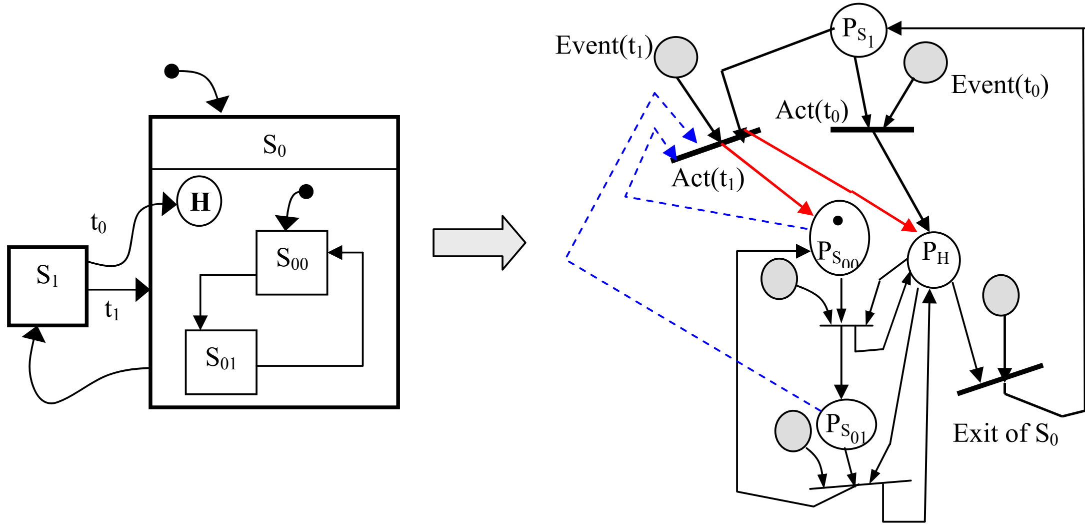


Figure 3.7: Composite state with separate initial and history vertices.

Composite state with separate initial and history vertices

In order to make the translation easier, we have so far taken the history vertex (wherever it exists) as the default entry vertex of the enclosing state. Such choice seems to us the most natural way to depict the behavior of a reactive component, the last exited state configuration of which, has to be restored when reentered again. Moreover, the reentrance via a distinct initial vertex inevitably erases this information and could thus lead to ambiguous behaviors of a history tagged component. Though such a design pattern is needed (see Fig.3.7), our approach can be easily extended to deal with a composite state having separate history and initial connectors in a conservative way as follows:

- The method still handles as explained previously the history entrance of the superstate S_0 (via the transition t_0),
- However, when we enter the state S_0 via the transition t_1, we have first to erase the last exited configuration by removing any token in intermediate control flow places. Then, we need to renew a token in initial places of the history tagged state.

To remove any token in intermediate places, we extend the Petri net with the notion of a partial function Opt that assigns to transitions their optional inplaces as follows: $PN = \langle R, Opt \rangle$ where R is the time Petri net (as defined in Subsection 3.1.1) and $Opt : T \longrightarrow 2^P$. A transition t is enabled from a marking M if $Pre(t) \subseteq M$ and

$\exists p \in Opt(t) : M(p) \geq 1$. But, once t is fired, it generates a new marking M' such that: $M' = M \setminus (Pre(t) \cup Opt(t)) \cup Post(t)$.

In other words, firing t needs the required tokens from its all classical inplaces and at least one of its optional inplaces. However, any tokens that could be available in further optional inplaces, shall be consumed, too. For instance, in Figure 3.7, t_1 becomes enabled when P_{S_1} receives a token. Thereafter t_1 fires consuming the control flow token from whichever the control flow place (either $P_{S_{00}}$ or $P_{S_{01}}$) is marked in the subnet related to S_0 because all of them are linked to t_1 as optional inplaces (via dashed lines). Note that firing of t_1 regenerates a token in both the first control place and the thread place to reset the configuration of S_0.

If the history vertex is a deep one, the same approach is recursively used to deal with inner history substates as usual by removing control flow tokens whichever the control place is marked thanks to the mechanism of optional input places, and renewing tokens in initial places (entry and thread inplaces).

Remark 3: Assume now that a boundary crossing transition t_2 exists from S_1 to the substate S_{01}. If the history vertex does not overwrite the initial connector, the reached configuration in this case is exactly (S_0, S_{01}) independently of the last exited active substate of S_0. Such a case is considered as ill-designed in view of its compositionality violation. Indeed, a deep history vertex specifies that its owner superstate has to store its configuration in order to be restored when reentered later, leading in so doing, to a predictable behavior. Anyhow, this case can also be handled within the above approach by means of optional places.

Remark 4: Even if the concept of "optional places" is not a standard notion of classical Petri nets it remains just a shorthand notation we use to represent some specific parts of the Petri net for more convenience (see Fig.3.8). Consequently, if one would use structural analysis tool of standard Petri nets, he has only to translate the enhanced Petri net ($TPNC$) to a standard Petri net by expanding the parts using the optional places (see Fig.3.8). However, we can just as well customize the firing rule according to this enhancement of Petri net. The exhaustive exploration of the state space of enhanced Petri nets is thus possible in order to use the analysis tools of marked graphs of standard Petri nets.

Remark 5: *deferred* events are discarded since we deal with safety-critical systems where events have to be handled as soon as dispatched. However, our approach can be easily adapted to tackle these events by simply removing their "*abort*" transitions. Hence, once such an event is dispatched while the control is not in the right state, the related *event flow* place will receive a token which remains until this state becomes active. Such a handling delay of an event would be achievable only if its "abort" transition had been removed from the subnet where the event is considered as deferred.

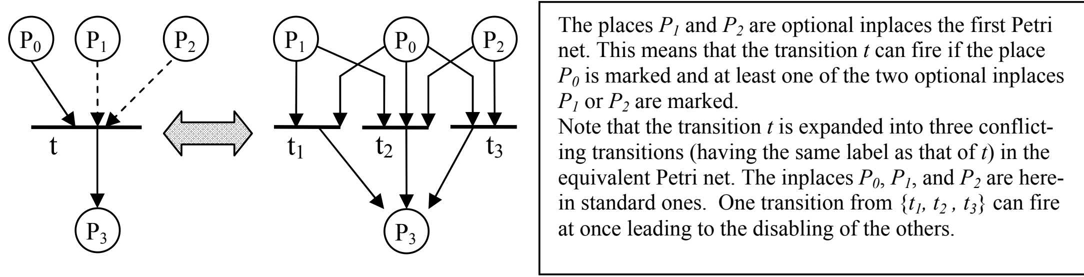


Figure 3.8: Expanding optional places into classical places.

3.2.2 Translation algorithms

As handling boundary-crossing arcs together with history vertices is too complex, we prefer to go by steps and defer their treatment to the second step of the translation.

Hence, during the first step of translation, we ignore the handling of boundary-crossing arcs that transgress the encapsulation concept of states. High-level transitions are as well boundary-crossing transitions which are handled in the same manner.

Once we obtain all subnets related to the state machine constituents, we combine them by means of a *parallel* operator such that every transition raising an event e, which has an out-place labeled with this event will be suitably connected to each event inplace whose label is e in other TPN-component (see the example depicted by figures 5.3 and 5.4 in SubSection 5.3.1). The link is just a receipt transition which we could additionally label with any time interval modeling dispatch delays of that event.

First step of the translation

Given some state machine D that consists of one topmost level state S_0, we ignore all boundary-crossing arcs and then we construct the related net as follows:

Let $S_1, \ldots, S_N$ be the direct substates of S_0 that may be connected together either sequentially or concurrently. First, we construct the subnet relating to each substate S_i (denoted by $[\![Si]\!]$) by calling recursively the same $Algorithm1$. Then, we use one of the two algorithms $Algo1 - Seq$ or $Algo1 - Par$ to combine together the resulting subnets $[\![Si]\!]$ by means of one of two connecting operators; the operator used may be either a parallel composition connector $||$ (if S_0 is a concurrent state) or a sequential composition $\triangleright$ (if S_0 is a sequential state).

If $Kind(S_0) = ConcurrentCompositeState$ then $[\![S_0]\!] = [\![S_1]\!] || \ldots || [\![S_N]\!]$.

If $Kind(S_0) = SequentialCompositeState$ then $[\![S_0]\!] = [\![S_1]\!] \triangleright \cdots \triangleright [\![S_N]\!]$.

The same approach is then employed recurrently to each substate whenever this one is a composite state. The sketch of $Algorithm1$ is a follows:

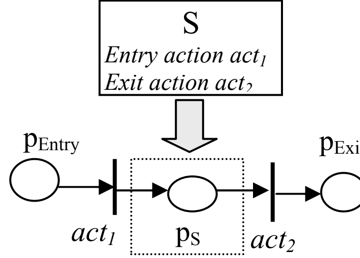


Figure 3.9: Handling in a simple state.

```
Algorithm Algorithm1
In: S: UML State Machine;
Out: [[S]]: TPNC;
Begin
 If S is a simple state
 Then Algo1-Simple-State (S, N)
 Else Begin
        Let Subnets − List := ∅; //subnets related to direct substates of S
        For each substate SSi of S
          Do Begin
                 Ni := Algorithm1(SSi);
                 Add(Subnets − List, Ni);
             End
        If Kind(S)=Sequential Then return Algo1-Seq(S,Subnets − List);
        If Kind(S)=Concurrent Then return Algo1-Par(S,Subnets − List);
        Link-Event-Places([[S]]);
      End
End Algorithm1.
```

Algo1-Simple -State handles only a simple state S by mapping it to one place p_S. If S does not have entry/exit actions, p_S becomes both an entry and an exit place of that subnet $[\![S]\!]$. Otherwise, it is translated into three places, p_{Entry}, p_S and p_{Exit} that are linked via entry/exit transitions as depicted in Figure 3.9. The same approach is also applied below where S is a composite state by substituting for p_S the subnet $[\![S]\!]$ related to this state S considered at first without its entry and exit actions. p_{Entry} is linked through the act_1 transition to the initial places of the subnet $[\![S]\!]$, namely those which would receive tokens to initiate the activity in that part of the system. Likewise, all final places of the subnet $[\![S]\!]$ have to be linked to p_{Exit} using the transition act_2. We give below the core of the subroutines $Algo1 - Seq$ and $Algo1 - Par$ in a verbose style:

```
Procedure Algo1-Seq (S: State Machine; Subnets − List: List of TPNC) returns TPNC
Begin
 For each arc Arc between SS_i and SS_j ∈ Substates(S)
  Do If SS_i is a simple state or Arc is triggerless Then
      Begin Let [[SS_i]] and [[SS_j]] be the subnets related resp. to SS_i and SS_j
```

```
      Switch (kind of Arc)
      Case (asynchronous action without duration):
         Add one transition t1 whose label is the name of Arc action;
         Join exit place of [[SSi]] to entry place of [[SSj]] via t1;
         Break;
      Case (asynchronous action with duration):
         Add a transition t0 labeled with Begin-action name of Arc;
         Add a transition t1 labeled with End-action name of Arc;
         Join the exit place of [[SSi]] to t0 as inplace;
         Add a middle place p as outplace of t0 and inplace of t1;
         Join t1 to the entry place of [[SSj]] as outplace;
         Break;
      Case (synchronous action):
         Add a transition t0 labeled with action name of Arc;
         Add a transition t1 labeled with Receive-Result of t0 action;
         Join the exit place of [[SSi]] to t0 as inplace;
         Add a middle place p as outplace of t0 and inplace of t1;
         Join t1 to the entry place of [[SSj]] as outplace;
         Break;
      End switch
      If the trigger event e exists Then create an event flow place whose label is e;
      If exit action exists in SSi
         Then add this event flow place as inplace of exit transition;
         Else add this event flow place as inplace of t1;
      End
  // assemble transitions in transactions as explained in Section 4.2.1
  If S is a history state Then add a thread place pH as a new entry place;
  Link pH at the same time as inplace and outplace respectively of initial and final transitions
of each (innermost) transaction of S;
  //Handle the entry and exit actions of S if they exist.
  If entry action of S exists Then
   Begin //the entry place of [[SS1]] is no longer initial place of [[S]]
      add a new entry place p of [[S]] ; // p is also initial place
      add an initial transition t labeled with the entry action of S;
      join p as inplace of t;
      If S is a history state
         Then join t to pH as outplace
         Else join t to the entry place of the first substate [[SS1]] ;
  End
  If exit action of S exists Then
      Begin // exit place of the last substate is no longer final place of S
         add a new final place p of [[S]] ;
         add an exit transition t labeled with the exit action of S;
         join exit place of the last substate [[SS1]] to t as inplace;
         join p as outplace of t;
         If S is history state Then
            Begin
               join pH only as inplace to t;
               add a token to the initial place of [[S]] ; //(≠ pH)
               join t to that initial place of [[SS1]] as outplace;
            End
      End
End Algo1-Seq.
```

Notice that if there is an exit action in a history tagged composite state S then we must add in $[\![S]\!]$ an exit transition to which p_H is joined only as inplace such that when t_{exit} fires it consumes the p_H token and disables, thus, any activity in $[\![S]\!]$. We should also reset this net by joining t_{exit} to the initial place of the first substate SS_1 of S (which may be the entry place of SS_1 if it has a history tag).

```
Procedure Algo1-Par (S: State Machine; Subnets − List: List of TPNC) returns TPNC
Begin
 Juxtapose together all regions nets ∈ Subnets-List;
 Add a fork place p_entry;
 Join p_entry to entry places of subnets through an entry transition t_entry;
 Label t_entry with the entry action name if it exits; Add a join place p_exit;
 Join p_exit to the final places of subnets through an exit transition t_exit;
 Label t_exit with the exit action name if it exits;
End Algo1-Par.
```

Once the different units of the Petri net related to a state machine are built, we ought to link them using the event places depicting the event flow between the parts of the state machine. In fact, the incoming event labeling any inplace, is just an event which some transition generates in an event outplace and the two places have thus to be joined via a receipt transition.

```
Procedure Link-Event-Places ([[S]] : List of TPNC) returns TPNC
Begin
 For all generated event outplace p labeled with e
    Do For all event inplace p' whose label is e
        Do Begin
              add a transition t labeled with "receive e";
              join p to t as inplace;
              join p' to t as outplace;
           End
End Link-Event-Places.
```

Second step of the translation

This step copes with boundary-crossing arcs, the triggering of which, causes the exit of related superstates starting from the innermost one in the active configuration. Any high-level transition belongs to this kind of arcs, too. Indeed, such a transition originates from the boundary of a composite state S, that is, it can occur whatever the substate the control is in. We expand any transition of that kind into a group of simple boundary-crossing arcs where each of them originates from one simple substate of the composite state.

For each boundary-crossing arc $BCArc$, the subroutine *Handle-Boundary-Crossing-Arc* generates a transaction t with an inplace labeled with the triggering event of $BCArc$. Next, it simply joins t to the entry place of the subnet related to the target

state of $BCArc$. However, it is quite difficult to cope with the source state (S) of $BCArc$. At first, we join its related place p_s to t as inplace, in such a way that $[\![S]\!]$ will be disabled whenever $BCArc$ occurs. But, this is not sufficient since the labels list of t has to contain names of all exit actions of S superstates, followed by the name of the action on $BCArc$. Next, we carry on with the handling of that boundary crossing arc through two phases to deal with upper composite states recursively until we reach the one which is the ancestor of the target state of $BCArc$.

Phase 1: we focus on the innermost composite states containing the simple state that is the source of the boundary crossing arc $BCArc$. We discern two cases according to whether the upper composite state is history tagged or not. These two cases are illustrated, respectively, by Figures 3.10(a) and 3.10(b) where thick arrows depict the transfer of control (i.e., a token) due to $BCArc$ from some place in the subnet related to the source composite state to outer places:

Case 1: all composite states of the ($BCArc$) source state are not history tagged (see Fig.3.10(a)). They have only to be exited by means of exit transitions we add to the $BCArc$ transaction t. Indeed, "t" is an uninterruptible sequence of *exit* transitions since exiting a substate using a boundary-crossing arc leads "at the same time" to exiting of all its enclosing superstates.

Case 2: the innermost composite states are not history tagged but the outermost one has history tag (see Fig.3.10(b)). When exiting the history tagged state, we have to renew a token in the initial place of its direct substate that either directly or transitively contains the source state of $BCArc$. So, when this history tagged state is reentered later (by getting a token in its thread place p_H), the control will be instantly resumed to its substate which was previously active.

Phase 2: the innermost composite state is history tagged enclosed in another one which is history tagged (see Fig.3.10(c)). We apply the same handling described above until we reach the composite state tagged with H_1. But, we apply a new handling of the outermost composite state which itself has a history tag H_2. The added exit transition related to the latter state should renew a token in the place p_{H1} so that when the control resumes to this upper state, we will be able to recover the last status reactivating its direct substate which is history tagged.

We give below a subroutine of which two loops implement respectively the two previous phases. The handling depends on whether superstates of the source state S have a history tag or not; when S is exited, the routine ensures that any superstate S' of S which has a superstate S" with history tag, becomes active once S" is reentered later. The main idea is to renew a token in the entry place of the innermost superstate S' (itself enclosed within a history outermost state S"). The entry place depends on whether S' is history tagged or not as explained above.

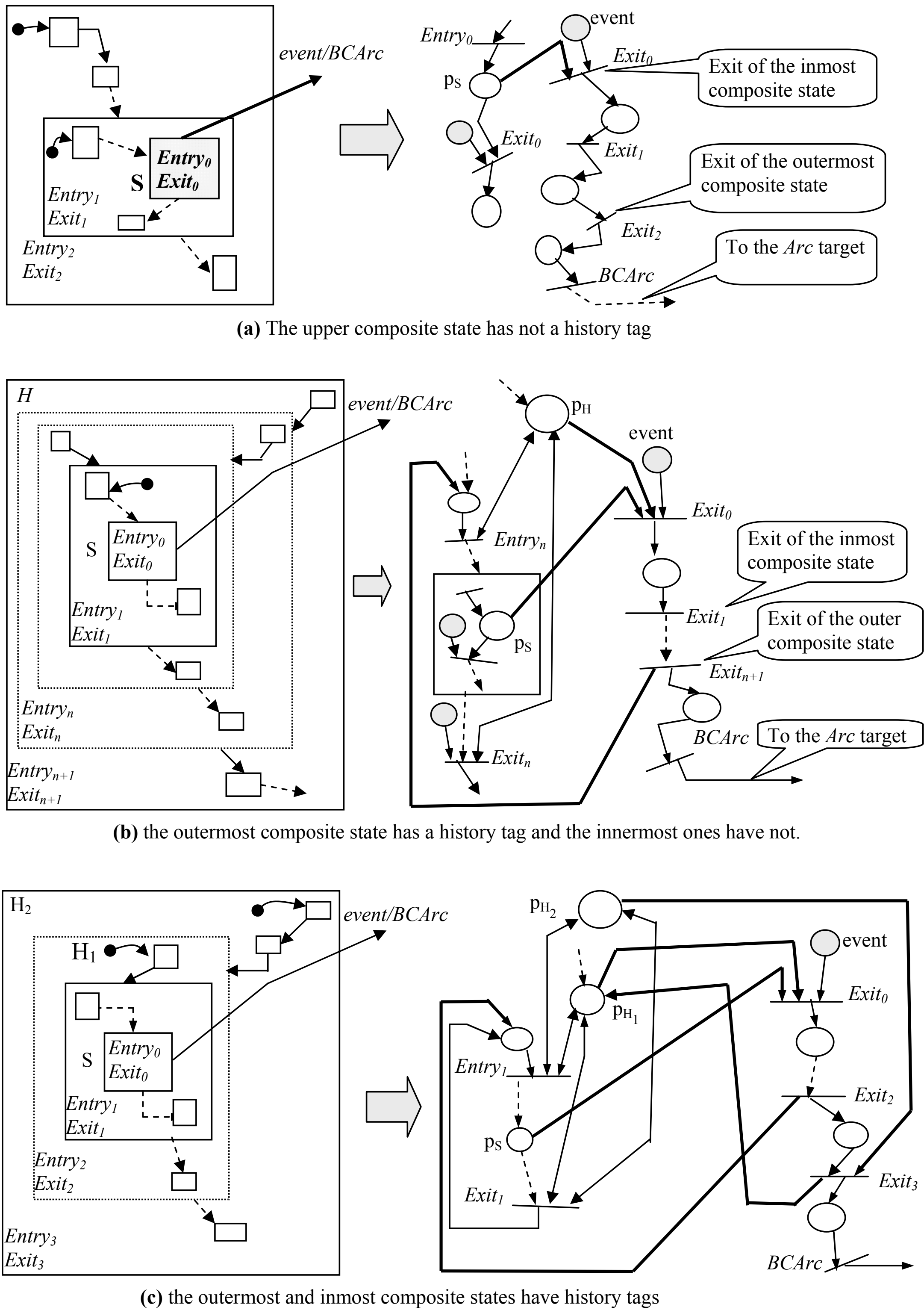


Figure 3.10: Handling of boundary crossing arcs.

```
Procedure Handle-Boundary-Crossing-Arc(⟦S0⟧ :INOUT TPNC; BCArc:IN State Machine Arc)
Begin
 Add a transaction t whose label at start is an empty set;
 Add an event flow event whose label is the trigger event of BCArc;
 S' := Source state of BCArc; // S' is a simple state
 S":= SuperState(S');
 While s' has not a history tag and S" does not contain the target state of BCArc
 Do Begin
      Add to Label(t) the exit action name of S';
      If S" has a history tag
          Then join t to the entry place of ⟦S'⟧ as outplace;
      S' := S";
      S":= SuperState(S");
    End
 While S" does not contain the target state of BCArc
  Do Begin
        Add to Label(t) the exit action name of S';
        If S' has history tag Then join the thread place of S' as its inplace;
        If S" has a history tag Then
            If S' has a history tag
              Then join t to the thread place of ⟦S'⟧ as outplace;
        S' := S";
        S":= SuperState(S");
     End
    // Hanling the target state of BcArc
 Let S be the target state of BCArc;
 Join the transaction t to the entry place of ⟦S⟧ as outplace;
 S' := S;
 S":= SuperState(S);
 While S" does not contain the source state of BCArc
   Do Begin
         If S" has a history tag Then
            For all ss ∈ C(S") such that ss ≠ S' //region not enclosing S
                Do join t to the entry place of ⟦ss⟧ as outplace;
            S' := S";
            S":= SuperState(S");
      End
End Handle-Boundary-Crossing-Arc.
```

Recall that an arc may enter any state either via its explicit entry or its history pseudo-state (if any). In case the arc enters a substate crossing the boundary of its composite states, we do not need to add any further handling when enclosing superstates (until reaching the least common ancestor of source and target states of the boundary-crossing arc) are sequential composite states. However, if one of the enclosing superstates is concurrent then each orthogonal region has to be entered via its default entry pseudo-state or via its history pseudo-state when the default one does not exist. These two cases are similarly handled since we use a common concept "entry place" to depict both of the two pseudo-states in our produced Petri subnets.

Exit of orthogonal regions

Whenever one region of a concurrent state is exited by a boundary-crossing arc, its orthogonal region should also be exited.

The solution we adopt consists in adding, as *optional* inplaces, some suitable places of the outermost orthogonal region to the exit transition related to $BCArc$. In other words, the enabling of this transition depends only on its classical inplaces. But, when it fires, tokens of both classical and optional inplaces will be consumed wherever the latter ones are available. Hence, even if an optional inplace is not marked, the enabled transition could fire. Therefore, instead of considering all places related to history-tagged composite states inside the concurrent region, we choose among them only thread places so that when the exit transition fires (due to the boundary-crossing arc) while the active configuration encloses any of those history-tagged states then it will disable it by removing its thread places without losing its last active configuration. Unfortunately, to handle no-history-tagged composite states in-side the orthogonal region, all of their places related to their substates have to be taken as optional inplaces of the exit transition whose firing disables them wherever the control is in.

```
Procedure Exit-Concurrent-Region ([[S]] :  INOUT TPNC; t:  IN TPNC arc)
Begin
 For all ss ∈ C(S)
    Do Begin
          Switch (Kind(ss)) // Kind of ss
          Case Simple State:  Begin
                                 take P_ss as optional inplace of t;
                                 add the exit action of ss to Label(t)
                              End
          Case concurrent State:  For all R ∈ C(ss) // Regions of ss
                                     Do Exit-Concurrent-Region (R,t)
          Case Sequential State:  If Tag(ss) is History
                                   Then Begin
                                           take P_H as optional inplace of t;
                                           add the exit action of ss to Label(t);
                                        End
                                   Else For all R ∈ C(ss) // substates of ss
                                           Do Exit-Concurrent-Region (R,t);
          End Switch
      End
End Exit-Concurrent-Region.
```

3.3 Simulation phase

3.3.1 Why using asynchronous transition systems?

On comparing the two diagrams of Figure 3.11 (considered at an abstract level closer to that of transition systems more than to StateCharts) in respect of the interleaving

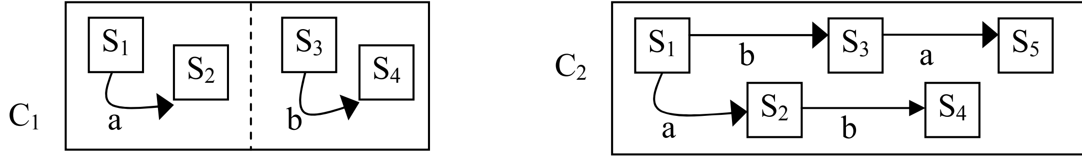


Figure 3.11: Interleaving vs. true concurrency semantics.

semantics, we can state that these are behaviorally equivalent (e.g., modulo the strong bisimulation [91]). Now, let us reconsider the comparison in the case a is a synchronous action. Although the execution of the first diagram may be blocked on a, it remains able to perform b once it receives its trigger event since this has to be done by an orthogonal region. However this scenario never happens in the second state machine. Therefore, a and b are considered independent in the first diagram whereas they are causally related in the second diagram.

In order to take into account such possibilities, we have to use behavioral models which depict enough the concurrency between independent actions such as asynchronous transition systems. Indeed these models suitably embed marking graphs and are discriminatory enough to distinguish the two schemes above (see Fig.3.11). We recall below the formal definition of the asynchronous transition system (ATS) [31].

Definition 1 *An asynchronous transition system is a quintuple $G = \langle Q, \hookrightarrow, \Sigma, q_0, I\rangle$ where*

- $\langle Q, \hookrightarrow, \Sigma, q_0\rangle$ *is a labeled transition system,*
- $I \subseteq \Sigma \times \Sigma$ *is a symmetric irreflexive relation called independence relation such that,*

1. $e \in \Sigma \implies \exists q, q' \in Q, (q, e, q') \in\hookrightarrow$
2. $(q, e, q_1) \in\hookrightarrow \wedge (q, e, q_2) \in\hookrightarrow \implies q_1 = q_2$
3. $e_1 I e_2 \wedge (q, e_1, q_1) \in\hookrightarrow \wedge (q, e_2, q_2) \in\hookrightarrow \implies \exists u \in Q, (q_1, e_2, u) \in\hookrightarrow \wedge (q_2, e_1, u) \in\hookrightarrow$
4. $e_1 I e_2 \wedge (q, e_1, q_1) \in\hookrightarrow \wedge (q_1, e_2, u) \in\hookrightarrow \implies \exists q_2 \in Q, (q, e_2, q_2) \in\hookrightarrow \wedge (q_2, e_1, u) \in\hookrightarrow$

Notation: we denote, henceforth, any transition (q, e, q') with $q \xrightarrow{e} q'$. The first rule of the independence relation "I" compels each event to be used as a label of at least of one transition whereas the second rule states the determinism property forcing transitions outgoing from a same source state to have distinct labels. The two last

rules define a diamond property asserting the independence relationship between e_1 and e_2.

Remark 6: Though reachability graphs are usually built as classical transition systems, we choose in our approach to enhance them with the concept of independence relation I to capture the extensive intra-components and inter-components concurrency. So, the state exploration process has to build the relation "I" to embed reachability graphs into asynchronous transitions systems which could be of interest for comparing state machines using variants of branching bisimulations that are adequate for true concurrency context (e.g., forth-back bisimulations [31]).

3.3.2 Modular generation of marking graph

The generation of the marking graph of a whole Petri net is achieved by combining the partial marking graphs of its TPN-components. This combination should handle two policies: the full concurrency between independent actions and the precedence relation between causally-related actions (emission-reception). Indeed, non-causally related events that belong to different components (or active objects) are assumed evolving concurrently because these components have their own threads along with their own events pools.

Once we generate partial marking graphs related to TPN-components, these graphs need to be suitably combined together in order to obtain the behavioral graph of the whole system. In a last phase, we add the time annotations as timing constraints on nodes and arcs of the graph. In our approach, passive objects are partitioned into different components, each of which owns its control thread with a unique event pool. The state diagram of a component consists of the collection of the cooperating state machines relating to its passive objects running under the control of its active object.

To generate a partial graph of any TPN-component, we need to add some pieces from its environment to be able to fire transitions in need of trigger events incoming from either cooperating components or the outer environment. In the latter case, albeit events are either sporadic or periodic, all of them are modeled by places once-marked and linked to their relating receipt transitions (see Fig.3.12). On the other hand, for each receipt transition of an event arriving from another component, we must also append its sending transition even if it is external. As explained later, this allows us to merge partial graphs of TPN-components to get the whole marking graph.

Generation of marking graph of one TPN-Component

Inside one state machine (modeling some component), we need to take care of the precedence relation between events' occurrences in order to comply with the run-

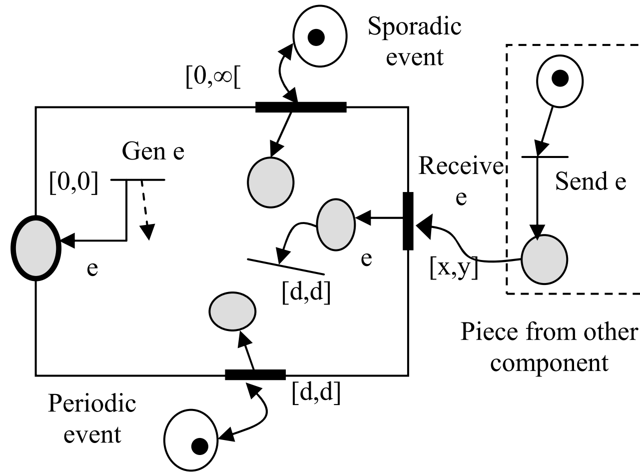


Figure 3.12: A TPN-Component.

to-completion constraint. The precedence relation certainly includes the precedence ordering between sending and reception actions of a same event thanks to the inherent causality relationships between the PN transitions that raise and consume these events. Consequently, in order to build the independence relation "I" during the state exploration process, we have to mainly find out non-intuitive causality relationships between events, the occurrences of which should belong to strict successive run-to-completion steps.

Among transitions of a TPN-Component (as illustrated in Fig.3.12), we discern three kinds:

- Computational transitions or transactions inside a TPN-component, referred to as "*internal act transitions*", depict the control flow, do some operations and thereby may raise events. In this fashion, transitions without triggering events belong to this set of transitions (denoted by $\mathcal{I}$).
- The other "internal" transitions are called "*internal receipt transitions*" because these are labeled with *event reception* actions, the firing of which will enable the previous act transitions (the set of such internal transitions is denoted by $\mathcal{T}_e$).
- Finally, "*external*" transitions (from pieces of other cooperating components) are added just to be able to produce events which trigger internal receipt transitions inside our component (the symbol $\mathcal{T}_s$ denotes the set of external transitions).

We give first the generation method with respect to the STATEMATE policy because it is simpler than the RHAPSODY one which needs more adaptation.

Remark 7: Recall that the STATEMATE policy considers all events (both internal and external) as equal. So, when dequeuing an event (whatever its kind), the triggered transitions in orthogonal regions (within one step) may raise internal events which will be queued and carried out within next steps. However, using the RHAPSODY policy a macro-step begins once an external event dequeued. The triggered transitions may raise internal events which need to be carried within the same step. So, as long as internal events exist in the event pool each one of them will lead to a micro-step. When all these internal events are handled, a new macro-step can then be launched by processing the next dequeued external event.

STATEMATE version of the simulation algorithm

At the beginning of each step, we compute the set $\mathcal{I}$ of enabled "*internal act*" transitions. Thereafter, we fire all these transitions in any order such that, within any interleaved path of them, we would never begin a new computational step until the set $\mathcal{I}$ becomes empty in order to comply with the run-to-completion policy of STATEMATE.

After producing each marking, fired transitions are evidently removed from $\mathcal{I}$ whereas new enabled ones are not added until the step is entirely finished. Recall that $\mathcal{I}$ is concerned with only internal transitions labeled with actions which generate signals or method invocations or carry out (abstract) operations. However, when the step ends (i.e. $\mathcal{I}$ becomes empty) we have first to compute the set $\mathcal{T}_e$ of new enabled "*internal receipt*" transitions, the firing of which will enable "internal act" transitions which would be included in $\mathcal{I}$.

Once $\mathcal{T}_e$ is built, all possible steps are concurrently explored by choosing and firing in a non deterministic way one transition from $\mathcal{T}_e$. The firing of any action from $\mathcal{T}_e$ triggers inside our considered component some "*internal act*" transitions which we immediately put into the set $\mathcal{I}$. Subsequently, *SimulationAlgorithm1*, given below, implements this approach by using a recursive function "*Explore1*" which explores the markings obtained from a given marking $\mathcal{M}$ by firing potential paths of interleaved transitions from $\mathcal{I}$ according to the run-to-completion rule (see Fig.3.13).

As illustrated by the bloc of lines 32-46 in the procedure "*Explore1*" (w.r.t. the case $\mathcal{I} \neq \emptyset$), the set $\mathcal{I}$ contains, at the beginning of a step, all enabled "*internal act*" transitions and it is updated continually after the firing of each transition until the set $\mathcal{I}$ becomes empty.

In this case, a new step starts (lines 2-31 w.r.t. $\mathcal{I} = \emptyset$) by computing the set $\mathcal{T}_e$ of "*internal receipt*" transitions. All possible branches are explored where each branch is initiated by firing one transition from $\mathcal{T}_e$ and then, recalculating the subsequent set

$\mathcal{I}$ of new enabled transitions. Recall that the ordering of firings of these transitions is arbitrary and the firing of a transition may disable other ones which are in conflict with it.

Lastly, notice that the firing of external transitions needs, always, to be made in a concurrent manner to the component progression as these are not under the control of its thread. Therefore, they may occur at any moment whatever the status the control is in. That is, at any reached marking and before firing internal transitions, we should randomly raise various external events by firing their relating transitions chosen in a non-deterministic way from their set $\mathcal{T}_s$. As the transitions of $\mathcal{T}_s$ belong to other threads, they may asynchronously fire at different instants, as well as equal instants. As a result, we have to explore every potential evolution inside our component due to the simultaneous firings of transitions of any subset of $\mathcal{T}_s$. This is illustrated by a loop exploring all possible evolutions, each of which is induced by the (concurrent) firing of all transitions of one of the subsets $E \in 2^{\mathcal{T}_s}$ through the procedure $Send-External-Events(M, E)$.

Similarly, when $\mathcal{I}$ and $\mathcal{T}_e$ are empty (bloc of lines 6-13 in "*Explore1*"), we must fire a number of transitions from $\mathcal{T}_s$ to enable some internal receipt transitions of $\mathcal{T}_e$ and thus initiate a new step. As expected, all potential evolutions owing to subsets $E \in 2^{\mathcal{T}_s}$ are concurrently explored.

Remark 8: Following what has been pointed out in case 1 of the first translation pattern in Subsection 3.2.1, we have to fire as a unique transaction the group of all enabled transitions in $\mathcal{T}_e$ which are related to the receipt of a same event since these transitions depict the receipt of a same event by one component.

Notations: The marking graph $G = \langle \mathfrak{M}, PNT, \mathcal{M}_0 \rangle$ where $\mathfrak{M}$ is the set of reached markings, $\mathcal{M}_0 \in \mathfrak{M}$ is the initial marking and the set $PNT \subseteq \mathfrak{M} \times \mathfrak{M}$ ("$\times$" is the Cartesian product) includes any potential transition (i.e., a computational step) from a reached marking to another one through the execution of an action labeling a Petri net transition that has been fired at this step. Furthermore, the notation $\mathcal{M}[t >$ means that the transition t is enabled at the marking $\mathcal{M}$.

```
Algorithm SimulationAlgorithm1
In: PN: Temporal Petri Net-Component; M0: initial marking;
Out: G: marking graph;
Begin
  𝔐 := {M0}; // explored Markings
  PNT := ∅; // Arcs related to firings of PN Transitions
  I := {t | M[t > and t is internal act transition};
  Explore1(M0, I, ∅);
End.
```

The procedure "*Explore1*" begins a new run-to-completion step (at line 3) when

$\mathcal{I}$ becomes empty; that is, no transition from the previous step is still enabled. In this case, the procedure collects new "*internal receipt*" transitions (line 4) and adds them into the set $\mathcal{T}_e$ together with any previous ones which have not been handled yet. Thereafter, if $\mathcal{T}_e$ is not empty (lines 15-30) then the procedure has to select and fire randomly one transition from $\mathcal{T}_e$ modeling the reception and de-queuing of some trigger event.

When no event from $\mathcal{T}_e$ can be dispatched (line 05), then the procedure will try to find out and perform some external transitions from $\mathcal{T}_s$ (lines 06-13). Indeed, the firing of any of such transitions represents an event sending (from an external component) whose receipt (in $\mathcal{T}_e$) would allow the start of a new step in our main component. Once the set $\mathcal{T}_s$ is built, the procedure explores all potential evolutions of the component by selecting arbitrarily any subset of $\mathcal{T}_s$ and firing, then, its transitions in any order (thanks to function $Send - External - Events$). Thereafter, the procedure itself is recalled since the sending of external events may make some "*internal receipt*" transitions enabled. In this case (line 14) where $\mathcal{T}_e$ becomes not empty, a next receipt event is randomly dispatched (lines 21-28) due to delays and unreliability of transmission. Note that before this step, other components are always capable of sending their events to our main component. Such a situation is depicted by the bloc of lines (16-20) where each subset of $\mathcal{T}_s$ of external transitions is performed, allowing hence the component to execute "internal receipt" transitions of $\mathcal{T}_e$ (bloc of lines 21-28).

However, we can fire only one transition from $\mathcal{T}_e$ at a time (lines 23-24) because of the run-to-completion policy. Thereafter, we compute the set $\mathcal{I}$ of enabled transitions from the new marking $\mathcal{M}$ (line 25) and we recall the procedure to begin a new computational step. As the choice of the dispatched event is randomly made, we have consequently to explore all possible evolutions through the loop of lines (21-28).

Whenever the procedure is called with a set $\mathcal{I}$ not empty, it will directly carry out the bloc of lines (33-46). As usual, external transitions are concurrently fired (lines 34-36), followed by the loop of lines (36-45) which allows exploring all possible evolutions by choosing and firing one among the "*internal act*" transitions (lines 39-41). The firing of any transition leads to, respectively, the update of the set $\mathcal{I}$ of enabled "*internal act*" transitions and the recall of the procedure.

```
Procedure Explore1 (𝓜:  In Marking; 𝓘, 𝓣_e:  In set of transitions)
01:Begin
02: If 𝓘 = ∅ then /* new run-to-completion step */
03: Begin
04:     𝓣_e := {t | 𝓜[t >² and Label(t) is an event Receipt};
05:     If 𝓣_e = ∅ then
06:      Begin
07:       𝓣_s := {t | 𝓜[t > and Label(t) is an external event sending};
```

[2]The Notation $\mathcal{M}[t >$ means that the transition t is enabled at the marking $\mathcal{M}$.

```
08:        For all E ∈ 2^{T_s} do
09:         Begin
10:           M' := Send − External − Events(M, E);
11:           Explore1(M', I, T_e);
12:         End
13:       End
14:     Else
15:        Begin
16:           T_s := {t | M[t > and Label(t) is an external event sending};
17:           For all E ∈ 2^{T_s} do
18:             Begin
19:                M' := Send − External − Events(M, E);
20:                T_e := {t | M[t > and Label(t) is an event Receipt}; //Update T_e
21:                For all t ∈ T_e do
22:                  Begin
23:                    M" := Fire(M', t); // M' --t--> M"
24:                    𝔐 := 𝔐 ∪ {M"};
25:                    PNT := PNT ∪ {(M', Labels(t), M")};
26:                    I := {t | M"[t > and t is internal act transition};
27:                    Explore1(M", I, T_e \ {t});
28:                  End
29:              End
30:        End
31: End
32:   Else
33:        Begin /* case I ≠ ∅*/
34:          T_s := {t | M[t > and Label(t) is an external event sending};
35:          For all E ∈ 2^{T_s} do
36:            Begin M' := Send − External − Events(M, E);
37:                   For all t ∈ I such that ∀t' ∈ I : (t', t) ∉ Priority do
38:                     Begin
39:                       M" := Fire(M', t); // M' --t--> M"
40:                       𝔐 := 𝔐 ∪ {M"};
41:                       PNT := PNT ∪ {(M', Labels(t), M")} ;
42:                       I := I \ ({t} ∪ {t' |° t ∩° t' ≠ ∅})^3;
43:                       Explore1(M", I, T_e);
44:                     End
45:            End
46:        End
47:End Explore1.
```

The function Send-External-Events, with the same inputs, returns at the last iteration, the same final marking although distinct paths of interleaved transitions of S are, actually, executed because transitions of S are pair-wise independent.

```
Function Send-External-Events (M:Marking; E:set of transitions) returns Marking
Begin
   If E = ∅ then return M
     Else
      For all transition t ∈ E do
       Begin
```

[3]Shorthand notations: $t^\circ = Post(t)$ and $^\circ t' = Pre(t')$.

```
            M' := Fire(M, t); // M --t--> M'
            𝔐 := 𝔐 ∪ {M'};
            PNT := PNT ∪ {(M, Labels(t), M')};
            return Send − External − Events(M', E \ {t});
        End
End Send-External-Events.
```

Remark 9: Note that the priority rule is taken into account thanks to $Priority$ relationship built according to the adopted rule priority during translation phases. So, in the third bloc of $SimulationAlgorithm1$, the fireable transitions (from the set $\mathcal{I}$) are only transitions $t \in \mathcal{I}$ such that $\forall t' \in \mathcal{I}, (t, t') \notin Priority$. This means that if there are two (internal act) transitions in conflict, say t and t' , and if t' has more priority than t, then t can never fire. Indeed, once t' fires, it disables all conflicting transitions including t. Note that external sending events and internal receipt events are not concerned with $Priority$ relationship since they occur somewhat in a concurrent manner according to the run-to-completion policy.

Remark 10: Can we have a non-terminating run-to-completion step? Within the STATEMATE semantics, a run-to-completion step finishes once all enabled *internal act* transitions (stored in $\mathcal{I}$) are fired. Thereafter, a trigger event is dequeued and dispatched by firing any *internal receipt* transition from $\mathcal{T}_e$. A new set $\mathcal{I}$ is then computed. As the number of arcs in a state machine is finite, the cardinality of set $\mathcal{I}$ can only be finite and hence the run-to-completion step will certainly terminate once all transitions of $\mathcal{I}$ fire. Within the RHAPSODY semantics, before commencing on a run-to-completion macro-step all internal events generated throughout the previous step are handled during this step itself. We may have a non terminating macro-step only in one case: if there is some loop of transitions $(t_0 t_1 t_2 \ldots t_k t_0)$ inside a state machine such that the firing of t_0 generates the trigger of t_1 and the firing of t_1 generates the trigger of t_2 and so on until the firing of t_k which generates the trigger of t_0. However, such a case can be considered as abnormal and ill-designed model.

RHAPSODY updated version of SimulationAlgorithm1

Currently, internal events generated during a step should be handled within the same step (in respect of the UML semantics). Accordingly, we tailor $SimulationAlgorithm1$ to get the new $SimulationAlgorithm2$ by adding a new set $\mathcal{T}_i$ of enabled transitions depicting the reception of internal events, as well as a new bloc at the start of subroutine $Explore2$, for handling this set $\mathcal{T}_i$. Hence, every time the set $\mathcal{I}$ (of enabled internal act transitions) becomes empty, we compute the set $\mathcal{T}_i$ and then check whether it is empty. If there are yet some internal events which have been generated in previous micro-steps then we initiate a new micro-step by choosing in a non-deterministic way a transition from the set $\mathcal{T}_i$ since the OMG specification does not say any thing about

it (though a fitting scheduling policy can be employed to meet some time deadlines).

Afterwards, the recursive call of "*Explore*2" with a new non-empty set $\mathcal{I}$ allows going to the third bloc (lines 60-75) where we fire all concurrent transitions until $\mathcal{I}$ becomes again empty. The interleaved paths start from the same marking and attain at last also a same marking so that this part of the graph would be diamond shaped because all these actions are pair-wise independent. Moreover, to launch a new macro-step, we need to have no enabled transition in $\mathcal{T}_i$. This implies that $\mathcal{I}$ remains empty, which means that the previous macro-stop has completely finished. Consequently, the second bloc (lines 30-59) of "*Explore*2" is entered to handle transitions from $\mathcal{T}_e$ specifically labeled with external events receipt actions. All paths are explored by choosing non-deterministically one transition from $\mathcal{T}_e$ and firing it in order to begin a new macro-step. Finally, external transitions of $\mathcal{T}_s$ always execute in a concurrent manner to all other transitions throughout all procedure blocs (see Fig.3.13).

```
Algorithm SimulationAlgorithm2
In: PN: Temporal Petri Net-Component; M_0: initial marking;
Out: G: marking graph Begin;
  𝔐 := {ℳ_0};
  PNT := ∅ ⊆ 𝔐 × Σ × 𝔐;
  ℐ := {t | ℳ[t > and t is internal act transition};
  Explore2(ℳ_0, ℐ, ∅, ∅);
end SimulationAlgorithm2.

    Procedure Explore2 (ℳ: Marking; ℐ, 𝒯_e, 𝒯_i: set of transitions)
01:Begin
02: If ℐ = ∅ then /* initiate a new run-to-completion micro-step */
03: Begin
   /* when a micro-step is finished (ℐ is empty), compute the set 𝒯_i
    of transitions modeling reception of internal trigger events.
    Next, select anyway and fire one transition from 𝒯_i.*/
04:   𝒯_i := {t | ℳ[t > and Label(t) is event Receipt and ∃p ∈° t | Label(p) is internal}
05:   If 𝒯_i = ∅ then
06:   Begin
07:       𝒯_s := {t | ℳ[t > and Label(t) is an external event sending};
08:       For all E ∈ 2^𝒯_s do
09:       Begin
10:          ℳ' := Send − anyway − External − Events(ℳ, E);
11:          Explore2(ℳ', ℐ, 𝒯_e, 𝒯_i);
12:       End
13:   End
14: Else
15:   Begin
16:      𝒯_s := {t | M[t > and Label(t) is an external event sending};
17:      For all E ∈ 2^𝒯_s do
18:      Begin
19:        ℳ' := Send − anyway − External − Events(ℳ, E);
20:        For all t ∈ 𝒯_i do
21:          Begin
22:            ℳ" := Fire(ℳ', t); // ℳ' --t--> ℳ"
```

23: $\mathfrak{M} := \mathfrak{M} \cup \{\mathcal{M}"\}$; $PNT := PNT \cup \{(\mathcal{M}', Labels(t), \mathcal{M}")\}$;
24: $\mathcal{I} := \{t \mid \mathcal{M}"[t >$ and $Label(t)$ is not an event Receipt$\}$;
25: $Explore2(\mathcal{M}", \mathcal{I}, \mathcal{T}_e, \mathcal{T}_i \setminus \{t\})$;
26: End
27: End
28: End
29: End

30: If $\mathcal{I} = \emptyset$ and $\mathcal{T}_i = \emptyset$ then /* new run-to-completion macro-step */
31: Begin
32: $\mathcal{T}_e := \{t \mid \mathcal{M}[t >$ and $Label(t)$ is an event Receipt$\}$;
33: If $\mathcal{T}_e = \emptyset$ then
34: Begin
35: $\mathcal{T}_s := \{t \mid \mathcal{M}[t >$ and $Label(t)$ is an external event sending$\}$;
36: For all $E \in 2^{\mathcal{T}_s}$ do
37: Begin
38: $\mathcal{M}' := Send - anyway - External - Events(\mathcal{M}, E)$;
39: $Explore2(\mathcal{M}', \mathcal{I}, \mathcal{T}_e)$;
40: End
41: End
42: Else
43: Begin
44: $\mathcal{T}_s := \{t \mid \mathcal{M}[t >$ and $Label(t)$ is an external event sending$\}$;
45: For all $E \in 2^{\mathcal{T}_s}$ do
46: Begin
47: $\mathcal{M}' := Send - anyway - External - Events(\mathcal{M}, E)$;
48: $\mathcal{T}_e := \{t \mid \mathcal{M}[t >$ and $Label(t)$ is an event Receipt$\}$;//Update $\mathcal{T}_e$
49: For all $t \in \mathcal{T}_e$ do
50: Begin
51: $\mathcal{M}" := Fire(\mathcal{M}', t)$; // $\mathcal{M}' \xrightarrow{t} \mathcal{M}"$
52: $\mathfrak{M} := \mathfrak{M} \cup \{\mathcal{M}"\}$;
53: $PNT := PNT \cup \{(\mathcal{M}', Labels(t), \mathcal{M}")\}$;
54: $\mathcal{I} := \{t \mid \mathcal{M}"[t >$ and t is an internal act transition$\}$;
55: $Explore2(\mathcal{M}", \mathcal{I}, \mathcal{T}_e \setminus \{t\}, \mathcal{T}_i)$;
56: End
57: End
58: End
59: End

60: For all $t \in \mathcal{I} \mid \forall t' \in \mathcal{I} : (t', t) \notin Priority$ do
61: Begin
62: $\mathcal{T}_s := \{t \mid \mathcal{M}[t >$ and $Label(t)$ is an external event sending$\}$;
63: For all $E \in 2^{\mathcal{T}_s}$ do
64: Begin
65: $\mathcal{M}' := Send - anyway - External - Events(\mathcal{M}, E)$;
66: For all $t \in \mathcal{I} \mid \forall t' \in \mathcal{I} : (t', t) \notin Priority$ do
67: Begin
68: $\mathcal{M}" := Fire(\mathcal{M}', t)$; $\mathcal{M}' \xrightarrow{t} \mathcal{M}"$
69: $\mathfrak{M} := \mathfrak{M} \cup \{\mathcal{M}"\}$;
70: $PNT := PNT \cup \{(\mathcal{M}', Labels(t), \mathcal{M}")\}$;
71: $\mathcal{I} := \mathcal{I} \setminus (\{t\} \cup \{t' \mid^{\circ} t \cap^{\circ} t' \neq \emptyset\})$;
72: $Explore2(\mathcal{M}", \mathcal{I}, \mathcal{T}_e, \mathcal{T}_i)$;
73: End
74: End

```
75:  End
End Explore2.
```

Synthesizing the marking graph of the whole system

The graph obtained by $SimulationAlgorithm1$ or $SimulationAlgorithm2$ is an asynchronous transition system $G = \langle Q, \hookrightarrow, \Sigma, q_0, I\rangle$ where $Q = \mathfrak{M}, \hookrightarrow = PNT, q_0 = \mathcal{M}_0$ and the independence relation "I" includes all pairs of non-conflicting events labeling transitions we can execute through the same steps. Note that the only case violating the determinism property of an asynchronous transition system is when two transitions triggered by the same event and labeled with the same action are concurrently fireable in orthogonal regions. Although this case is rare, a possible solution is to distinguish labels of the two transitions by adding some indices.

The Petri net of the entire system is built of inter-linked TPN-components related to its state machines which progress at different paces. Hence, generating the complete marking graph needs a complex handling because it should take into account both the run-to-completion policy within components and the concurrency between them. To that effect, we proceed progressively with it by combining partial marking graphs of TPN-components two by two. The marking graph of a combination of two components results from a parallel (merging) product of their partial marking graphs where each part can evolve independently of the other one when performing internal transitions. But, when executing external transitions, the two parts should synchronize (see Fig.3.13). In other words, we let each graph progress if it executes some action which does not belong to the set of common events otherwise the two graphs should synchronize. Formally,

Definition 2 . *Parallel (Merging) product of reachability graphs*

Let G_1, G_2 *be two graphs where:* $G_1 = \langle Q_1, \Sigma_1, \hookrightarrow_1, q_{0_1}, I_1\rangle$ *and* $G_2 = \langle Q_2, \Sigma_2, \hookrightarrow_2, q_{0_2}, I_2\rangle$. *The parallel product of* $G1$ *and* G_2 *yields a new graph* $G_1 || G_2 = G = \langle Q, , \Sigma, \hookrightarrow, q_0\rangle$ *with:*

- $Q \subseteq Q_1 \times Q_2$,
- $q_0 = (q_{0_1}, q_{0_2})$ *where* q_{0_1} *and* q_{0_2} *represent respectively initial markings of involved components,*
- $\Sigma \subseteq \Sigma_1 \cup \Sigma_2$,
- $\hookrightarrow = \{(q_i, q_j) \xrightarrow{a} (q'_i, q'_j) \mid (a \in \Sigma_1 \cap \Sigma_2 \wedge (q_i \xrightarrow{a} q'_i) \in \hookrightarrow_1 \wedge (q_j \xrightarrow{a} q'_j) \in \hookrightarrow_2) \vee (a \notin \Sigma_1 \cap \Sigma_2 \wedge ((q_i \xrightarrow{a} q'_i) \in \hookrightarrow_1 \wedge q'_j = q_j) \vee ((q_j \xrightarrow{a} q'_j) \in \hookrightarrow_2 \wedge q'_i = q_i))\}$.

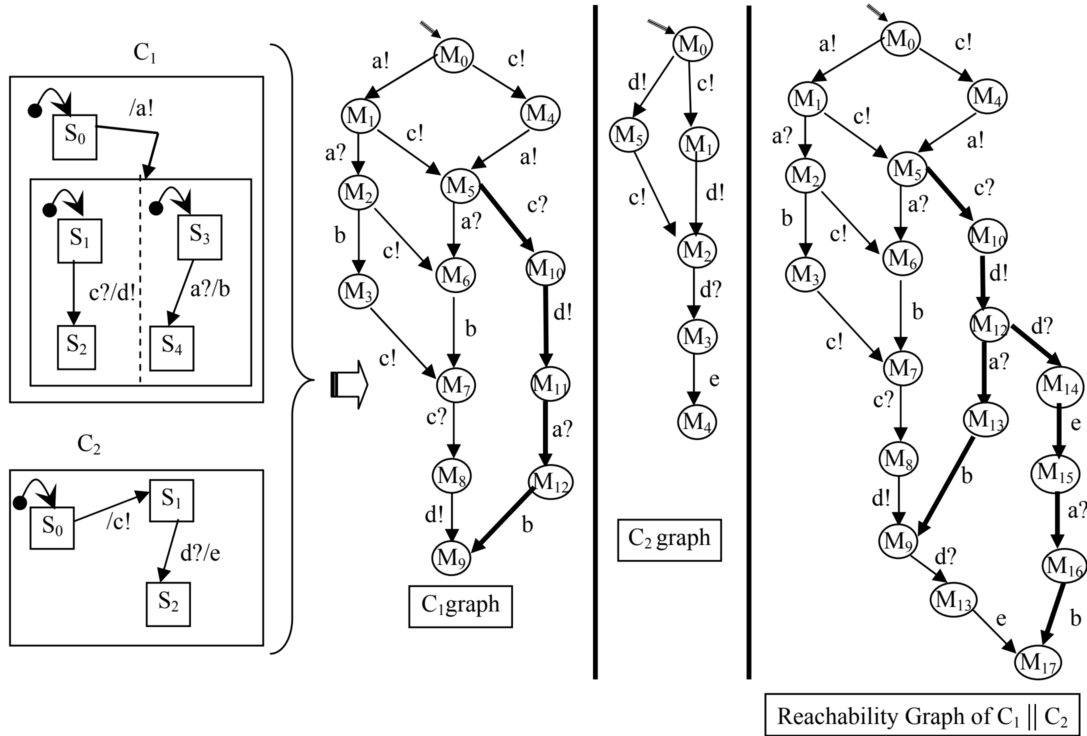


Figure 3.13: Partial reachability graphs of components C1 and C2 and their synchronization product.*Paths with thick lines will be discarded if we apply the second algorithm of state space exploration.*

- $I = I_1 \cup I_2$.

Let $D_1, \ldots, D_N$ be the state machines related to the components of the whole system Sys. Let $[\![.]\!]$ denote the semantic function that maps each diagram into its reachability graph and let $||$ be the parallel operator over these behavioral graphs. Then, $[\![Sys]\!] = [\![D_1]\!] || \ldots || [\![D_N]\!]$.

3.4 Adding time constraints in the marking graph

3.4.1 Time annotations on UML state machines

Since the UML Standard was adopted, it has been used in a large number of time-critical systems. As the design of these systems requires the observation of quantitative system aspects, a consensus had emerged that UML 1.x is lacking in some key areas related to real-time systems, in particular, quantifiable notion of time. Fortunately, UML has all the requisite mechanisms for addressing these issues through its extensibility faculties [102].

In fact, besides signal and method call events, the UML standard provides another kind of events modeling the passage of time. This event referred to as *time event*, denotes the time elapsed since the current state was entered and it occurs at the instant when a specified deadline has transpired. An action triggered by such an event, is called wait time (i.e. *after d*) action. Conversely, whatever the trigger event the state machine receives, the raised actions belong to one of the following kinds, each of which does not hold any measurable notion of time:

- Asynchronous action that raises a signal event.
- Synchronous action that raises a call event and waits for a response.
- (Abstract) operation without raising any event.

To overcome drawbacks of UML 1.x, the recent version UML 2.0 [103] introduces a new sub-package ***SimpleTime*** which adds meta-classes to represent time and durations, as well as actions, to observe time passing. Unfortunately, this package fits for time annotations on sequence diagrams rather than on state diagrams. For that reason, we prefer to use timing constructs and stereotypes given in the OMG profile of time specification [102].

In the domain model of this time profile, an event is assumed to occur instantaneously. That is, it takes place at a particular time instant and has no duration. An event occurrence can be associated with a time value relative to some clock to identify the time when it occurred. Such an event is called *timed event* though it may be of any kind (e.g., the sending and receiving of signals, the invocation of an operation).

When an event occurs, it may cause a number of stimuli to be generated. To allow modeling of stimuli that have an associated timestamp, the time profile introduces the notion of timed stimulus which is a stimulus that has at least one associated time value (timestamp).

It is also very useful to have a common abstraction for an action that takes time to complete: a *timed action*. This provides a generic facility for modeling behavior that has a definitive start time and a definitive end time. A special kind of timed action is a deliberate delay action, which delays execution for some time interval.

Thus, former actions may be easily tagged (stereotyped) by time notes to express quantitative requirements upon execution or transmission delays as depicted commonly by the followings:

- Timed Action; an action with duration (start time and end time).
- Timed stimulus; an action raising an event having duration or transmission delay.

- Action raising Timed Event; some action that produces an event with occurrence time.

It is worth noting that to make our temporal semantics conform to synchrony assumption, every synchronous action is mapped into two transitions which the first one produces a call event and the second transition is labeled with the receipt action of the result. However, an asynchronous action is mapped into only one transition whose label is a sending action of its related signal.

For more clarity, when we come across any arc stereotyped with a *timed action* in a state machine, we consider it as an operation we translate into two actions *start* and *end* without any raising of signal or call event. Conversely, we use the stereotype *timed stimulus* to adorning only arcs that generate events.

3.4.2 Mapping time annotations into time constraints

To sum up, our mapping approach consists in extracting time annotations of the state machine and translating them into time formulas over logical clocks [1]. Then, we assign these constraints to related nodes and transitions of the marking graph in a similar way to timed automata [1].

Definition 3 *(timing constraint) Let χ be a finite set of clocks ranging over $\mathbb{R}^{\geq 0}$ (set of non negative real numbers). The set $\Psi(\chi)$ of timing constraints on χ is defined by the following syntax: $\psi ::= true | x \ll c | x - y \ll c | not(\psi) | \psi \wedge \psi$ where $x, y \in \chi$, $c \in \mathbb{R}^{\geq 0}$ and $\ll \in \{<, \leq\}$. Other assertions such as, $x > 3$, $2 \leq x < y + 5$, $\psi \vee \psi$ can be defined as abbreviations.*

We add two mappings δ_1, δ_2 to the transition system as follows:

- $\delta_1 : Q \longrightarrow \Psi(\chi)$,
- $\delta_2 : (\hookrightarrow) \longrightarrow \Psi(\chi) \times 2^{\chi}$.

The first mapping δ_1 assigns to each node (in Q) a sojourn condition called activity condition which may be $true$, whereas the second mapping δ_2 associates with each transition (in $\hookrightarrow$) a firing condition along with a set of clocks initializations $\{h_i := 0 \mid i \in I\}$ which may be empty. In a timed graph (see Fig.3.14(a.3)), the system can stay in any reached node as long as the activity condition is $true$. Meanwhile, the system can leave this node through any outgoing arc whose firing condition becomes $true$. If the sojourn condition in the node becomes $false$ and if firing conditions on all outgoing arcs are not yet $true$, we get a time deadlock situation. In order to uncover such timing inconsistencies in produced timed graphs, one can make use of

the various methods which are proposed for their analysis [1]. We outline, below, the different cases concerning the mapping of the state machine time annotations into time constraints over logical clocks that we put on nodes and transitions of produced marking graphs. We, exactly, discern five cases related to time stereotypes that we commonly find on state machines arcs:

Case 1: *RTaction* stereotype including delay information that specifies start and end instants, allows us annotating timed actions (see Fig.3.14(a.1)).

If the event place receives a trigger token after the control flow place has been marked (Fig.3.14(a.2)), the start transition becomes enabled but does not fire until it waits d_1 units of time and the end action fires d_2 units after enabling the start action. Note that when receiving a trigger event, the abort transition cannot fire because it has less priority than others. However, if a conflicting enabled transition fires and consumes the token of a control flow place then our first transition becomes disabled, allowing therefore the abort action to consume its event trigger.

In the graph portion of Fig.3.14(a.3), the arc incoming to the marking $\mathcal{M}_1$ should initialize the clock h in order to enforce execution of "$start-a$" at the specified instant d_1. Thereafter, at any marking (either $\mathcal{M}_3$ or $\mathcal{M}_4$) from which "$end-a$" is fireable, all outgoing arcs can be taken only if the clock value is equal to d_2.

Consider, now, what would happen throughout a run-to-completion step if we have two enabled conflicting transitions where the first one should execute immediately and the second one should wait until a start time is reached. Is the choice always made in favor of the urgent transition against the delayed one? In respect of the UML semantics, any enabled action has to execute immediately after reception of its trigger event otherwise abort action consumes it. So, in our approach, the choice between two conflicting actions has to be done at the moment of the trigger receipt independently of their true execution times, thereby, an exit action of a previous state arises and a transient status is reached waiting for firing of the chosen action.

Case 2: *Timed stimulus* labels any state machine arc with an *RTstimulus* stereotype which includes both instants of request sending and response receiving (see Fig.3.14(b)).

As for timed action, timed stimulus is split into two basic actions (send and receive) that are handled by means of the same mapping described in case 1.

Case 3: *Timed event* is denoted by a state machine arc with *RTevent* stereotype including only send instant. Subsequently, this case is handled as the stereotype *RTstimulus* but without taking into account the constraint time of the event receiving.

Case 4: The stereotype *RTdelay* on a state machine arc depicts an empty action that needs to fire after a specified delay elapsed since it was lastly enabled (see Fig.3.14(c)).

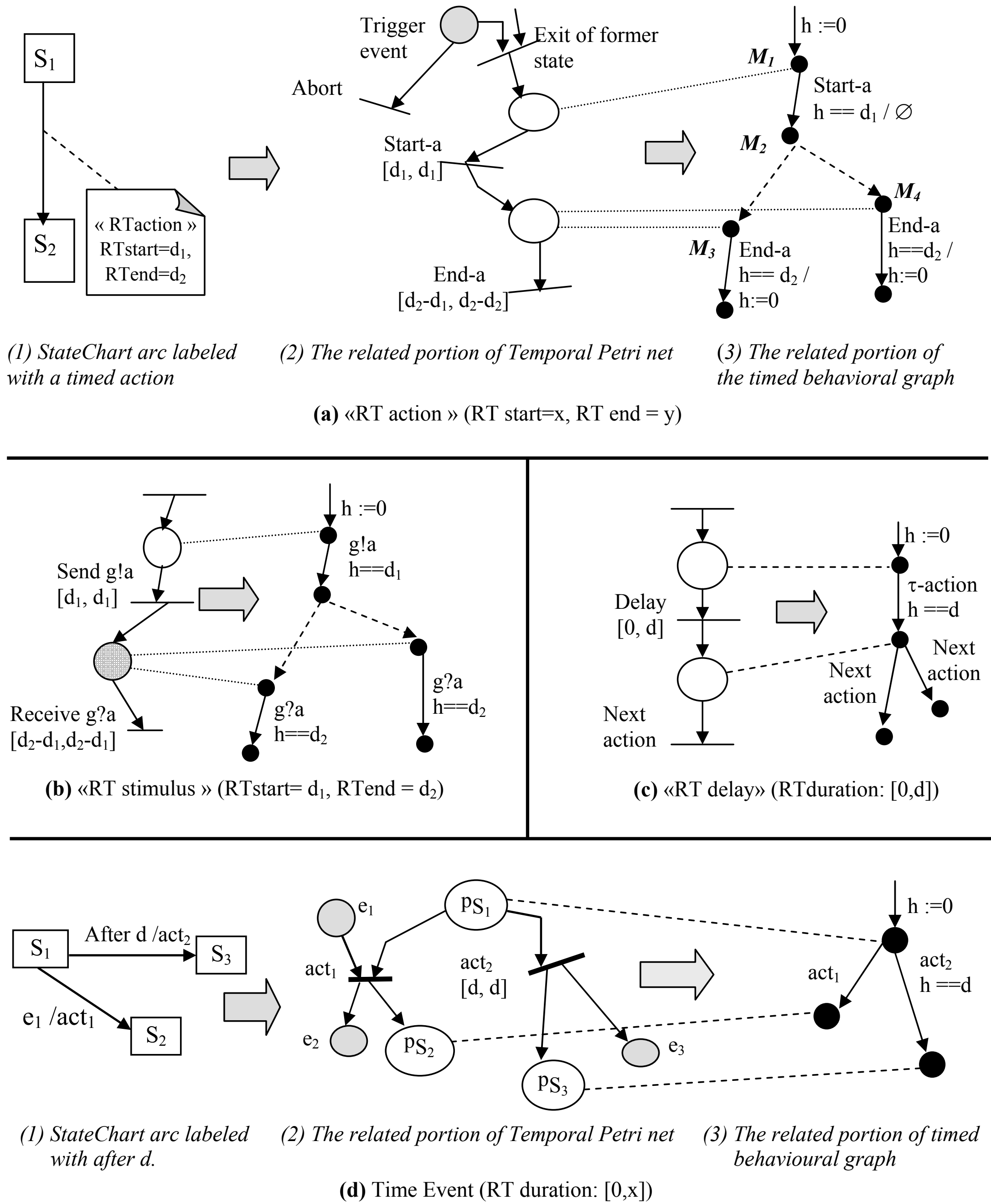


Figure 3.14: Mapping time annotations into time constraints.

Case 5: The time event (*After d*) is a particular label on state machine arcs (see Fig.3.14(d.1)). It depicts a timeout of a given deadline d relative to the time of entry into the source state S_1. As illustrated by Figure 3.14(a.2), when the action act_1 is not triggered (by receipt of e_1) before d units of time since the source state S_1 was entered, then exactly at instant d the action act_2 is launched. Such a scenario is easily mapped into the graph portion of Figure 3.14(a.3).

Remark 12: Time Stereotypes can be extended to deal with more complex constraints instead of those with precise start and end instants. The above mapping can also be straightforwardly customized to get time constraints in terms of inequalities over logical clocks.

3.5 Conclusion

In this chapter we have presented a new approach towards the comprehensive formalization of UML state machines for analysis and verification purposes at early stages of software development. The implementation model is a rigorous formalism, namely modular time Petri nets. We have enhanced that formalism with some suitable concepts (such as use of a modular style, thread places, event flow and control flow places, decomposition of some actions into instantaneous actions, etc.) so that our method succeeds in dealing with both all kinds of composite states and pseudo-states and allows us capturing the temporal annotations on state machine arcs. Once a state diagram has been converted into inter-linked Petri net modules related to state machines of involved components, we could gradually generate a whole reachability graph from those of components. Note that state space exploration can be tailored with respect to any intended run-to-completion policy. Time annotations are also mapped into time constraints to put onto nodes and arcs of produced graphs.

In the next chapters, we take advantage of the yielded behavioral models to proceed to the substitutability checking as well as semantics consistency and compatibility analysis of objects and components (designed by means of UML state diagrams) in respect of properties depicted by UML 2.0 sequence diagrams.

Chapter 4

Formal Semantics of UML Sequence Diagrams

4.1 Introduction

Scenarios-based specifications have become increasingly accepted as a means of requirements elicitation for concurrent systems such as telecommunications software. Indeed scenarios describe in an intuitive and visual way how system components and users interact in order to provide system level functionality. They are also used during the more detailed design phase where the precise inter-process communication must be specified according to formal protocols [43, 61, 92, 116, 137, 83].

The Unified Modeling language (UML [103]) which is an OMG standard and multi-paradigm language for description of various aspects of complex systems, adopted early this kind of visual and flexible notations for expressing the interactions between system components and their relationships, including the messages that may be dispatched among them. More precisely, UML contains 3 kinds of these interactions diagrams: sequence diagrams, communication diagrams and timing diagrams [103].

Recently, many established features of MSC (Message Sequence Charts) [76] have been integrated into the version 2.0 of UML [103], namely the interaction operators among fragments and the adoption of partial order among interaction events rather than the related messages.

Each interaction fragment alone is a partial view of the system behavior but when combined all together by means of the new interaction operators, interactions provide relatively a whole system description.

However in spite of the expressiveness and precise syntactic aspects of UML notations, their semantics remain described in natural language with sometimes OCL formulas. Accordingly, to use automated tools for analysis, simulation and verification

of parts of produced UML models, UML diagrams should be given a precise and formal semantics by means of rigorous mathematical formalisms [137, 61].

In this context, this chapter presents our formal approach (extending that of [48]) to defining branching time semantics for UML Sequences Diagrams in a denotational style. Our approach deals with the partial order among events and yields lattice-like graphs that faithfully specify the intended behaviors by recording both traces of all interaction components together with branching bifurcations. We provide our mathematical structure with few generalized algebraic operations making it easy to give formal definitions of interaction operators in a compositional manner. Next, we extend our formalism with logical clocks and time formulas over these clocks to express timing constraints of complex systems. Some given algorithms show how to extract time annotations of sequence diagrams and transform them into timing constraints in our timed graphs.

Obviously, this approach alleviates more the hard task of consistency checking between UML diagrams, specifically interaction diagrams with regards to state diagrams. Timeliness and performance analysis of timed graphs related to sequence diagrams could take advantage of works on model checking of timed automata [1].

The chapter is structured as follows: The following Section 4.2 presents the motivation of our formalization approach and the related word. Section 4.3 shows the basic features of UML Sequences Diagrams and in Section 4.4 we present our formal model and its algebraic operations. Then, in Section 4.5, the semantics of sequence diagrams are given in a compositional style by combining the denotations of interaction fragments using our algebraic operations and Section 4.6 presents a temporal enhancement of our graphs with logical clocks and temporal formulas and then we give the method to extract into our timed graphs the timing constraints from time annotations on sequence diagrams. Finally, we give concluding remarks.

4.2 Motivations and related work

In the following subsections, we argue the choice of using UML sequence diagrams as an interaction specification language as well as assigning them a branching time semantics instead of the classical traces based semantics. We then explain our formalization approach and compare it to other related methods.

4.2.1 Fragment-based specifications versus monolithic model specifications

Formal behavioral requirements' specifications can be classified into two categories: assertion-based specifications and model-based specifications [30].

With assertion-based specifications, high-level requirements are decomposed according to a fragment-based approach into more precise lower-level requirements that are mapped one-to-one to formal assertions expressed either as some modal logic assertion or graphical sequence-based or state-based chart. On the other hand, with model-based specifications, a single monolithic formal model (either as a state-based or algebraic-based system) is created to capture the combined intended behavior by the lower-level requirements.

We advocate the assertion-based over the model-based approach to verification for requirements' specifications because the former allows the system developers to modularize their thinking and focus on each property (or sets of properties) in isolation. In addition, it is much easier to verify the behavior of the actual system against each assertion (or sets of assertions) than comparing the equivalence of two monolithic formal models. When a requirement changes, it is harder to adjust the monolithic model without affecting the behavior related to other requirements whereas a fragment based approach has much lower maintenance cost and is more traceable than the monolithic approach because requirements are represented, one-to-one, by separate assertions. Furthermore, the fragment based specifications are suitable to represent, not only allowable behaviors, but also illegal behaviors. For instance, the "*neg*" and "*assert*" operators over interaction fragments in sequence diagrams allow depicting forbidden scenario in two different ways[1].

However, we deem that, in order to effectively preserve the aforementioned advantages of assertion based specifications, we have to extend this fragment based approach to the verification step where partial implementation models need to be extracted from the state diagrams using a modular exploration algorithms and then these could be compared to their related specification fragments to uncover potential inconsistencies.

4.2.2 Branching time semantics versus trace semantics for sequence diagrams

According to the UML specification [103], the semantics of an *Interaction* is given as a pair of sets of traces, representing valid traces and invalid traces respectively [61, 70, 117]. The traces that are not included neither in valid nor in invalid traces are

[1] See the papers [110], [64], and [116], which discuss several different meanings of operators specifying negative scenarios and their impact on the semantics of UML.

not described and we cannot know whether they are valid or invalid. These traces are referred to as contingent in [117]

Trace semantics are often defined in a denotational style and the semantics function (denoted by $[\![.]\!]$) for interaction operators, extensively makes use of set operators. For instance in [86, 112, 111, 61, 70, 117], the semantics function maps the combined interaction through the alternative operator into the union of trace based sets related to its combining fragments as follows: $[\![alt(P, Q)]\!] = [\![P]\!] \cup [\![P]\!]$). As the semantics function yields pairs of traces sets, the set operations are generalized to pairs of sets, as follows: $< A, B > \cup < C, D >=< A \cup C, B \cup D >$. Such a definition can lead to some inconsistencies: Assume $< A, B >=< \{ab\}, \{axy\} >$ and $< C, D >=< \{axy\}, \{ab\} >$ where symbols a, b, x and y denote interaction events. Then, we get $< A, B > \cup < C, D >=< \{ab, axy\}, \{ab, axy\} >$, which does not make sense. It is obvious that these problems particularly result from the aforementioned linear time semantics where valid and invalid traces are unrelated to each other.

Unfortunately, trying to overcome this flaws by using Readies (resp.Failures) traces semantics [116], raises other kinds of problems. The notion of contingency does not comply with the readies (respectively, failures) traces semantics where we have to show after every trace the set of actions which can be accepted (respectively, refused).

Besides this, we think that linear time semantics which were applied first to the previous (basic UML 1.x) sequence diagrams are no more appropriate to the new (UML 2.x) high level sequence diagrams. Indeed traces semantics cannot capture many opportune information brought by interaction operators such as the branching time structure (e.g., alt), independence relation between event occurrences in parallel fragments, mutual exclusion ...[48, 96].

Consequently, we promote the use of branching time semantics in the interleaving and true concurrency contexts in such a way that all above information could be used for consistency checking of specification models against state diagrams particularly for reactive systems.

4.2.3 Related work in formalization of UML Interactions

Because of the widespread use of interaction diagrams in complex systems, many efforts have been made to give them formal meanings in order to allow systematic tool support during design, implementation and validation phases.

Besides the textual semantics given in UML specification document [103], many approaches [61, 116] present formal semantics of sequence diagrams (SD) where the runs are widely defined in terms of pairs of valid and invalid traces and do not record information about choice opportunities and coordination actions. Contrary to our

approach, these linear time semantics are not enough faithful to allow complete consistencies checking over UML dynamic diagrams particularly in concurrent systems. Moreover, timing constraints are not handled within these models.

On the other hand, some papers attempt to synthesize high level diagrams (StateCharts [137, 43], Petri nets [24]) from SD or MSC. However, in our opinion the sequence diagrams are less structured description languages (in spite of the recent improvements) as are assembly languages for high level programming ones. Furthermore the built high level models seem too unfolded or flattened and their high level syntactic constructs are not suitably used and may generate some inconsistencies with regards to the original sequence diagrams [137]. In this later work, the authors try to retrieve state diagrams of objects involved in interactions described by means of relatively simple sequence diagrams. The approach consists in deriving flat automata of some object from its lifelines in all interaction fragments and then combines them by means of simple interaction operators (choice, strict sequencing and loop). The other operators are discarded as the parallel operator so that the resulting automaton is flat and unfolded (without orthogonality) and may generate some irregular behaviors because of the removal of coordination information when extracting partial views.

An interesting work [43] considers good and bad interactions of reactive systems as safety and liveness properties that are described in terms of Büchi automata allowing refinement. SD traces become only prefixes of accepted infinite sequences and the various combinations between automata are not specified with regard to SD operators.

Another paper [92] uses process algebra terms to characterize the traces of scenario based specification that are defined by a causal ordering. It proves a canonical solution for correcting race conditions within the system behavior by weakening the causal relationship.

Note that many papers were proposed to overcome shortcomings of UML 1.x specification that relies on the ordering of messages instead of related actions. Hence authors of [24] and [115] proposed a formal semantics to the interaction diagrams of UML 1.x by the generation of an order relation that schedules the message emissions and receptions and can be automatically translated into a flattened Petri net or automata. Similarly, [83] presented a methodology to convert UML 1.x SD to a context-free grammar and applied parsing theory to locate non-determinism behavior. Additional information is discussed to attain deterministic behaviors for embedded systems modeling. Also, the approach of [135] proposes a formal semantics of UML 1.x sequence diagrams in terms of ordered hierarchical tree structure that represents the hierarchical relations among the messages (method invocations).

Contrary to these approaches, the new specification of UML 2.0 [103] adopts an ordering over events occurrences corresponding to sending and receiving of messages

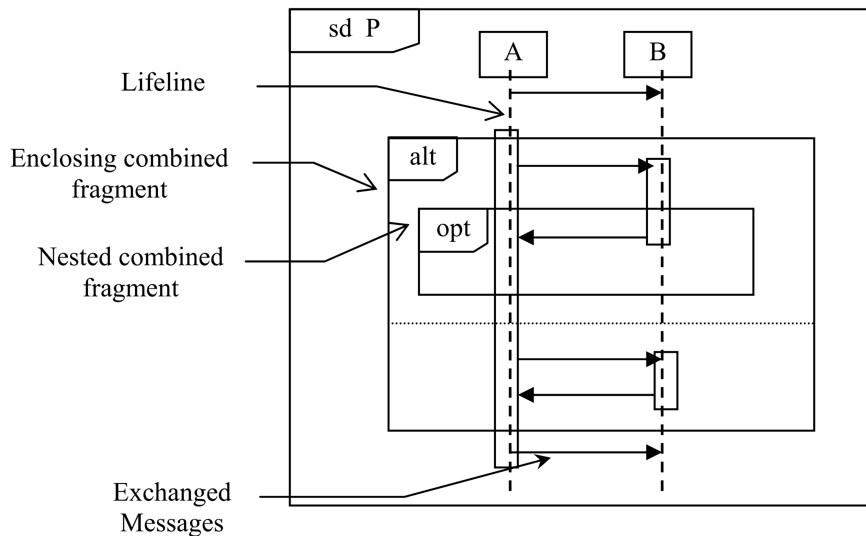


Figure 4.1: Constituents of sequence diagrams.

and encompasses high level features of MSC [76] for composition of Interactions allowing the description of more complex behaviors. At last, all the above works do not pay attention to time annotations on sequence diagrams.

4.3 Interactions and sequences diagrams

In contrast to UML state machines which focus on internal behaviors of components, UML sequence diagrams express in an intuitive and visual way how system components should interact and communicate to provide system level functionality. Now, many established features of Message Sequence Charts [76] have been integrated into the version 2.x of UML [103], namely, interaction operators among fragments and adoption of partial order among interaction events rather than the related messages. Each interaction fragment alone is a partial view of the system behavior but when combined altogether by means of the new interaction operators, interactions provide a more complete system specification.

4.3.1 Features of sequence diagrams

The notation for an interaction in a sequence diagram is a solid-outline rectangle of which upper left corner contains a pentagon. Inside this pentagon, the keyword *sd* is written followed by the interaction name and parameters [103].

In a sequence diagram (see Fig.4.1 and Fig.4.2(a)), participants (components, objects . . .) that participate in the interaction are located at the top of the diagram

across the horizontal axis. From each component shown using a rectangle, a lifeline is drawn to the bottom and the dispatching of every message is depicted by a horizontal arc going from the sender component to the receiver one. These messages are ordered from top to bottom so that the control flow over time is shown in a clear manner. Each message is defined by two events:

Each message m is defined by two events: message emission $m!$ and message reception $m?$ and events situated on the same lifeline are ordered from top to down (see figures 4.2(b), 4.2(c) and 4.2(d) depicting respectively the behavioral graphs of enclosed interaction fragments of the sequence diagram given in Fig.4.2(a)).

Accordingly, a message defines a particular communication among communicating entities. This communication can be "raising a signal", "invoking an operation", "creating", or "destroying an instance". The message specifies not only the kind of communication but also the sender and the receiver. The various kinds of communication involved in distributed and concurrent systems are considered in UML sequence diagrams. Messages may hence be either synchronous or asynchronous [103].

Besides this, UML 2.x sequence diagram introduces the notion of basic interaction fragment that only represents finite behaviors without branching (when executing a sequence diagram, the only branching is due to interleaving of concurrent events), but these can be composed to obtain more complete descriptions. Basic interaction fragments can be composed in a composite interaction fragment called combined interaction or combined fragment using a set of operators called interaction operators (see Fig.4.1). The unary operators are *opt*, *loop*, *break*, *assert* and *neg*. The others have more than one operand, such as, *alt*, *par*, seq_{weak} and seq_{strict}. The combined fragments, themselves, can be recursively combined together until obtaining a more complete diagram [103].

The notation for a combined fragment in a sequence diagram is a solid-outline rectangle. The operator is in a pentagon at the upper left corner of the rectangle. The operands of a combined fragment are shown by tiling the graph region of the combined fragment using dashed horizontal lines to divide it into regions corresponding to the operands (see Fig.4.1).

Thus, each complete sequence diagram (or interaction overview diagram) *sd* is built up by combining its interaction fragments by means of a set of interactions operators $IntOp = \{alt, opt, par, seq_w, seq_s, loop, break, assert, neg\}$ where:
- Interaction operators from $\{alt, opt, break\}$ perform, respectively, choices between behaviors of two interaction fragments, between one fragment and nothing or between a nested fragment with the remainder of the enclosing fragment.
- seq_w and seq_s operators are, respectively, weak and strict versions of sequencing between behaviors of two operands.

- *loop* allows to repeat the operand behavior many times.
- *neg* allows to define forbidden scenarios.
- *assert* defines the only valid (i.e., allowed) scenarios.

4.3.2 Outline of the translation approach

Interactions in UML 2.x sequence diagrams are considered as collections of events instead of ordered collections of messages as in UML 1.x. These stimuli are partially ordered based on which execution thread they belong to. Within each thread the stimuli are sent in a sequential order while stimuli of different threads may be sent in parallel or in an arbitrary order.

The dynamic semantics of sequence diagrams remains described in natural language [103]. So, to allow using automated tools for their analysis and verification, sequence diagrams should be given a formal semantics by means of rigorous mathematical formalisms [43, 85]. In this context, we defined in [48] a branching time semantics for UML sequence diagrams which record faithfully the intended traces of events related to exchanged messages between different components together with branching choices.

It is worth noting that although we use an intermediate object-based formal model to implement sequence diagrams, at last, we have to abstract away from them some implementation details in order to derive labeled transition systems (also referred to as a behavioral graph or automaton) $\mathcal{A}_{Spec} = \langle Q, \Sigma, \hookrightarrow, q_0, \mathcal{F}, \mathcal{E} \rangle$ where :

- Q is the set of reachable states with some initial one q_0,
- Σ is the set of events,
- $\hookrightarrow \subseteq Q \times \Sigma \times Q$ is the set of transitions labeled with events whose occurrences cause changes of the current state,
- $\mathcal{F} = \{q \in Q | \forall a \in \Sigma, \nexists q \xrightarrow{a} \}$ is the set of final states depicting successful termination statuses[2],
- and $\mathcal{E}$ is the set of erroneous states which depict forbidden computations that the system should avoid.

Below, we summarize this formal method which uses a few generalized algebraic operations over transition systems to give formal meanings of the interaction operators in a compositional manner.

[2]Any sink state (without successors) which does not belong to $\mathcal{F}$ would be considered as a deadlock state.

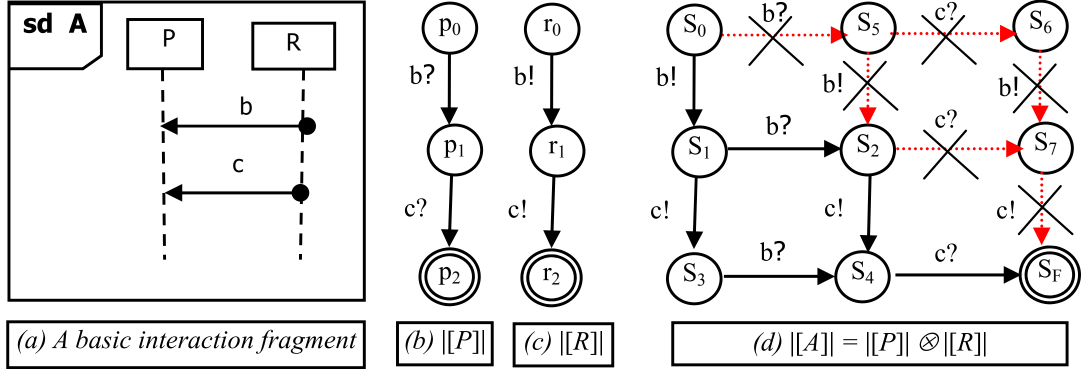


Figure 4.2: Mapping of basic interaction fragment.

We build the transition system (i.e., behavioral graph) of *sd* by combining graphs related to the involved fragments by means of three associative algebraic operations defined in [48]: choice "⊕", synchronization product "⊗" and star "∗". The combined graphs themselves are recurrently built by combining models of their interaction fragments and so on [48].

For instance, the behavioral graph of the sequence diagram of Fig.4.2(a), is illustrated by Fig.4.2(d). It results from an adequate combination of the graphs given respectively in Fig.4.2(b) and Fig.4.2(c) and related to the lifelines of its participants.

Let $[\![.]\!]$ be the semantics function that builds up the transition system of *sd* (note it $[\![sd]\!]$) by mapping its interaction fragments into transition systems in an incremental way:

1. First, we assign to each participant lifeline P its graph $[\![P]\!]$ by extracting the tree depicting sequences of ordered events that are sent or received on the lifeline of P (see figures 4.2(b) and 4.2(c)).

2. Then, for any basic interaction fragment X, we build its graph $[\![X]\!]$ by combining the basic graphs related to involved participants $\{P_i\}_{i\in\{1...n\}}$: $[\![X]\!] = [\![P_1]\!] \otimes ... \otimes [\![P_n]\!]$. Note that such a combination product has to preserve the orderings upon events from the same lifelines along with the natural orderings between sending and receipt events of every involved message. This amounts to prune many paths transgressing the corresponding orderings (see Fig.4.2(d)).

3. Next, we recurrently achieve various algebraic combinations (using choice "⊕", synchronization product "⊗", and star "∗") among yielded graphs depending on the interaction operators over the related fragments until we obtain the whole transition system $[\![sd]\!]$, as follows:

- Let X be an interaction fragment obtained by combining two fragments X_1 and X_2 via an interaction operator $\odot_{sd}$. Let $\odot_G$ be the graph operation which is the mapping of $\odot_{sd}$ by $[\![.]\!]$. Then, $[\![X]\!] = [\![X_1 \odot_{sd} X_2]\!] = [\![X_1]\!] \odot_G [\![X_2]\!]$.
- The mapping of any interaction operator from $\{alt, opt, break\}$ is a choice operation $\oplus$ of the related graphs. However, each interaction operator from $\{par, seq_s, seq_w\}$ is mapped in a parameterized version of the synchronization product $\otimes$ which takes care of orderings among events within interleaved sequences. The last operator *loop* is obviously mapped into the star operation "$*$" on graphs [48].

4.4 Formal model for an Interaction behavior

Notations. Let Σ be a vocabulary of symbols. Σ^* is the set of all finite words over Σ including the empty word ϵ. Let $w \in \Sigma^*$, $|w|$ denotes the length of w and $w(n)$ denotes the nth symbol of w. If $u, v \in \Sigma^*$, then $u.v$ denotes the concatenation of u and v.

We present below the intermediate implementation model of SD we use for the definition of the branching time semantics of sequence diagrams. This model is a *lattice-like* graph which records faithfully from sequences diagrams the intended traces of events together with the choices possibilities. Moreover, it chiefly preserves the partial order among events in such a way that the structure may contain diamond-shaped parts.

$G = \langle \mathcal{O}, \Sigma, <_\Sigma, S, s_0, T \rangle$ where :

- $\mathcal{O}$ is the set of participants involved in an interaction.
- Σ is the set of events occurrences. Note that Σ contains an unobservable event $\tau \neq \epsilon$ modeling change of control flow. This event τ may be also adorned with a guard.
- $<_\Sigma \subset \Sigma \times \Sigma$, is a set of pairs of events occurrences where each one represents a binary relation between two occurrences to describe that one event must occur before the other in a valid trace. This mechanism provides a partial order on events occurrences so that the set of possible sequences is more restricted.
- $S = \{s_k : \mathcal{O} \longrightarrow \Sigma^* \mid k \in \mathbb{N}\} = \{s_k : o_i \longrightarrow w \mid o_i \in \mathcal{O} \wedge w \in \Sigma^* \wedge \forall n \leq |w|, \forall m \leq n : (w(n), w(m)) \notin <_\Sigma\}$. Every mapping s_k from S assigns to each participant some word (trace of events occurrences) at some point k of its evolution such that the binary relation $<_\Sigma$ among events remains preserved. Hence, these mappings constitute the nodes of the graph.

- $s_0 \in S$ represents the initial node where no event is recorded. $\forall o_i \in \mathcal{O}, s_0(o_i) = \epsilon$.

- $T : S \times \Sigma \longrightarrow S$

  $(s_i, e) \longrightarrow s_j$

  Each transition records any occurring event different from τ onto the trace of the relating object. For more convenience, we write $s_i \overset{e}{\vdash} s_j$.

  If $e = \tau$ then $\forall o \in \mathcal{O} : s_j(o) = s_i(o)$

  If $e \neq \tau$ then there exits exactly one object o where:
  $e \in lifeline(o) \wedge s_j(o) = s_i(o).e \wedge \forall o' \in \mathcal{O}, \forall e' \in s_i(o') : (e, e') \notin <_\Sigma$.
  (e should never occur if it precedes any other recorded event via the partial order)

  $\forall o' \neq o : s_j(o') = s_i(o')$($s_j$ does not record e onto o' trace if $o' \neq lifeline(e)$).

We henceforth call final nodes leaf nodes rather than acceptance nodes because sequence diagrams are only some pieces of the expected behavior. So, any recorded trace is only a prefix of some whole traces we can only compute from state diagrams.

Remark 1 We can easily abstract the transition system $\mathcal{A}_{Spec} = \langle Q, \Sigma, \hookrightarrow, q_0, \mathcal{F}, \mathcal{E} \rangle$ from the implementation graph $G = \langle \mathcal{O}, \Sigma, <_\Sigma, S, s_0, T \rangle$ as follows: $Q = S$, $\hookrightarrow = T$, $q_0 = s_0$, $\mathcal{F}$ is computed from Q as explained before, and $\mathcal{E}$ will be yielded when handling *neg* and *assert* operators.

We define below, two binary and one unary algebraic operations on these kinds of graphs. These operations are generalized making it possible to define the formal semantics of interaction operators on interaction fragments in a compositional style.

4.4.1 Choice operation

This operation achieves an adjunct of graphs. Choice is made between them via internal τ-actions. Let G_1, G_2 be two graphs where: $G_1 = \langle \mathcal{O}^1, \Sigma^1, <^1_\Sigma, S^1, s^1_0, T^1 \rangle$, $G_1 = \langle \mathcal{O}^2, \Sigma^2, <^2_\Sigma, S^2, s^2_0, T^2 \rangle$, Such that $\mathcal{O}^1 = \mathcal{O}^2$
$G_1 \oplus G_2 = \langle \mathcal{O}, \Sigma, <_\Sigma, S, s_0, T \rangle$, where :

- $\mathcal{O} = \mathcal{O}^1 = \mathcal{O}^2$.
- $\Sigma = \Sigma^1 \cup \Sigma^2$.
- $<_\Sigma = <^1_\Sigma \cup <^2_\Sigma$.
- $S = S^1 \cup S^2 \cup \{s_0\}$ where $s_0 \in S$ is new initial node.
- $T = T^1 \cup T^2 \cup T'$ where $T' = \{s_0 \overset{\tau}{\vdash} s^1_0, s_0 \overset{\tau}{\vdash} s^2_0\}$.

4.4.2 Parameterized cartesian product

This product achieves merging of all pairs of traces from the two graphs but in such a way that the partial order among events remains preserved. Whenever we try to concatenate two traces, we should check that no event occurrence of the second trace is ordered before an event from the previous trace.

Let G_1, G_2 be two graphs where:
$G_1 = \langle \mathcal{O}^1, \Sigma^1, <^1_\Sigma, S^1, s^1_0, T^1 \rangle$, $G_1 = \langle \mathcal{O}^2, \Sigma^2, <^2_\Sigma, S^2, s^2_0, T^2 \rangle$,
$G_1 \otimes_{Prior} G_2 = \langle \mathcal{O}, \Sigma, <_\Sigma, S, s_0, T \rangle$ where :

- $\mathcal{O} = \mathcal{O}^1 \cup \mathcal{O}^2$.
- $\Sigma = \Sigma^1 \cup \Sigma^2$.
- $<_\Sigma = <^1_\Sigma \cup <^2_\Sigma \cup Prior$
- $Prior \subseteq (\Sigma^1 \times \Sigma^2) \cup (\Sigma^2 \times \Sigma^1)$ is a subset of new particular order relations among events.
- $S = PRUNE((S^1 \otimes S^2) \cup (S^2 \otimes S^1))$. This function $PRUNE$ removes unreachable nodes from the initial node through T.

  $$s_k \in S : o \rightarrow s_k(o) = \begin{cases} s^1_i(o) \otimes s^2_j(o) & \text{where } s^1_i \in S^1 \wedge s^2_j \in S^2 \\ s^2_i(o) \otimes s^1_j(o) & \text{where } s^2_i \in S^2 \wedge s^1_j \in S^1 \end{cases}$$

  $$s^1_i(o) \otimes s^2_j(o) = \begin{cases} s^1_i(o).s^2_j(o) & \text{if } o \in \mathcal{O}^1 \cap \mathcal{O}^2 \wedge \forall e \in s^1_i(o), \forall e' \in s^2_j(o) : (e', e) \notin <_\Sigma \\ s^1_i(o) & \text{if } o \notin \mathcal{O}^2 \\ \epsilon & \text{otherwise} \end{cases}$$

  $$s^2_i(o) \otimes s^1_j(o) = \begin{cases} s^2_i(o).s^1_j(o) & \text{if } o \in \mathcal{O}^1 \cap \mathcal{O}^2 \wedge \forall e \in s^2_i(o), \forall e' \in s^1_j(o) : (e', e) \notin <_\Sigma \\ s^2_i(o) & \text{if } o \notin \mathcal{O}^1 \\ \epsilon & \text{otherwise} \end{cases}$$

- $s_0 = s^1_0 \otimes s^2_0 = s^2_0 \otimes s^1_0$ (hence $\forall o \in \mathcal{O} : s_0(o) = \epsilon$).
- $T : S \times \Sigma \longrightarrow S$

  $T = \{(s_k, e, s_{k'}) \mid s_{k=}s^1_i \otimes s^2_j, s_{k'} = s^1_m \otimes s^2_n, \exists o \in \mathcal{O} : (s^1_i(o), e, s^1_m(o)) \in T^1 \vee (s^2_j(o), e, s^2_n(o)) \in T^2\} \cup \{(s_k, e, s_{k'}) \mid s_{k=}s^2_i \otimes s^1_j, s_{k'} = s^2_m \otimes s^1_n, \exists o \in \mathcal{O} : (s^2_i(o), e, s^2_m(o)) \in T^2 \vee (s^1_j(o), e, s^1_n(o)) \in T^1\}$.

4.4.3 Star Operation

This operation adds to the graph new τ-transitions outgoing from leaf nodes to the initial node. Furthermore, it adds a new empty node s_ϵ connected to the initial node by

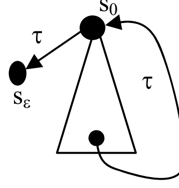


Figure 4.3: Star operation.

a τ-transition (see Fig.4.3). Let G_1 be the starting graph $G_1 = \langle \mathcal{O}^1, \Sigma^1, <^1_\Sigma, S^1, s^1_0, T^1 \rangle$. Then, $Star(G_1) = \langle \mathcal{O}, \Sigma, <_\Sigma, S, s_0, T \rangle$ where:

- $\mathcal{O} = \mathcal{O}^1$,
- $\Sigma = \Sigma^1$,
- $<_\Sigma = <^1_\Sigma$,
- $S = S' \cup S"$
- $S' = \{s_k : O \longrightarrow \Sigma^*, k \in \mathbb{N}\}$ where for all k we have :
  $$s_k : o \longrightarrow s_k(o) = \begin{cases} (s^1_k(o))^+ & \text{if } s_k \in LEAF(S^1) \\ s^1_k(o) & \text{otherwise} \end{cases}$$
- $S" = \{s_\epsilon : \mathcal{O} \longrightarrow \{\epsilon\}\}$

  The sole node s_ϵ records empty traces for all objects.
- $s_0 = s^1_0$.
- $T = T^1 \cup T' \cup T"$ such that:
  - $T' = \{s_F \overset{\tau}{\vdash} s^1_0 \mid s_F \in LEAF(S^1)\}$ where $LEAF(S) = \{s \in S \mid \nexists s' \in S, e \in \Sigma : s \overset{e}{\vdash} s'\}$
  - $T" = \{s^1_0 \overset{\tau}{\vdash} s_\epsilon\}$.

Property 1 *The two operations $\otimes$ and $\oplus$ on graphs are associative.*

Lemma 4 *Let G be the graph $\langle \mathcal{O}, \Sigma, <_\Sigma, S, s_0, T \rangle$.*
$\forall s_k \in S, \forall o \in \mathcal{O} : u = s_k(o) \Rightarrow (\forall i, j \in \mathbb{N} : i, j \leq |u| \wedge (u(i), u(j)) \in <_\Sigma \Rightarrow i < j)$.

Proof. Definitions of S and T compel event occurrences concerned by $<_\Sigma$ to occur in a way so that the partial order remains preserved. The other events may appear in any order in the sequence. ■

Definition 5 *Let u and w be two sequences from $s_k(o)(o \in \mathcal{O}, s_k \in S)$. u and w are equivalent (we write $u \approx w$) if and only if $\forall a \in \Sigma : a \in u \Leftrightarrow a \in w$. The precedence relation is preserved for ordered events in both u and w.*

Definition 6 *Let s_i and s_j be two nodes from S. s_i and s_j are equivalent (we write $s_i \approx s_j$) if and only if $\forall o \in \mathcal{O} : s_i(o) \approx s_j(o)$.*

Definition 7 *Let G be a graph $= \langle \mathcal{O}, \Sigma, <_\Sigma, S, s_0, T \rangle$. A reduced graph (automaton) may be obtained from the graph G by reducing the equivalent nodes into equivalence class of nodes as follows: $G = \langle \mathcal{O}, \Sigma, <_\Sigma, S', s_0, T' \rangle$*

$S' = \{[s_k] \mid s_k \in S\}$ *where* $[s_k] = \{s_i \in S \mid s_i \approx s_k\}$.

$T' = \{[s_k] \overset{e}{\vdash} [s_{k'}] \mid \exists s_i \in [s_k], \exists s_j \in [s_{k'}], \exists (s_i \overset{e}{\vdash} sj) \in T\}$.

Remark 2 On the other hand, we can unfold our graph (namely cycles and diamond shapes) in order to obtain the equivalent transition system.

4.4.4 Handling of synchronous messages

Although the subset $Prior$ is used particularly to handle specific features of interaction operators used among combined fragments (as explained later), we can also use it to handle synchronous messages when assembling jointly many lifelines in one interaction fragment or when combining many sequence diagrams by means of interaction operators. We have only to add into the partial order subset $Prior$ other general orderings with regards to send and receive events of those specific messages.

Let $\mathcal{M}_{Syn}$ be the set of synchronous messages between two combined fragments sd_i and sd_j (which could be only lifelines of participants).$Prior$ is then increased with the set $\bigcup_{m \in \mathcal{M}_{Syn}} (Prior_m)$ where :

$Prior_m = \{(m!, e) \mid m! \in \Sigma_j \wedge \exists e \in \Sigma_i : (m?, e) \in <_{\Sigma_i}\}$

$\cup \{(m?, e') \mid m? \in \Sigma_i \wedge \exists e' \in \Sigma_j : (m!, e') \in <_{\Sigma_j}\}$

This means that once the send event ($m!$) occurs the executing thread will stop until the receive event ($m?$) occurs on the other lifeline thanks to its precedence level against the successive events of the send event. Similarly, if we observe first a receive event on the second participant lifeline, the related thread should stop until the send event occurs on the first participant. Note that only related threads to these events should synchronize and other concurrent threads could continue performing parallel activities and generating others events.

4.5 Formal semantics of Interaction fragments and operators

4.5.1 Lifeline of a participant

We associate to each interaction fragment X a related graph denoted $[\![X]\!]$.
Let P be a participant in some interaction. Its graph is $[\![P]\!] = \langle \mathcal{O}, \Sigma, <_\Sigma, S, s_0, T\rangle$ where:

- $\mathcal{O} = \{P\}$ is a singleton set consisting in the only one participant P.
- $\Sigma = \{e \mid \exists m : e = Receive(m) \wedge Receiver(m) = P \vee e = Send(m) \wedge Sender(m) = P\}$
- $<_\Sigma = \{(e, e') \in \Sigma \times \Sigma \mid$ the event occurrence e occurs before e' on the lifeline(P)$\}$.

Frequently, the order on a same lifeline is total, i.e., if we take two event occurrences e, e' on the same lifeline then $e < e'$ or $e' < e$. But if the lifeline of P contains a coregion area then the order of event occurrences on this part is insignificant.

- $S = \{s_k : \mathcal{O} \longrightarrow \Sigma^*, k \in \mathbb{N}\}$
  $= \{s_k : P \longrightarrow w \mid \forall n \leq |w|, \forall m \leq n : (w(n), w(m)) \notin <_\Sigma\}$
- $s_0(P) = \epsilon$
- $T = \{(s_i, e, s_j) \mid s_j(P) = s_i(P).e \wedge e \in \Sigma\}$

Each transition yields thus a new trace onto the target node by adding its labeling event occurrence to the previous trace recorded in the source node of the transition.

For example, we use in Fig.4.4 a notational shorthand called "coregion area" for combined fragments where the order of events occurrences on the lifeline is insignificant.

4.5.2 Basic Interaction fragment

Recall that a basic interaction fragment is a piece of an interaction which involves many participants without using any interaction operator.

Let sd be a basic interaction between two participants P_1 and P_2. Herein $\mathcal{O}^1 \cap \mathcal{O}^2 = \emptyset$. The graph related to sd is obtained by a parallel merge of the graphs relating to the participants lifelines with respect to the partial order between send and receive events.

$[\![sd]\!] = [\![P_1]\!] \otimes_{Prior} [\![P_2]\!]$ where :

$Prior = \{(e, e') \in (\Sigma^1 \times \Sigma^2) \cup (\Sigma^2 \times \Sigma^1) \mid \exists$ message $m : e = send(m) \wedge e' = receive(m)\}$

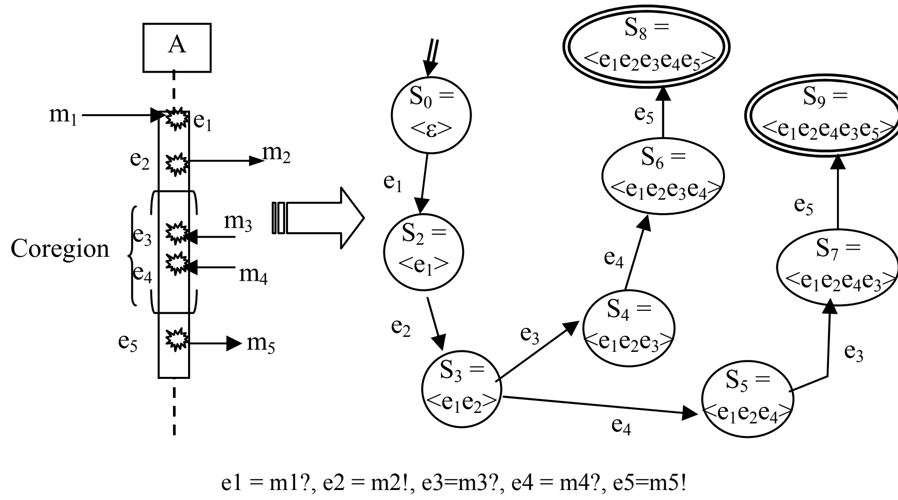


Figure 4.4: The graph related to a lifeline of a participant ($s_6 \approx s_7, s_8 \approx s_9$).

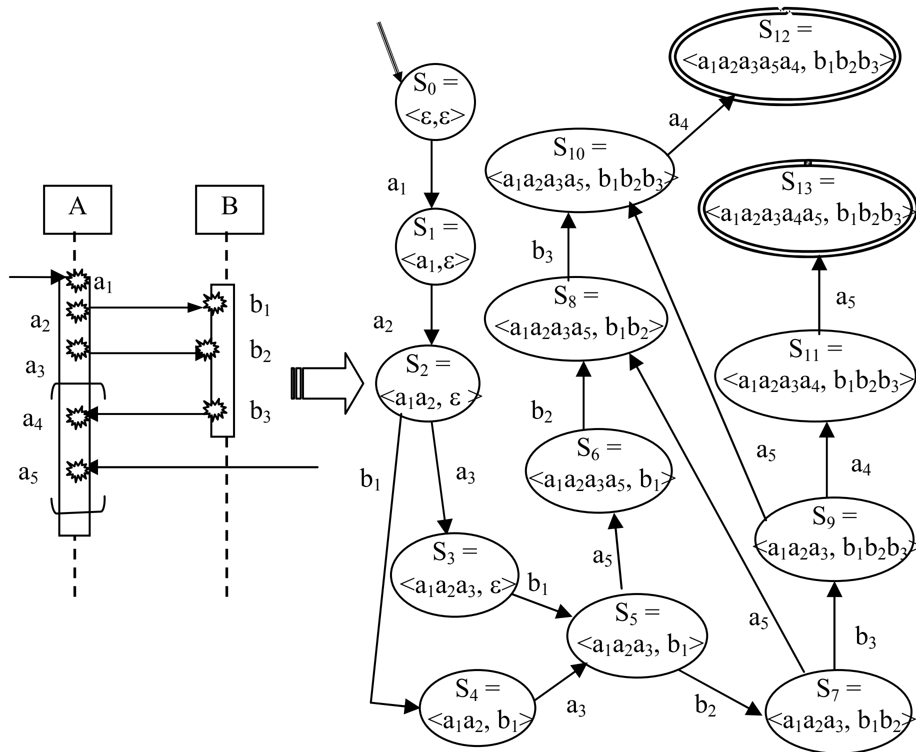


Figure 4.5: The graph related to a basic interaction fragment.

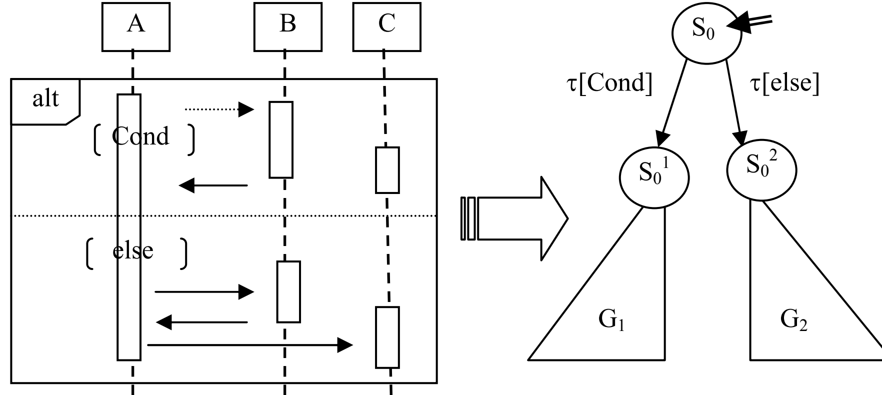


Figure 4.6: The graph related to an interaction fragment with *alt* operator.

4.5.3 Choice operator *alt*

Let sd be an interaction fragment combined from two interaction fragments sd_1 and sd_2 by means of the choice operator *alt*. This operator indicates that the resulting fragment represent a choice of behavior. At most one the operands will be chosen.

Formally, $[\![sd_1\ alt\ sd_2]\!] = [\![sd_1]\!] \oplus [\![sd_2]\!]$ (see Fig.4.6)

4.5.4 Operator *break*

Even though *break* is a unary operator which operand is a nested fragment in an enclosing interaction fragment, this operator can reduce to an interaction operation *alt* between the nested fragment and the remainder of the enclosing interaction fragment.

4.5.5 Operator *opt*

Let sd' be a new interaction fragment obtained by applying the operator *opt* on another interaction fragment sd. The operator *opt* designates that the resulting fragment represents a choice of behavior where either the sole operand happens or nothing happens. Formally, this means:

$[\![sd']\!] = [\![opt(sd)]\!] = [\![sd\ alt\ sd_\emptyset]\!] = [\![sd]\!] \oplus [\![sd_\emptyset]\!]$

The empty interaction fragment $sd_\emptyset$ is mapped into an empty graph as follows:

$[\![sd_\emptyset]\!] = \langle \mathcal{O}, \emptyset, \emptyset, \{s_0\}, s_0, \emptyset\rangle$ where $\mathcal{O}$ is the same collection of participants in sd and s_0: $O \longrightarrow \{\epsilon\}$ associates to each participant in $\mathcal{O}$ an empty sequence of events.

4.5.6 Operator *par*

Let sd be interaction fragment combined from two interaction fragments sd_1 and sd_2 by means of the parallel operator *par*. We realize here an interleaving between all

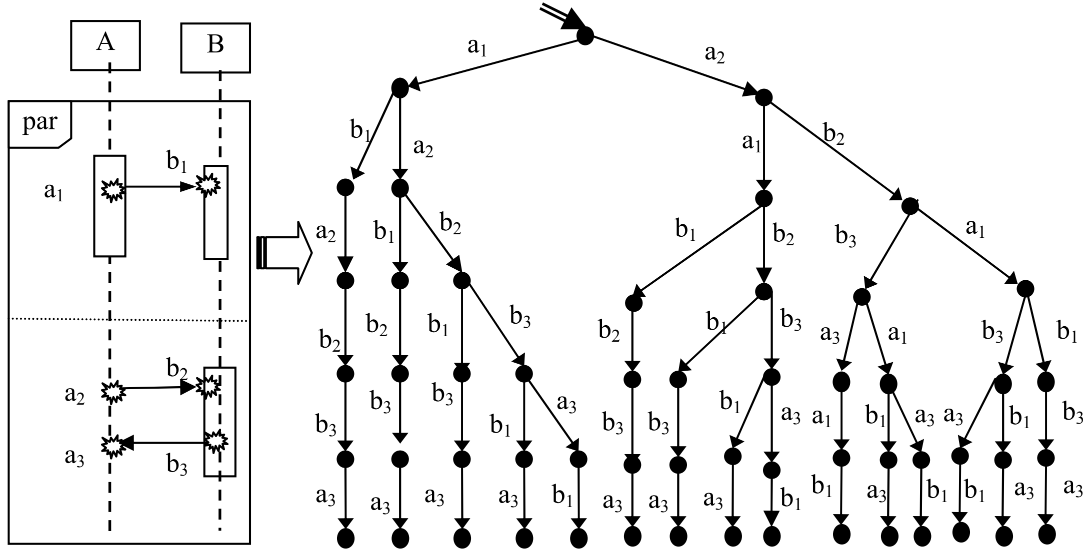


Figure 4.7: The graph related to an interaction fragment with *par* operator.

sequences occurring in diagrams (see Fig.4.7) without adding further orderings.

$[\![sd_1 \ par \ sd_2]\!] = [\![sd_1]\!] \otimes_{Prior} [\![sd_2]\!]$ where $Prior = \emptyset$.

4.5.7 Operator of strict sequencing seq_s

Let sd be an interaction fragment combined from two interaction fragments sd_1 and sd_2 by means of the strict sequencing operator seq_s.

Similarly to sequential composition in process algebra, the semantics of strict sequencing defines a strict ordering of the operands on the first level. Therefore, the former operand should first be carried out entirely and after that only, the second one is executed. That is why, all events occurrences of the first interaction fragment are granted more priority over those of the second one.

$[\![sd_1 \ seq_s \ sd_2]\!] = [\![sd_1]\!] \otimes_{Prior} [\![sd_2]\!]$ where $Prior = \Sigma^1 \times \Sigma^2$.

We carry out, in this case, concatenations between paths belonging respectively to the two diagrams sd_1 and sd_2 such that events occurrences from the first diagram always occur before those of the second diagram.

4.5.8 Operator of weak sequencing seq_w

Let sd be an interaction fragment combined from two interaction fragments sd_1 and sd_2 by means of the weak sequencing operator seq_w.

The weak sequencing expresses three properties [103]:

- The ordering of events occurrences within each of the operands are maintained

in the result.

- Occurrence specifications on different lifelines from different operands may come in any order.
- Occurrence specifications on the same lifeline from different operands are ordered such that an event occurrence of the 1st operand comes before that of the 2nd one.

Hence, the weak sequencing reduces to a parallel merge between events occurrences on different lifelines but restricted by strict sequencings among events occurrences belonging to same lifelines (thanks to the set $Prior$ of added precedence relations). Formally, $[\![sd_1\ seq_w\ sd_2]\!] = [\![sd_1]\!] \otimes_{Prior} [\![sd_2]\!]$
where $Prior = \{(e, e') \in \Sigma^1 \times \Sigma^2 \mid e, e'$ belong both to the same lifeline(o)$\}$.

4.5.9 Operator *loop*

Let sd be a combined fragment from an interaction fragment sd_1 by means of the loop operator *loop* parameterized by a guard G given as un integer $\in \{min \dots max\}$.

The loop operator would be repeated a given number of times as long as the guard is fulfilled. $[\![loop(G, sd)]\!] = ([\![(sd\ seq_s\ loop(G-1, sd)]\!]$.

However this solution does not pay attention when iterating to the evaluation event of the loop guard. So we have to add τ-transitions to record this internal choice.

$[\![loop(G, sd)]\!] = ([\![(sd\ seq_s\ sd_\tau)\ seq_s\ loop(G-1, sd)]\!]$.

The graph of τ-interaction fragment sd_τ is: $[\![sd_\tau]\!] = \langle \mathcal{O}, \{\tau\}, \emptyset, \{s_0, s_1\}, s_0, \{s_0 \overset{\tau}{\vdash} s_1\}\rangle$.

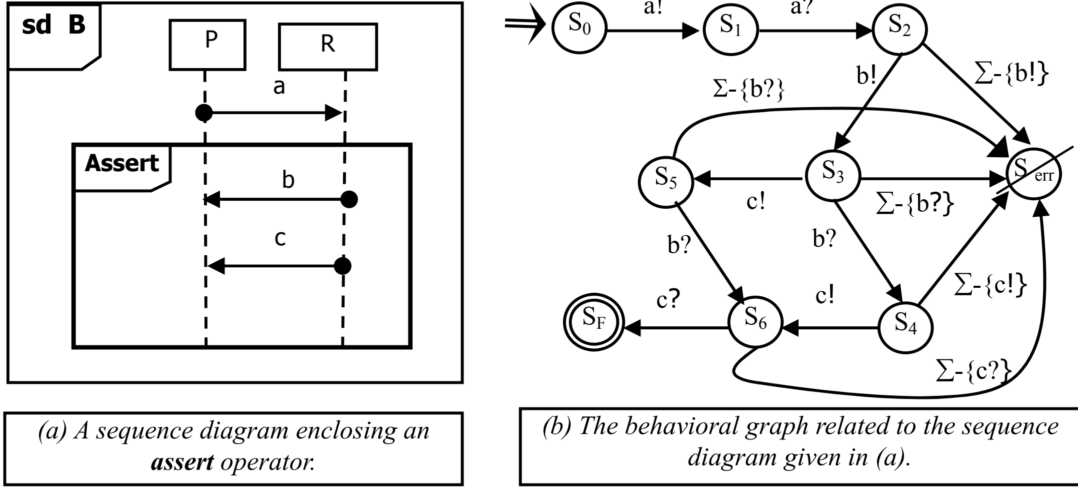
When the number of iterations is undefined ($max = \infty$), the correct solution consists in using the star operation on traces with adorning loop transitions with τ which models guard evaluation (see Fig.4.3). So, the related graph should be:

$[\![loop(G, sd)]\!] = Star([\![sd]\!])$

Remark 3 we prefer to use strict sequencing rather than weak one to avoid a pathological case of divergence in loop combination along with an asynchronous communication

4.5.10 Operators *neg* and *assert*

Note that the operators *neg* and *assert* need a particular handling. Indeed, the operator *neg* when wrapping an interaction fragment, means that all sequences of its enclosed fragment are forbidden. That is, the system has to discard them (see Fig.4.9(a)). Conversely, the operator *assert* specifies that sequences of its enclosed fragment are



(a) A sequence diagram enclosing an **assert** operator.

(b) The behavioral graph related to the sequence diagram given in (a).

Figure 4.8: A Sequence diagram with *assert* operator.

the only valid continuation. Namely, all sequences other than those specified by the enclosed fragment are forbidden (see Fig.4.8(a)).

Instead of trace based models, we still opt to tackle these two operators using our behavioral graphs which preserve branching time structures. However, for depicting the forbidden sequences, we need to enhance our specification graph with the concept of erroneous states such that every path containing or ending into one of these particular states, has to be considered as illegal and forbidden sequence. As expected, we consider that any system implementation is error-safe when the erroneous states are unreachable.

For instance, Fig.4.8(b) illustrates the behavioral graph of the sequence diagram of Fig.4.8(a) enclosing an *assert* operator. It depicts the only allowed sequences by forcing the passage of control into erroneous states through transitions labeled with any action not in the set of allowed ones, as specified in the *assert* fragment. On the other hand, Fig.4.9(b) illustrates the behavioral graph of the sequence diagram of Fig.4.9(a) enclosing an interaction fragment under the scope of a *neg* operator. The relating path in the graph consists thus of erroneous states depicting the fact that it is forbidden.

Formally, let $[\![sd]\!] = \langle Q, \Sigma, \hookrightarrow, q_0, \mathcal{F}, \mathcal{E} \rangle$. Then,

- $[\![neg(sd)]\!] = \langle Q, \Sigma, \hookrightarrow, q_0, \mathcal{F}', \mathcal{E}' \rangle$ where $\mathcal{F}' = \emptyset$ and $\mathcal{E}' = Q \setminus \{q_0\}$.
- $[\![assert(sd)]\!] = \langle Q', \Sigma, \hookrightarrow \cup \hookrightarrow', q_0, \mathcal{F}, \mathcal{E}' \rangle$ where $Q' = Q \cup \{q_{err}\}$, $\mathcal{E}' = \mathcal{E} \cup \{q_{err}\}$, $\hookrightarrow' = \{(q, a, q_{err}) \mid q \in Q \wedge a \in \Sigma \wedge \nexists q' \in Q, (q, a, q') \in \hookrightarrow\}$.

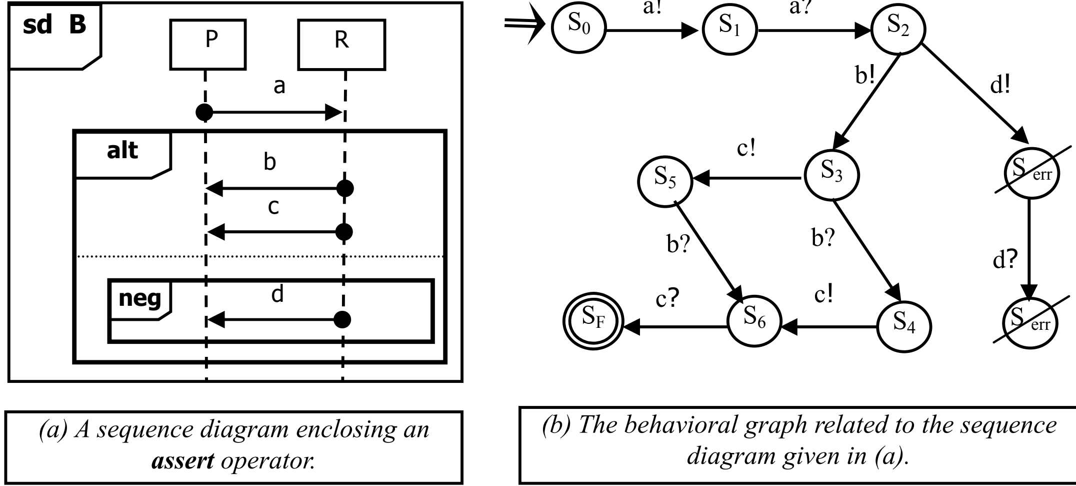


Figure 4.9: Specification of a train control system.

4.6 Extraction of timing information

The sequence diagram in Fig.4.10 shows how time and timing notations may be applied to describe time observation and timing constraints [103]. The "User" sends a message "Code" and its duration is measured. The "ACSystem" will send two messages back to the "User". "CardOut" is constrained to last between 0 and 13 time units. Furthermore, the interval between sending of Code and the reception of "OK" is constrained to last between d and $3 * d$ where d is the measured duration of the "Code" signal. We also notice the observation of the time point t at the sending of "OK" and how this is used to constrain the time point of the reception of "CardOut".

Our approach consists in extracting time formulas over logical clocks from the time annotations in sequence diagrams. Then, we adorn related nodes and transitions in our graph by these timing conditions in a similar way to timed automata [1].

Definition 8 *(timing constraint) Let χ be a finite set of clocks ranging over $\mathbb{R}^{\geq 0}$ (set of non negative real numbers). The set $\Psi(\chi)$ of timing constraints on χ is defined by the following syntax: $\psi ::= true|x \ll c|x - y \ll c|not(\psi)|\psi \wedge \psi$ where $x, y \in \chi$, $c \in \mathbb{R}^{\geq 0}$ and $\ll \in \{<, \leq\}$. Other assertions such as, $x > 3$, $2 \leq x < y + 5$, $\psi \vee \psi$ can be defined as abbreviations.*

4.6.1 Enhancing graphs with timing constrains

We add two mappings δ_1, δ_2 as follows:

- $\delta_1 : Q \longrightarrow \Psi(\chi)$

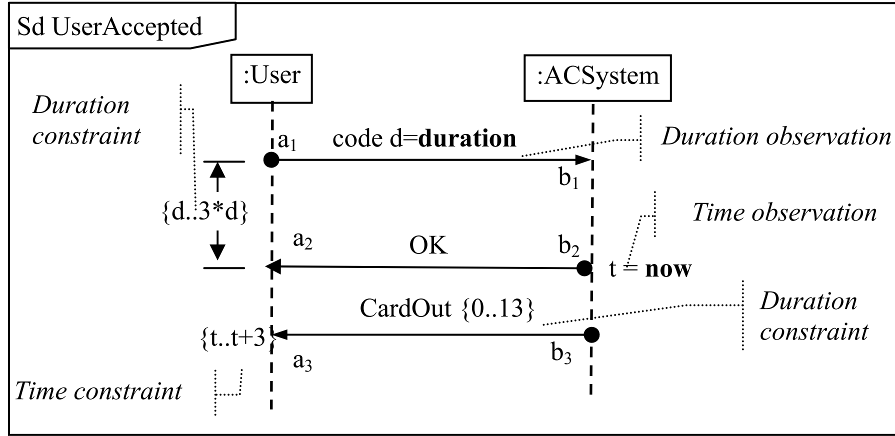


Figure 4.10: Sequence Diagram with timing concepts.

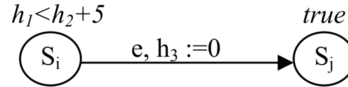


Figure 4.11: Timed graphs.

- $\delta_2 : (\hookrightarrow) \longrightarrow 2^{\chi}$

The first mapping δ_1 assigns to each node a condition called activity *condition* which may be *true*. The second mapping δ_2 associates with each transition a set of clocks initializations which may be empty.

The behavior of the new timed graph becomes as follows:

The control could stay in a node s_i (see Fig.4.11) while the constraint $\delta_1(s)$ is fulfilled but once $\delta_1(s)$ becomes $false$ we should leave s_i by execution of an instantaneous event occurrence e (an event occurs with no duration [103]). It's obvious that the control could stay indefinitely in s_j (Fig.4.11) if its activity condition is *true*.

When a transition t occurs, all clocks ($h_i \in \delta_2(t)$) are reset to zero. These clocks therefore start measuring time progress since this point but may be used later at different instants.

4.6.2 Extracting time constrains from sequence diagrams

Time observations, timing constraints are related to points on the lifelines of the sequence diagram. These points are the instances at which send or receive events occur. Likewise, duration observations or constraints are related to messages, each one of them is related to two events on the same lifeline or on two different lifelines.

The main idea of our approach to handling time constraints is to generate a logical clock "h" at any related time observation point "t". Any outgoing arc from this point

will be adorned with initialization of the clock "h" making it possible to count time progress from this starting point.
For a time constraint of the form "$t + a \dots t + b$", we search out in our graph the set of nodes of which outgoing transitions are labeled with the event related to this constraint. Every such a node should receive a timing constraint of the form $a \leq h \leq b$.

```
Algorithm Extract_time & Duration_constraints
In:  sd :  Sequence Diagram; G :  Graph ;
Out:  G' :  a timed graph;
Begin
 χ := ∅;
 For each s ∈ S do δ1(s) = true;
 For each time observation t at an event occurrence e
 Do Begin
      Generate a new clock h;
      χ := χ ∪ {h};
      // h measures time progress since the observation point.
      Find out the set A of transitions labeled with e;
      For each t ∈ A do δ2(t) = δ2(t) ∪ (h, 0);
      For each time constraint c of the form {t + a ... t + b} on event occurrence e';
      Do Begin
          Find out the set N of nodes whose outgoing transitions are labeled with e';
          For each s ∈ N do δ1(s) = δ1(s) ∧ h ≥ a ∧ h ≤ b;
         End
   End
```

Likewise, for handling duration constraint, we generate two logical clocks; "h_1" at start point and "h_2" at final point related to duration observation "d". Any outgoing arc from these points will be tagged with initializations of related clocks so that the difference between their values $(h_1 - h_2)$ gives later the duration between the two events.

For any duration constraint of the form "$a(d) \dots b(d)$" between two events e and e', we add in a similar way a clock h_3 related to the first event e for counting the time progress since this first point. Next we search out in our graph the set of nodes of which outgoing transitions are labeled with the second event e'. Every such a node should then receive a timing constraint of the form $a(h_1 - h_2) \leq h_3 \leq b(h_1 - h_2)$.

```
     For each duration observation d on a message m
 Do Begin
      Let e be the event occurrence related to sending (m);
```

```
Let e' be the event occurrence related to receiving (m);
Generate two new clocks h1 and h2;
χ := χ ∪ {h1, h2};
// h1 starts at the sending moment of m.
Search out the set A of transitions labeled with e;
For each t ∈ A do δ2(t) = δ2(t) ∪ (h1, 0);
// h2 starts at the receiving moment of m.
Find out the set B of transitions labeled with e';
For each t ∈ A do δ2(t) = δ2(t) ∪ (h2, 0);
For each duration constraint c of the form {a(d)...b(d)}
        between two events (e”, e”') or on a message m'
Do Begin
    Let e” be the first event occurrence or the sending event of m';
    Let e”' be the second event occurrence or the receiving event of m';
    Generate a new clock h3;
    χ := χ ∪ {h3};
    // h3 measures time since the occurrence of e”
    Find out the set A of transitions labeled with e”;
    For each t ∈ A do δ2(t) = δ2(t) ∪ (h3, 0);
    Find out the set N of nodes whose outgoing transitions are labeled with e”'
    For each s ∈ N do δ1(s) = δ1(s) ∧ h3 ≥ a(h1 − h2) ∧ h3 ≤ b(h1 − h2);
   End
End

For each duration constraint c of the form {a...b} between

        two events occurrences (e, e') or on a message m

Do Begin

    Let e be the first event occurrence or the sending event of m;

    Let e' be the second event occurrence or the receiving event of m;

    Generate a new clock h; χ := χ ∪ {h};

    Find out the set A of transitions labeled with e;

    For each t ∈ A do δ2(t) = δ2(t) ∪ (h, 0);

    Find out the set N of nodes whose outgoing transitions are labeled with e';

    For each s ∈ N do δ1(s) = δ1(s) ∧ h ≥ a ∧ h ≤ b;

End

End Extract_time & Duration_constraints.
```

At last, we notice that the above approach can be also extended in straight way to handle other possible cases of time constraints.

The timed graph related to the diagram of Fig.4.10 is illustrated by Fig.4.12.

4.7 Conclusion

We have given, in this chapter, a formal semantics for UML 2 sequence diagrams by using a faithfully branching time structure rather than traces. The implementation

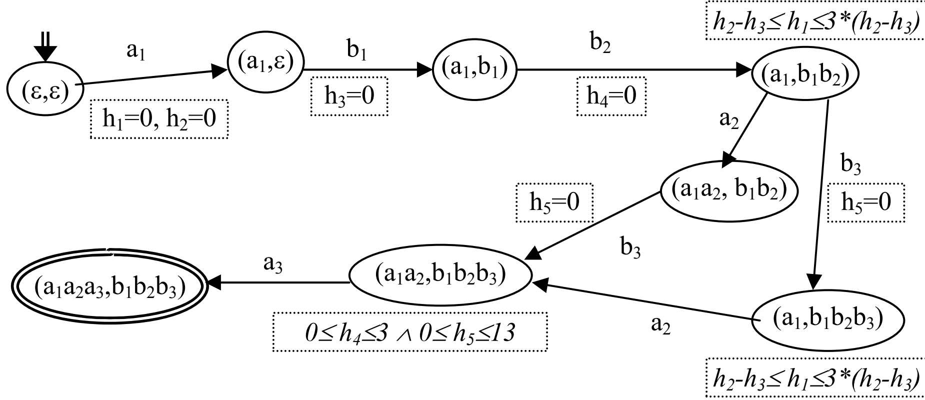


Figure 4.12: The timed graph related to the sequence diagram depicted by Fig.4.10

model (a lattice-like graph) records both traces of all interaction components together with branching bifurcations and can be directly unfolded into a transition system capturing the intended behavior. The graphs related to interaction fragments are equipped with few generalized algebraic operations which help us define the formal semantics of all interaction operations in a compositional manner. Moreover, we have proposed a method to extract time properties of UML interactions into time constraints we add to our graph in order to enable timeliness and performance analysis.

Hence, resulting graphs modeling valid and invalid behaviors could be now compared to the state diagram to achieve semantically and temporal consistencies checking.

Chapter 5

Consistency Checking of Dynamic Views

This chapter deals with the consistency checking of UML sequence and state diagrams standing for, respectively, specification and implementation models of reactive systems .

5.1 Introduction

At pointed out in previous chapters, the Unified Modeling Language (UML) includes notations for scenarios-based specification called sequences diagrams [103]. Indeed, these notations are widely used for describing design requirements since these allow designers to express in an intuitive and visual way how system components (seen as black or grey boxes) have to interact and communicate in order to provide system level functionalities. Hence, a sequence diagram of the system under design can be seen as its specification model (or a piece thereof) depicting an expected interaction between its components and showing the messages that may be dispatched among them. Conversely, UML uses each of the separate state machines (i.e, UML StateCharts) to focus on describing the internal behavior of only one component in such a way that the execution of these state machines altogether, correctly realize the specified interactions of the sequence diagrams. That is, the state diagram consisting of individual state machines could be considered as the implementation model of the system under design, while its sequence diagrams constitute its specification.

It should be noticed that such a step is error-prone since it is manually achieved. Consequently, the key issue to support the design of reliable software systems (particularly safety-critical ones) is to be able to check that their sequence diagrams are faithfully synthesized into state machines which have to correctly implement the ex-

pected interactions. In other terms, designers have to check the semantics consistency of UML dynamic models composed of state and sequence diagrams.

In this chapter, we present our method [49, 51, 52, 53, 54] for automating the process of checking wether the design model of a reactive system behaves according to the collection of scenarios specified by its sequence diagrams (see Fig.5.1). This process shall also test the absence of deadlocks and forbidden scenarios.

In view of the fact that UML diagrams lack for formal semantics, we make use of a formalization method defined in [48] to translate positive (resp. negative) sequence diagrams into transition systems which depict all the allowed (resp. forbidden) computations of the system and preserve the branching structure. Similarly, we use the formalization method defined in [47], which translates the state diagram into an enhanced modular Petri net, whose reachability graph represents the behavioral (implementation) graph of the system under design.

We propose then an exhaustive checking method using an adequate comparison criterion which copes well with the features of sequence diagrams depicting only a subset of allowable and/or forbidden scenarios. The method consists in proving that any computation of the implementation graph either is within allowable scenarios or does not lead to erroneous statuses of the specification graph related to the sequence diagram. Indeed, the system should avoid erroneous states because these depict forbidden computations specified by interaction fragments under the scope of *neg* or *assert* operators (see Subsection 4.5 for more details).

However, such a global checking process often suffers from the state explosion problem when tackling medium and large systems. Moreover, one may need only to check that some new component could be safely substituted for the previous one without doing (from the scratch) the consistency checking of all the system that has been previously proven.

Consequently, we propose in a second part of the chapter to thoroughly revise and extend our component based approach given in [54, 53, 52] to address the aforementioned issues. Herein, we do not require that UML state machines implement all scenarios since these are not mandatory (as this was considered in [49]). Furthermore, we deal with sequence diagrams which may contain interaction operators *neg* and *assert*. Hence, the checking process uses now an appropriate comparison relation over the behavioral models enhanced with the concept of erroneous statuses.

The other contributions of our work are explained throughout the steps of our modular checking process summarized as follows (see Fig.5.1):

- Given a component we would like to check, we make use of the formalization method of [47] in a modular fashion to extract its specification graph by consid-

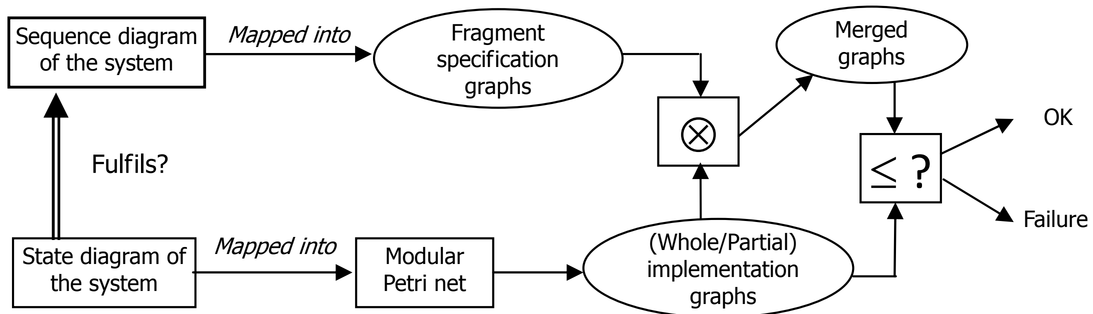


Figure 5.1: The approach for consistency checking of UML dynamic diagrams.

ering only sequence diagrams including our component as participant. Besides interaction operators considered in [47], we deal in this chapter with *neg* and *assert* operators thanks to the concept of erroneous states.

- Similarly, we give a modular method to extract from the whole state diagram the partial behavioral graph in relation to the component [49].

- The two previous steps yield, thus, partial specification and implementation graphs which shall encompass, respectively, intended and realized interactions of this component with its environment. Afterwards, we give a fitting method which uses these behavioral graphs to incrementally check both the compatibility between the system components, and the consistency between each component state machine and all sequence diagrams where this component is involved.

- Our method for consistency checking copes well with sequence diagrams that can only specify partial allowable and/or forbidden scenarios (and not mandatory ones as in LSC [28]). Indeed, we use a fitting observational comparison criterion based on synchronization and hide operators over these graphs in order to check that each computation in the implementation graph, does either belong to the set of allowable sequences of the specification graph (reaching thus a final status) or not lead the system to a deadlock or erroneous status according to this specification graph.

- Such a component based checking approach can be gradually used to achieve the consistency checking of the whole state diagram of the system with its comprehensive interaction diagrams.

The remainder of the chapter is organized as follows: Section 5.2 presents more details about the motivation of our approach and the related work. Section 5.3 recalls specification and implementation graphs related to sequence and state diagrams and

presents a running example. Then, it explains our exhaustive approach to deal with the consistency checking. Next, Section 5.4 gives an algorithm to partially explore the Petri net related to the state diagram to yield the partial graph related to one component. Then, the compatibility issue related to composition of components, along with the modular consistency checking, are explained and discussed both in the interleaving and true concurrency contexts. Finally, a conclusion is given in Section 5.5.

5.2 Modular versus exhaustive consistency checking

Once the entire state diagram of the system is translated into a Petri net, one could check the validation of the dynamic view by comparing its reachability (i.e, implementation) graph against specification models derived from sequence diagrams. The exhaustive exploration of the state space of any system may be performed by means of any classical enumeration algorithm. However, construction and exploration of such graphs for complex systems mostly raise the state explosion problem. Moreover, we frequently need just to check whether a new component can be substituted to another one without introducing bugs like deadlock or new unsuitable behaviors.

Therefore, since we advocate the use of a modular approach, we propose a method for partial exploration of the reachability graph with respect to some main component. We generate only partial paths of the system consisting of actions[1] belonging to this component, besides all of their causally linked actions from other cooperating components. In fact, as components' state machines have often to asynchronously collaborate for their progress, the performance of the latter actions is necessary to unlock the progress of the main component by providing it with the needed triggering events which could enable the former actions of this main component. These are called causally linked actions and constitute the "common glues of composition" that allow us, later, to combine components and check their compatibility.

Note that the generation of partial graphs more compact than the whole state space, is a practical means often used to avoid the state explosion problem. Thus, besides work on-the-fly checking [68], various approaches such as partial order based techniques [7, 59, 94, 123] were addressed to explore some parts of reachability graphs so that some properties could be checked or preserved. Whereas these approaches try to exploit structural (symmetric or modular) characteristics of formal synchronous models, our exploration method is component based and can be applied to synchronous models as well as asynchronous models based on event triggered actions.

Furthermore, our component-based method is generalized to achieve the compatibility and consistency checking of components. We show hence, using a synchroniza-

[1] An action has to be seen as an instantaneous event.

tion product and branching simulations over partial implementation and specification graphs, how to check the compatibility between the cooperating components (i.e. absence of deadlocks and erroneous statuses owing to their composition). Moreover, this method allows verifying, in a modular way, that sequence diagrams are realizable by state machines. In fact, in addition to generating partial reachability graph with relation to some component, our approach consists in extracting and generating its specification model from sequence diagrams where the component is involved in, as a participant. The resulting graph should distinguish between allowable and forbidden sequences due to the use of *neg* and *assert* operators.

Thereafter, we check whether the partial reachability graph of a component matches its specification graphs using a fitting comparison means based on a *branching simulation* because sequence diagrams depict only partial views of allowable (i.e., intended) behaviors of the system under design.

Obviously, the main benefit of such an approach is to allow us achieving a modular and parallelizable checking of consistency and compatibility of UML artifacts at early stages of the development process. In contrast to our modular approach that copes with both allowable and forbidden sequences due to *neg* and *assert* operators, other papers propose approaches that see sequences as mandatory and neglect the fragment style of UML diagrams for their analysis.

For instance, the authors of the papers [85] and [75] propose mappings of UML specifications made up of sequence and state diagrams into π-calculus processes and then use bisimulation based equivalences (instead of preorder relations) to compare π-expressions and check coherence of related diagrams.

Other approaches aim at mapping UML models into input languages of model checking tools. However, only subsets of UML features are considered and the properties we want to check have to be expressed in high level logics. For instance, the approach in [17] promotes the use of Spin tool to deal with automated consistency checking of a subset of UML diagrams whereas in [114] class, sequence and state diagrams are transformed into description logic for checking consistency between different versions of these diagrams.

Likewise, the authors of [35] describe experiments of using their tool RuleBase for static model checking UML models. This tool automatically converts UML models to the input for RuleBase, allows user to specify constraints graphically using a variation of sequence diagrams, and presents model checking results as sequence diagrams consisting of states and events in the original UML model. In [40] the authors present a framework to address the consistency aspects of UML/SPT models including syntactic, semantic, concurrency-related and time consistency. The framework uses schedulability analysis for checking time consistency between statecharts and sequence diagrams.

The approach of [34] presents an UML/Analyzer tool to treat consistency rules as black-box entities and observes their behavior during their evaluation to identify what model elements they access. Similarly, the approach of [33] verifies the consistency of entire sequence and state diagrams using dynamic meta-modeling (DMM) rules.

Lastly, an algorithmic approach is given in [87] to a consistency check between UML sequence and state diagrams. However, this approach lacks for a formal semantics model which is needed for precise and intensive handling of the problems such that those related to allowable/forbidden sequences of UML 2 sequence diagrams.

5.3 Exhaustive consistency checking of dynamic diagrams

5.3.1 Formalization of sequence and state diagrams

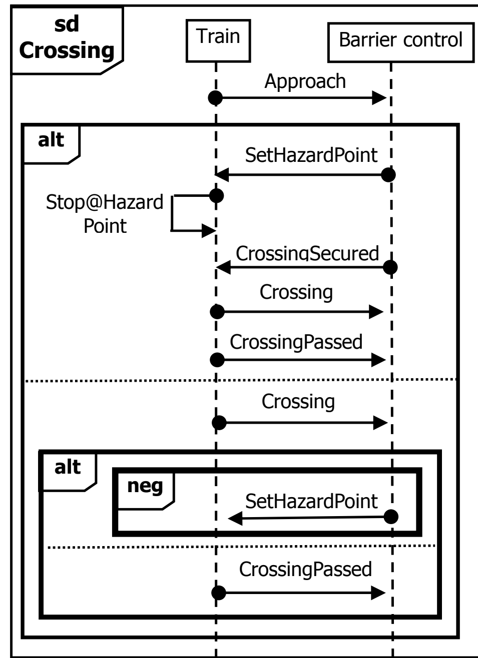
Recall that in order to achieve the consistency checking of dynamic diagrams, these have to be mapped into labeled transition systems $G = \langle Q, \Sigma, \hookrightarrow, q_0, \mathcal{F}, \mathcal{E} \rangle$ where:

- Q is the set of states with some initial one q_0,
- Σ is the set of events,
- $\hookrightarrow \subseteq Q \times \Sigma \times Q$ is the set of transitions labeled with events whose occurrences cause changes of the current state,
- $\mathcal{F} = \{q \in Q | \forall a \in \Sigma, \nexists q \xrightarrow{a}\}$ is the set of successful termination states[2],
- and $\mathcal{E}$ is the set of erroneous states which depict forbidden computations.

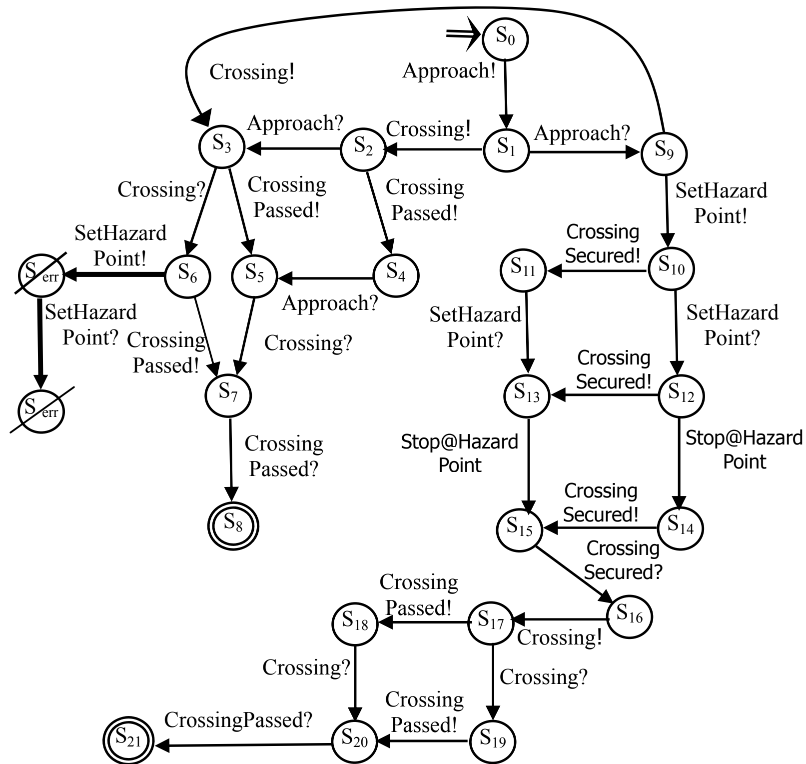
Implementation (i.e., reachability) graphs need, as expected, to be augmented with the concepts of erroneous and final states as follows:
- $\mathcal{F}$ is computed by inserting therein all markings (i.e., states) without successors,
- and $\mathcal{E} = \emptyset$ since these graphs cannot encompass forbidden computations.

Example1: To clarify our approach, we present in Fig.5.2(a) the specification of a train control system which aims at providing a safe, efficient and smooth train traffic across some borders. Before passing a crossing of a road and a railway track, the train should inform the barrier control of its passage by sending an alert message "*approach*" to the barrier control. Then, this component can either let the train pass the crossing or send to the train a "*setHazardPoint*" message to order it to stop before entering the crossing area. In such a case, the train should stay stopped until it receives another request "*CrossingSecured*" for resuming its passage through the crossing area. Notice

[2] Any sink state (without successors) which does not belong to $\mathcal{F}$ would be considered as a deadlock state.



(a) The sequence diagram of a train control system.



(b) The behavioral graph related to the sequence diagram given in (a).

Figure 5.2: Specification of a train control system.

that this sequence diagram contains a forbidden scenario (via a *neg* operator) stating that the barrier control should never send, too late, a "*SetHazardPoint*" message which lets the train stop while it has begun its passage through the crossing area.

The behavioral graph of this sequence diagram is given in Fig.5.2(b). Note that forbidden scenarios are translated into paths ending into erroneous states while allowable scenarios are mapped into paths consisting of error-safe states, some of which may be final. Also, some of the implied sequences (e.g., *approach!; crossing!; Approach?; ...*) are not intended scenarios and illustrate one of the possible results of visual misinterpretations which can be introduced due to the imprecise semantics of sequence diagrams. For instance, the previous trace is due to the adoption of the order among events instead of messages as someone might think of. However, these unintended transitions can be discarded by using a strict sequencing operator between the "*approach*" message and the rest of the diagram. We will explain later, how to detect such shortcomings by checking the compatibility and consistency of behavioral graphs related to sequence and state diagrams.

We present in Fig.5.3 two simplified state machines related to the components of the train control system. We then try to check whether the two components always evolve without leading to irregular situations. Fig.5.4 presents two Petri net modules related to the two state machines of the system. The two modules are linked by means of some appropriate "glue" consisting of intermediate places and transitions to convey triggering events between them. We give, in Fig.5.5, only a fragment of the whole state space of the Petri net modeling the train control system. Some of its computations are present on the sequence diagram of Fig.5.2(a) while others are not specified therein. Unfortunately, it can reach erroneous states through sequences (i.e., red upward arrows) that the sequence diagram specifies as forbidden.

5.3.2 Exhaustive consistency checking of dynamic diagrams

Given some system *Sys* under design, we want to check that its state diagram *semantically fulfills* sequence diagrams. That is, the state machines of its components faithfully realize the intended interactions of sequence diagrams. Recall that to enable such a checking process, sequence and state diagrams have to be translated into behavioral graphs (i.e., labeled transition systems) called respectively specification and implementation graphs. Afterwards, one can proceed to the consistency checking of the dynamic diagrams of the system by comparing these specification and implementation graphs.

To this purpose, we need to define a comparison criterion that makes it possible to check that the state diagram of the system *Sys* semantically fulfills a sequence (or

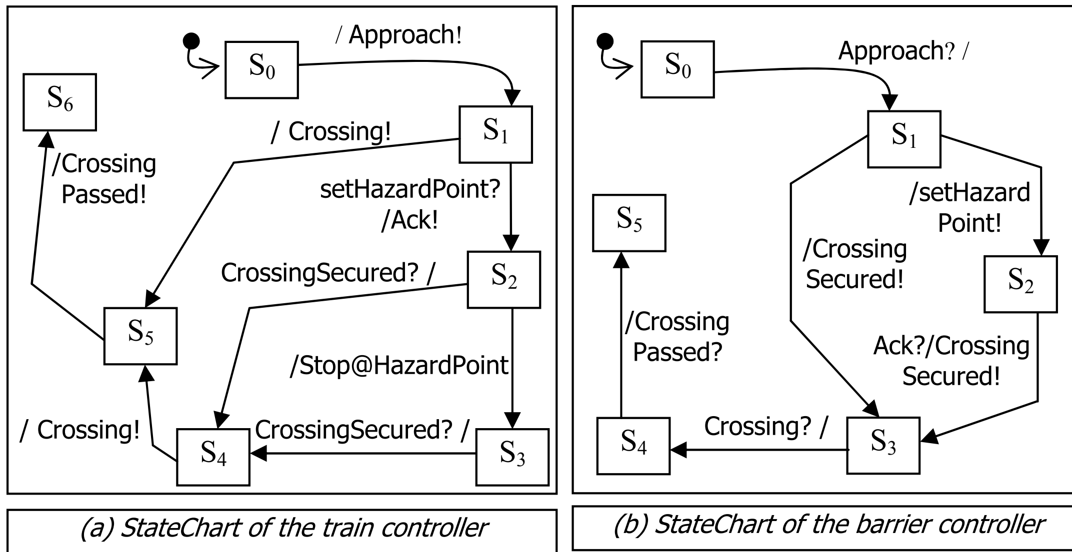


Figure 5.3: UML State machines of the train control system.

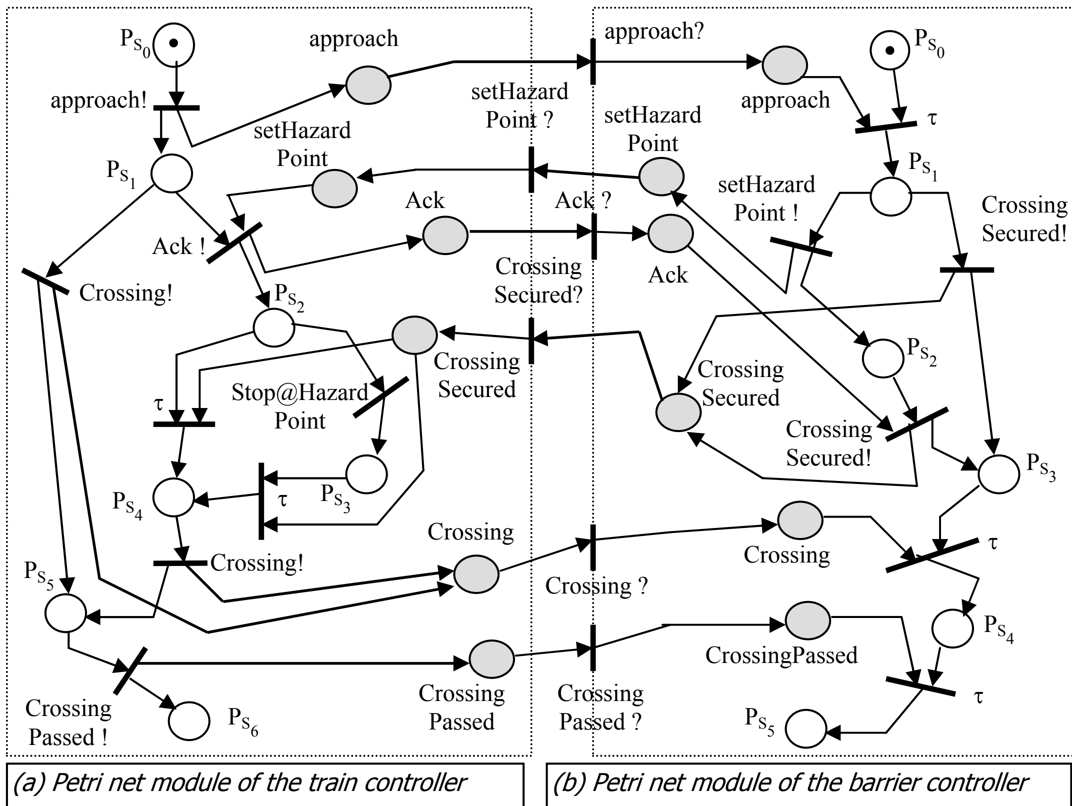


Figure 5.4: Translation of the state diagram into a modular Petri net
The symbol τ denotes silent actions representing internal control moves.

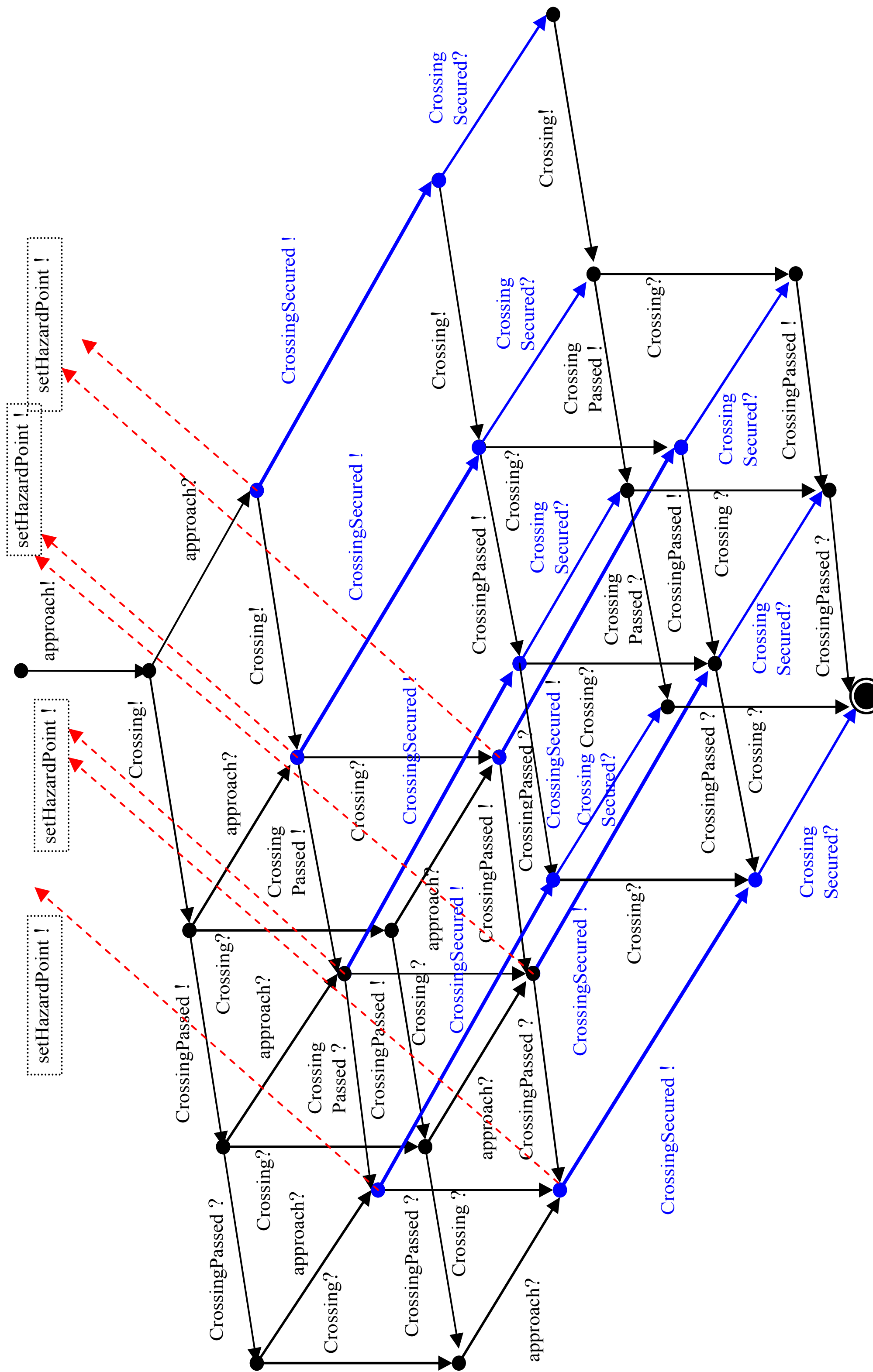


Figure 5.5: Whole state space of the train control system.

an overview interaction) diagram *sd* (denoted by $Sys \vDash sd$).

It should be noted that only a subset of requirements are depicted in terms of scenarios which elicit no more than a subset of prefixes of the computational traces that a state machine can carry out. Hence, our definition shall ensure that components involved in a sequence diagram will correctly cooperate to achieve some of its intended interactions, but not necessarily all of them.

That is why we could not use the following statement for defining the fulfillment relation: "$Sys \vDash sd$ iff $[\![sd]\!] \preceq [\![Sys]\!]$". This would amount to consider the state diagram as valid only if simulates all the valid interactions specified by the sequence diagram, as would be the case if all its scenarios are mandatory. Moreover, such a definition would require the system implementation to simulate forbidden sequences derived from enclosing interaction fragments under the scope of the operators *neg* or *assert* in the specification graph.

Therefore, we propose a new method to overcome the aforementioned problem for achieving the consistency checking of state and sequence diagrams of our system. This consists in proving that the synchronization product (i.e., merging) of their behavioral graphs can be simulated by the behavioral graph of the state diagram. Indeed, such a verification process allows us to handle, within the branching time semantics, both forbidden and allowable scenarios (i.e., valid interactions that are not mandatory).

This approach considers the system properties as a process (i.e., the specification graph) which is combined with the system model (i.e., implementation graph) by means of a synchronization (i.e., merging) operator. Then, the resulting graph has to be related, via a simulation preorder, to the implementation graph. The simulation relation amounts to assert that the combined graph encompasses a part or all of only valid computations we can derive from a sequence diagram. Since the implementation graph does not contain erroneous states, it simulates only these valid computations which would be available as prefixes of or interleaved with sequences computed from the related state diagram, along with the preservation of their branching structures, too.

In other terms, the implementation graph correctly cooperates with the specification graph when yielded sequences (in the combined graph) do not lead to deadlock or erroneous situations. The first situation may happen when the former is unable to synchronize on the performance of some common action with the latter. On the contrary, the second case may occur when the former offers a common (synchronized) action that lead the former to an erroneous state belonging to a forbidden path.

The underlying comparison relation of our method is given below. That is, a tailored version of simulation [130], denoted by the symbol $\preceq$.

Definition 9 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two labeled transition systems: $\mathcal{A}_i = \langle Q_i, \hookrightarrow_i, \Sigma, q_{0_i}, \mathcal{F}_i, \mathcal{E}_i \rangle$ for $i = 1, 2$. A simulation is a relation $\mathcal{R} \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2, p\mathcal{R}q$ if and only if:*

- $\forall a \in \Sigma, \forall p' \in Q_1 : p \overset{a}{\longrightarrow} p' \implies \exists q' \in Q_2 : q \overset{a}{\longrightarrow} q' \wedge p'\mathcal{R}q'$.
- $p \in \mathcal{F}_1 \implies q \in \mathcal{F}_2$ *(i.e., successful termination states, on one side, have to be declared as final on the other side, to distinguish between them and deadlock statuses).*
- $p \in \mathcal{E}_1 \implies q \in \mathcal{E}_2$ *(i.e., erroneous states that identify forbidden sequences on one side, have to be also erroneous on the other side).*

$\mathcal{A}_2$ simulates $\mathcal{A}_1$ ($\mathcal{A}_1 \preceq \mathcal{A}_2$) if their initial nodes are related by some simulation $\mathcal{R}$.

Next, we give the definition of our combination operator between implementation and specification graphs. Let $\mathcal{A}_{Impl}$ denote the implementation graph $\langle Q_{Impl}, \hookrightarrow_{Impl}, \Sigma_{Impl}, q0_{Impl}, \mathcal{F}_{Impl}, \emptyset \rangle$ issued from the translation of the state diagram of the system and let $\mathcal{A}_{Spec}$ denote the specification graph $\langle Q_{Spec}, \hookrightarrow_{Spec}, \Sigma_{Spec}, q0_{Spec}, \mathcal{F}_{Spec}, \mathcal{E}_{Spec} \rangle$ issued from the translation of its sequence diagram.

Definition 10 *A synchronization product $\otimes_G$ of two graphs $\mathcal{A}_{Impl}$ and $\mathcal{A}_{Spec}$ yields a new transition system $\mathcal{A} = \langle Q, \hookrightarrow, \Sigma, q_0, \mathcal{F}, \mathcal{E} \rangle$ such that:*

- $Q = Q_{Impl} \times Q_{Spec}$ *with* $q_0 = (q0_{Impl}, q0_{Spec})$.
- $\Sigma = \Sigma_{Impl}$ *(since $\Sigma_{Spec} \subseteq \Sigma_{Impl}$).*
- $\hookrightarrow = \{(p_1, q_1) \overset{a}{\longrightarrow} (p_2, q_2) | a \in (\Sigma_{Impl} \cap \Sigma_{Spec}) \wedge (p_1 \overset{a}{\longrightarrow} p_2) \in \hookrightarrow_{Impl} \wedge (q_1 \overset{a}{\longrightarrow} q_2) \in \hookrightarrow_{Spec}\} \cup \{(p_1, q_1) \overset{a}{\longrightarrow} (p_2, q_1) | a \in \Sigma_{Impl} \wedge p_2 \in Q_{Impl} : (p_1 \overset{a}{\longrightarrow} p_2) \in \hookrightarrow_{Impl} \wedge \nexists q_2 \in Q_{Spec} : (q_1 \overset{a}{\longrightarrow} q_2) \in \hookrightarrow_{Spec}\}$.
- $\mathcal{F} = \mathcal{F}_{Impl} \times \mathcal{F}_{Spec}$.
- $\mathcal{E} = Q_{Impl} \times \mathcal{E}_{Spec}$.

According to the synchronization operator, wherever the two graphs are able to individually achieve a same action throughout their evolution, these have to synchronize its performance in the combined graph. However, when the implementation graph can carry out an action which the specification graph does not offer at that point of its progress, we allow the combined graph to perform this action since the sequence diagram depicts only a fragment of possible interactions and not all of them. Conversely, the actions that the specification graph may offer but the implementation graph could

not carry out, shall not be inserted in the merged graph since the UML state machines may implement only a part of the valid interactions of the sequence diagrams.

In the following definition, we say that the state diagram of the system Sys *semantically fulfills* a sequence diagram sd (denoted by $Sys \vDash sd$) if the synchronization product of their derived graphs can be simulated by the graph of the system Sys.

Let the symbol $[\![Sys]\!]$ denote the implementation graph $\mathcal{A}_{Impl}$ of the system and let the symbol $[\![sd]\!]$ denote its specification graph $\mathcal{A}_{Spec}$.

Definition 11 *$Sys \vDash sd$ iff $[\![Sys]\!] \otimes_G [\![sd]\!] \preceq [\![Sys]\!]$.*
Let $\boldsymbol{SD}$ be a subset of sequence diagrams. $Sys \vDash \boldsymbol{SD}$ iff $\forall sd \in \boldsymbol{SD}, Sys \vDash sd$.

The fulfillment definition requires only that a part of the scenarios from the specification graph $[\![sd]\!]$ are realized by the implementation graph $[\![Sys]\!]$, interleaved perhaps with some of its particular actions, providing that these sequences do not lead to erroneous or deadlock states. Indeed, interactions brought by the implementation graph should not make the merged graph reach any erroneous or deadlock states specified in the specification graph which, indeed, may contain some forbidden sequences that are derived from enclosing interaction fragments under the scope of the operators *neg* or *assert*.

We give, now, the following proposition that asserts that the aforementioned definition is adequate for ascertaining that forbidden scenarios never occur once we prove that the system fulfills its sequence diagram. We, particularly, show that the statement Sys fulfills sd given in this definition, implies that $\mathcal{L}_{\mathcal{E}}\{[\![sd]\!]\} \cap Prefixes(\mathcal{L}\{[\![Sys]\!]\}) = \emptyset$, where $\mathcal{L}_{\mathcal{E}}\{[\![sd]\!]\}$ denotes the language of all forbidden computations of it argument and $\mathcal{L}\{[\![Sys]\!]\}$ denotes the set of sequences of Sys . That is, the system will not be able to achieve any negative sequence specified by an interaction fragment wrapped by either the *neg* or *assert* operators in sd.

Proposition 12 $Sys \vDash sd \implies \mathcal{L}_{\mathcal{E}}\{[\![sd]\!]\} \cap Prefixes(\mathcal{L}\{[\![Sys]\!]\}) = \emptyset$

Proof.

Let $Sys \vDash sd$. We use a proof by "reductio ad absurdum".

Suppose $\mathcal{L}_{\mathcal{E}}\{[\![sd]\!]\} \cap Prefixes(\mathcal{L}\{[\![Sys]\!]\}) \neq \emptyset$.

Thus, $\exists \omega \in \mathcal{L}_{\mathcal{E}}\{[\![sd\ ]\!]\}$ such that $\omega \in Prefixes(\mathcal{L}\{[\![Sys]\!]\})$. We deduce that ω is a sequence that makes both the two graphs $[\![Sys]\!]$ and $[\![sd\ ]\!]$ evolve, respectively, to states p and q where q is an erroneous state in $[\![sd]\!]$. So, also is the combined state$(p, q) \in [\![Sys]\!] \otimes_G [\![sd]\!]$ (by Definition 10).

On the other hand, $Sys \vDash sd \implies [\![Sys]\!] \otimes_G [\![sd\ ]\!] \preceq [\![Sys]\!]$. Hence, we get $(p, q) \preceq p$. According to the definition 9 of simulation, the state p (in $[\![Sys]\!]$) should also be an erroneous state. However, $\mathcal{E}_{Impl} = \emptyset$. We deduce, thus, that our hypothesis is false. ■

For instance, if we examine the graph resulting from the combination of the behavioral graph (Fig.5.2(b)) related to the initial sequence diagram (Fig.5.2(a)) with the behavioral graph of the whole system (given in Fig.5.5), we notice that erroneous states could be reached therein. However, we can see that the whole graph of the corrected train control system (given in Fig.5.13(b)) safely simulates the combined graph of Fig.5.13(a) resulting from its combination with the behavioral graph of the improved sequence diagram (given in Fig.5.11).

5.4 Modular consistency checking

In this section, we tailor the exhaustive verification approach given in Subsection 5.3.2 for the component based framework. Indeed, this seems suitable to cope with the modular based design of systems, the state machines of which should individually and cooperatively conform with subsets of requirements that are elicited in terms of graphical scenarios.

Consequently, we check only one component implementation against each of its sequence diagrams, one by one. We first build *partial* specification and implementation graphs related to one component from, respectively, the sequence and state diagrams. Next, we proceed to their consistency checking by proving that their full synchronized product can be *simulated* by the partial implementation graph related to the component state machine.

The full synchronization operator has to deal with a partial implementation graph related to one component that may implement a part of computations of the specification graph and may encompass, just as well, some (inter)actions which do not belong to its sequence diagram we are checking. On the other hand, this sequence diagram may contain messages depicting interactions between other participants, and which are not causally linked to our component actions[3].

So, the uncommon and unrelated interactions should not be considered for synchronization when combining partial graphs. Indeed, these events model interactions which one of the two graphs carries out under its sole control and without the cooperation of the other graph, and vice versa.

The combined graph would contain, hence, sequences containing interactions that the component implementation graph is not compelled to realize. When comparing the combined graph with the component implementation graph, these actions are hidden into $\tau-$actions although they could be interleaved with implemented sequences. Con-

[3]Two actions are causally linked if the occurrence of the latter needs that of the former. In context of StateCharts, this is because the performance of the former provides the latter with its trigger event or activates its source state.

sequently, we need to use a branching simulation as observational comparison criterion that takes care of the interleaving of implemented sequences with such $\tau-$actions.

Before going into more details about the consistency checking, we start by explaining in the following subsections the methods for building the partial implementation and specification related to one main component, respectively, from state and sequence diagrams.

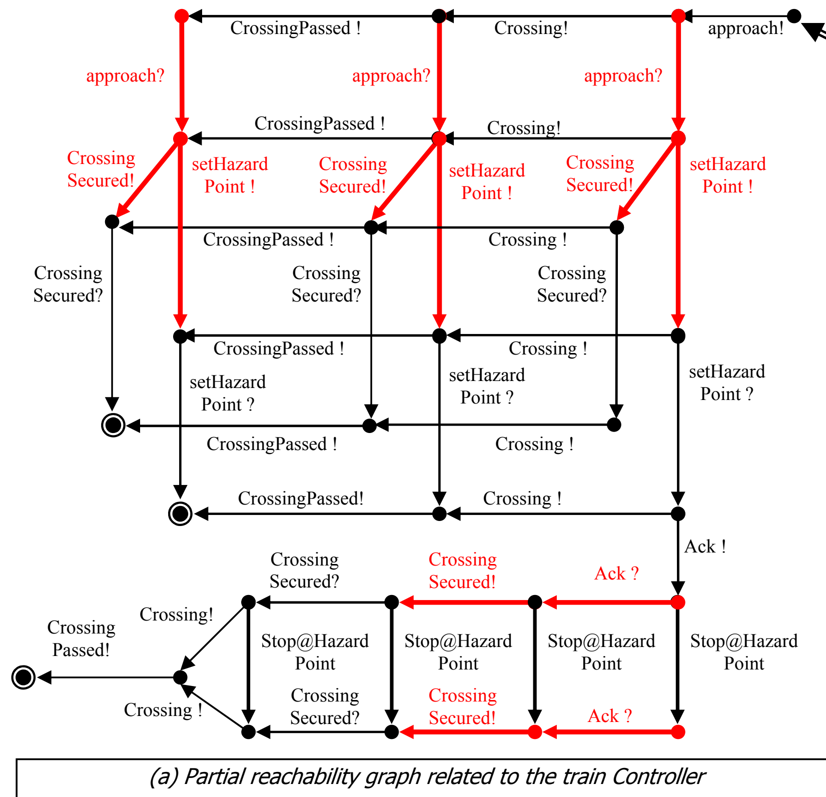
5.4.1 Partial reachability graph

We have explained in Chapter 3 a method of mapping the state diagram consisting of components' state machines (see Fig.5.3) into a modular Petri net (see Fig.5.4) which we aim to explore, here, in a component-oriented fashion. In fact, we want to yield the implementation model of one component by building the reachability graph of the Petri net module that is related to this component state machine.

However, this partial state exploration has to take into account the asynchronous communication between cooperating components through the flow of triggering events dispatched among them. Hence, the partial reachability graph of a Petri net module (i.e., implementation model of its related component) should include not only its transitions (the component actions) but also its causally linked transitions from other Petri net modules related to cooperating components that provide them with the triggering events, as explained below. For instance, red colored arcs in Fig.5.6, illustrate such causally linked transitions we have to add to partial graphs of the train control system components. It is worthwhile to note that such additional transitions constitute common "glues" between partial implementation graphs of components for synthesizing the system graph and then allow us to check their compatibilities.

The generation of the partial graph related to the execution of some main component C_0 of a system Sys is based on favoring transitions of this component C_0. Initially, we should execute only enabled transitions from C_0. Every transition, which does not need any trigger event from its environment, will fire directly. The transitions outgoing from active statuses (configurations of the component) but which require incoming events from other components, will be handled differently.

If there is a transition t_0 in C_0 requiring a trigger event from C_1, we should make this second component prioritized but we restrict its execution to only sequences of C_1 transitions helping t_0 to become enabled. However if C_1 evolution, itself, is interrupted at any transition t_1 because it needs an incoming event from a third component C_2, we must perform all transitions from this component making t_1 enabled and so on. This process is repeated recursively in order to enable and perform all transitions of C_0 and any transitions in other components enabling those of the prior component C_0.



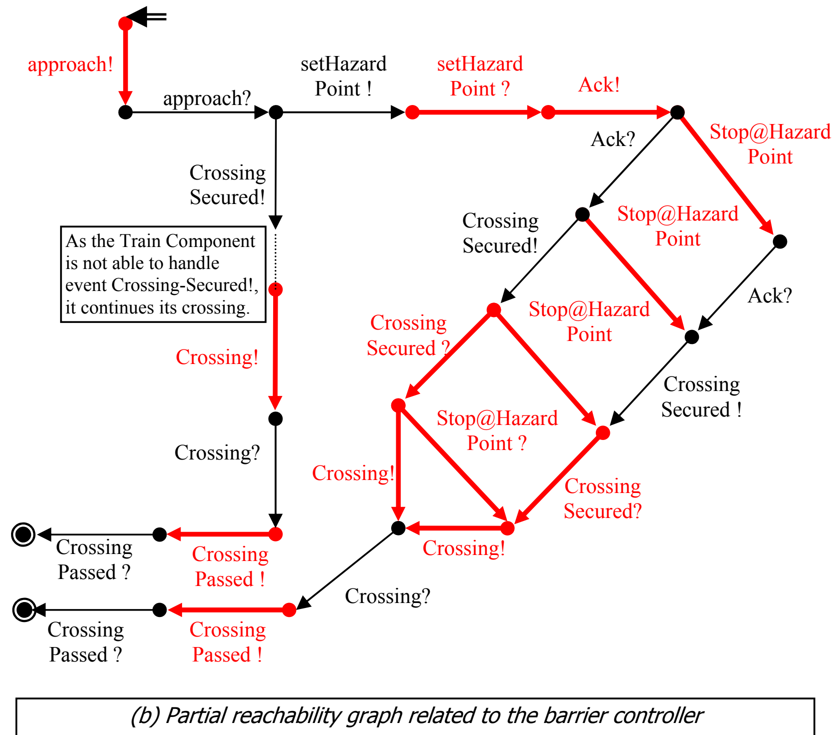


Figure 5.6: Partial behavioral graphs related to train and barrier controllers state machines.

To avoid infinite recurrence, we clearly should prevent any new blocked transition from insertion into the set of transitions we are trying to enable if it already exists therein.

For instance, the partial graph of the train controller component (in Fig.5.6(a)) begins with the sending event *Approach*! and then it can either send *Crossing*! or receive *setHazarPoint*?. However, for performing the receipt transition, we have to make its cooperating barrier controller component evolve by performing the sequence of causally linked transitions *Approach*?; *setHarzardPoint*! which enables the receipt transition in our main component (i.e., train controller).

Procedure $Generate-Partial-Graph$ given below, builds the partial reachability graph related to some component C_0. It uses a stack to store the markings of the Petri net we have to explore, starting from the initial marking. The first action (line 5) in the iterative block (lines 3-30) is to pop a marking M for exploration.

Algorithm 13 ***Procedure*** *Generate-Partial-Graph*

```
In:  Marking M0, Component C0;
Out:partial Reachability graph;
1: Explored :={M0};
2: Push (M0);
3: while not empty (stack) do
4:  begin
5:    M := Pop();
6:    Set1 := Fireable-Transitions (M,C0);
7:    for all t ∈ Set1 do
8:    begin
9:          M' := Successor (M,t);
10:          if M' ∉ Explored then
11:          begin Push (M');
12:               Explored := Explored ∪ {M'}
13:          end if
14:     end for
15:     Set2 := Potentially-Enabled-Transitions (M,C0);
16:     for all t ∈ Set2 do
17:     begin
18:       Find the set X of transitions in other components
            which enable t by generating its trigger event;
19:       for all (t',C') ∈ X do
20:       begin
21:             for all M' ∈ Unlock(M,{t},t',C') do
22:             begin M" := Successor (M',t);
23:                    if M" ∉ Explored then
24:                    begin Push (M");
25:                          Explored := Explored ∪ {M"}
26:                    end if
27:             end for
28:       end for
29:     end for
30: end while
end Generate-Partial-Graph.
```

Then, we look for enabled transitions (including potential ones) from this marking M which belongs to the prioritized component C_0. We discern two kinds of transitions:

- The first set (set_1) (line 6) includes enabled transitions we can fire directly (lines 7-14). After firing any transition, the marking obtained (returned by *Successor* function) is pushed into the stack so that it can be handled (i.e., explored) later.
- The second set (set_2) (line 15) contains potentially enabled transitions whose control flow inplaces are marked but not those supporting event flows. Intuitively, each of these transitions denotes in the StateChart an arc outgoing from an **active** configuration but whose trigger event is not yet available. So, to make the transitions of set_2 fireable, we should find recurrently all sequences of causally linked transitions (throughout lines 16-29) in other components and fire them in order to generate the required triggering events:
  - To make a transition "t" fireable in the current prioritized component, we compute the set X (line 18) of all transitions in other components whose firings produce the needed tokens in events flow inplaces of "t" rendering it enabled.
  - Next, we handle each pair (t', C') (lines 19-28) from X by calling a recursive function "*Unlock*" whose arguments are items of this pair together with the current marking and the set of transitions that we want to make fireable.
  - The procedure "*Unlock*" returns a set of new reached markings each of which enables t and has thus to be separately explored (if not yet explored) (lines 21-27).

Algorithm 14 ***Function*** *Unlock*

```
In:  Marking M, Set of Transitions T, Transition t', Component C';
Out:  set of Markings;
1:  Set of Markings SM := ∅;
2:  if t' is fireable from M then Handle-Successor(M)
3:  else if the control flow inplace p of t' is marked then
4:  begin
5:     Calculate the set X of transitions which enable t'
         by generating a token in its trigger event inplace;
6:     for all (t",C") ∈ X and t" ∉ T ∪ {t'} do
7:        for all M' ∈ Unlock(M,T ∪ {t'},t",C") do
8:           Handle-Successor(M');
9:        end for
10:    end for
11: end
12: else for all t" ∈ Pre(p) and t" ∉ T ∪ {t'} do
13:       for all M' ∈ Unlock(M,T ∪ {t'},t",C') do
```

```
14:          if t′ is fireable at M′ then Handle-Successor(M′)
15:          else
16:           begin
17:              Calculate the set X of transitions enabling
                   t′ by generating its trigger event;
18:              for all (t”,C”) ∈ X and t” ∉ T ∪ {t′} do
19:                for all M” ∈ Unlock(M′,T ∪ {t′},t”,C”)
20:                  do Handle-Successor(M”);
21:           end if
22:         end for
23:        end for
24:   return SM;
end Unlock;
```

The arguments of the routine *Unlock* are the current marking M, the transition t' we want to fire, the component C' owner of t', and a set T of potentially enabled transitions we are trying to enable. If t' is enabled in the marking M then we immediately fire it by calling the routine "*Handle-Successor*" (line 2), otherwise we check whether the control flow inplaces of t' are marked (line 3):

- When these places are marked, this means that t' is potentially enabled too. So, the same handling (lines 4-11) could be achieved for t' as that has been done for t in the first algorithm. In such a case, we search all transitions in other components whose firings generate event tokens making it possible to enable t'. For each causally linked transition t" in some component C", we recall the same routine "*Unlock*" in order to fire it and make thus t' enabled.
- On the other hand, if control flow inplaces of t' are not marked then we have to let its owner component C' progress to a marking (i.e., status configuration in the StateChart vocabulary) that makes t', fireable or at least potentially enabled. In other terms, we should achieve the following steps:
  - First, we separately search any transition preceding t' in the same component C' (lines 12-13) and call the routine "*Unlock*" to fire it so that control flow inplaces of t' become marked (i.e. source configuration of t' becomes active).
  - Thereafter, if t' becomes enabled then we immediately fire it by calling the routine "*Handle-Successor*" (line 14), otherwise t' is only potentially enabled and needs thus the usual handling of its event inplaces (lines 16-21) using again the routine "*Unlock*".

At last, the called function "*Unlock*" returns a set SM of reached markings each of which enables the first locked transition t in the main component C_0 that has first

called this routine. Note that tokens in control flow inplaces of t cannot be consumed during *"Unlock"* execution because they can be used only by transitions outgoing from the same component active configuration.

Procedure *Handle-Successor* generates the successor marking M_2 by firing the unlocked transition t' from the marking M_1. M_2 is added to the set SM of new markings and pushed into the stack if it has not been explored yet.

Algorithm 15 ***Procedure*** *Handle-Successor*

```
In Marking M1;
 M2 := Successor(M1, t'); SM := SM ∪ {M2};
 if M2 ∉ Explored then
   begin Push (M2);
         Explored := Explored ∪ {M2}
   end if
end Handle-Successor;
```

5.4.2 Composition and compatibility

Let Sys be a system under design. Formally, the reachability graph of its related Petri net (note it $[\![Sys]\!]$) is a labeled transition system $G = \langle Q, \Sigma, \hookrightarrow, q_0, \mathcal{F}, \mathcal{E}\rangle$ where: Q is the set of reachable markings of the Petri net (i.e., configurations of Sys) with q_0 as its initial marking M_0, Σ is the set of events (actions) which are labels of the Petri net transitions, $\hookrightarrow \subseteq Q \times \Sigma \times Q$ is the set of transitions whose firings change the markings, $\mathcal{F}$ is the set of successful termination states, and $\mathcal{E} = \emptyset$ is the set of erroneous states (i.e., all implementation traces are allowed, by default). Since the state diagram of the system is the collection of all its interacting components' state machines which evolve in a parallel way, we deduce that the means to synthesize its entire reachability graph $[\![Sys]\!]$ shall be the parallel (synchronized) product of partial reachability graphs related to all its components C_i (denoted by $[\![C_i]\!], i \in I$).

Definition 16 *Let G_1 and G_2 be two graphs: $G_1 = \langle Q_1, \Sigma_1, \hookrightarrow_1, q_{0_1}, \mathcal{F}_1, \mathcal{E}_1\rangle$, $G_2 = \langle Q_2, \Sigma_2, \hookrightarrow_2, q_{0_2}, \mathcal{F}_2, \mathcal{E}_2\rangle$. A full synchronized product $\otimes_P$ of two graphs G_1 and G_2 is the graph $G = \langle Q, \hookrightarrow, \Sigma, q_0, \mathcal{F}, \mathcal{E}\rangle$ such that:*

- $Q = Q_1 \times Q_2$ *with* $q_0 = (q_{0_1}, q_{0_2})$.
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\hookrightarrow = \{(p_1, q_1) \xrightarrow{a} (p_2, q_2) | a \in (\Sigma_1 \cap \Sigma_2) \wedge (p_1 \xrightarrow{a} p_2) \in \hookrightarrow_1 \wedge (q_1 \xrightarrow{a} q_2) \in \hookrightarrow_2 \} \cup \{(p_1, q_1) \xrightarrow{a} (p_2, q_2) | a \in (\Sigma_1 \setminus \Sigma_2) \wedge ((p_1 \xrightarrow{a} p_2) \in \hookrightarrow_1 \wedge q_1 = q_2) \vee (a \in (\Sigma_2 \setminus \Sigma_1) \wedge (q_1 \xrightarrow{a} q_2) \in \hookrightarrow_2 \wedge p_1 = p_2)\}$
- $\mathcal{E} = \mathcal{E}_1 \times \Sigma_2 \cup \Sigma_1 \times \mathcal{E}_2$

- $\mathcal{F} = TransitiveClosure(\mathcal{F}_1 \times \mathcal{F}_2)$.

$TransitiveClosure(\mathcal{F}) = \mathcal{F} \cup \{p \in Q | \exists q \in \mathcal{F}, p \stackrel{\omega}{\Longrightarrow} q)\}$ *where* $\stackrel{\omega}{\Longrightarrow}$ *is a finite sequence of transitions whose labels are in* $((\Sigma_1 \setminus \Sigma_2) \cup (\Sigma_2 \setminus \Sigma_1))$.

On combining two graphs, each one of them progresses individually if it has to perform a non-shared action, otherwise the two graphs should synchronize (see the combined graph of Fig.5.7(a) resulted from synchronization of partial reachability graphs of Fig.5.6). For successful termination of the combined subsystem, once one of the two components reaches its final state, the other component will solely determine wether the subsystem can succeed to reach its whole final state. In fact, the latter can reach its final state when the component reaches also its final state by achieving zero or more of its internal actions that do not need any interaction with its cooperating component (which is terminated and cannot, in no way, prevent the subsystem to reach the whole final state). All states throughout a sequence ω consisting of such internal actions are considered as final.

Property 2 *The operation* $\otimes_P$ *over behavioral graphs is commutative and associative.*

According to this property, the whole reachability graph could be gradually built by combining partial graphs of its components by pairs. However, the parallel combination requires a comprehensive and correct cooperation (i.e, fully compatibility) between interacting components to yield the behavioral graph of the whole system. The following definition provides the means to uncover any incompatibility situation between two cooperating components (i.e., deadlock status consisting in a sink or erroneous state from which the combined subsystem is unable to progress).

Definition 17 *Let* C_1, C_2 *be two interacting state machines (modeling two components). We say they are compatible (with each other) (respectively, fully compatible) if the parallel product graph* $(\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket)$ *is a deadlock-free and error-free subgraph of (respectively, isomorphic to) the reachability graph* $\llbracket C_1 || C_2 \rrbracket$ *of the combined state diagram* $C_1||C_2$[4].

The main target of this compatibility definition is to ensure that the individual progress of each component which relies on a *controlled* correct cooperation with its environment, will not be prevented (or altered) from its effective execution when plugged in some environment. Indeed, when generating its partial reachability graph, we make other cooperating components progress such that the main component succeeds to evolve towards its correct completion or to progress without reaching any irregular

[4]The symbol || denotes the parallel combination of communicating state machines.

status. In other words, this main component controls the others in order to successfully terminate or safely progress. However, when exploring the entire reachability graph of a subsystem, our main component has no more control over its environment (its cooperating components). It could not make the other components evolve as it likes to realize its intended functionality.

The full synchronization product of partial implementation graphs of components, consists in making them cooperate together to achieve their sequences with merge of common events. Unfortunately, this operation may fail and lead to deadlock situation when the combined components are not compatible. Otherwise, these are said compatible if such statuses are absent. However, the synchronized graph may still not include sequences of independent interactions (i.e., not causally linked actions) that we shall find in the behavioral graph of the combined system whose components have, actually, no control on each other. Hence, compatibility implies that the former graph be a subgraph of the latter where some sequences of independent actions are pruned. Otherwise, if the two graphs are isomorphic, this implies that the behavior of the whole subsystem would consist of exactly controlled behaviors of its constituents, as expected, leading thus to a fully compatibility between them.

Proposition 18 *Let $\{C_1, \ldots, C_n\}$ be the set of components of a system Sys. If all pairs of these components are compatible (respectively fully compatible) then the product graph ($[\![C_1]\!] \otimes_P \cdots \otimes_P [\![C_n]\!]$) is a deadlock-free and error-free subgraph of (respectively, isomorphic to) the reachability graph of the system $[\![Sys]\!]$.*

Proof.

Assume $\{C_1, \ldots, C_n\}$ are compatible (respectively fully compatible). We need to prove that ($[\![C_1]\!] \otimes_P \cdots \otimes_P [\![C_n]\!]$) is a deadlock-free and error-free subgraph of (respectively, isomorphic to) ($[\![C_1|| \ldots ||C_n]\!]$). This holds for n=2 in view of Definition 17. We assume now that the property holds for k components and prove that it remains true for k+1. Let G_1 be the synchronization product graph ($[\![C_1]\!] \otimes_P \cdots \otimes_P [\![C_{k+1}]\!]$)) and let G_2 be the reachability graph ($[\![C_1|| \ldots ||C_{k+1}]\!]$):
$G_i = \langle Q_i, \Sigma_i, \hookrightarrow_i, q_{0_i}, \mathcal{F}_i, \mathcal{E}_i \rangle|_{i=1,2}$ where $\mathcal{F}_2 = \emptyset, \mathcal{E}_2 = \emptyset$.
We have to exhibit two injective mappings (respectively, bijections) f and g such that:
$f : Q_1 \longrightarrow Q_2$, $g : (\hookrightarrow_1) \longrightarrow (\hookrightarrow_2)$,
and $\forall t \in\hookrightarrow_1$: $t = (q_1, a, q_2) \implies g(t) = (f(q_1), a, f(q_2))$.

We build f and g by induction on the length of the sequence of fired transitions "$t_1 \ldots t_m$" as follows:

- It is easy to prove $f(q_{1_0}) = q_{2_0}$ as the states q_{1_0} and q_{2_0} denote the same initial configuration (unique initial net marking).

- Assume that this property is true up to some sequence $(t_1 \ldots t_m)$. So, by hypothesis,

we have $f(q_{1_m}) = q_{2_m}$ where $q_{1_0} \stackrel{t_1 \ldots t_m}{\Longrightarrow} q_{1_m}$ and $q_{2_0} \stackrel{t_1 \ldots t_m}{\Longrightarrow} q_{2_m}$. We have to prove that the property remains true when executing any additional transition.

- $\forall t \in \hookrightarrow_1$ where $t = (q_{1_m}, a, q_{1_{m+1}})$. This means that t is fireable from q_{1_m} in at least one component C_i. As q_{1_m} is a tuple of components' states $(\ldots s_i \ldots)$ and $q_{1_{m+1}}$ is as well a tuple of components' states $(\ldots s'_i \ldots)$, there should be, following Definition 16 of "$\hookrightarrow$", a transition $s_i \stackrel{a}{\longrightarrow} s'_i \in \hookrightarrow_i$ (set of transitions of $[\![C_i]\!]$). According to the induction hypothesis, q_{2_m} depicts the same marking as q_{1_m} since they are reached from the same initial marking by the same sequence whose *length* $\leq m$. So, the transition t is also fireable from q_{2_m}. According to Definition 17, the synchronization product is a deadlock-free and error-free subgraph where $q_{1_{m+1}}$ is not an erroneous or deadlock state. Hence, $q_{2_m} \stackrel{a}{\longrightarrow} q_{2_{m+1}}$ and $f(q_{1_m}) = q_{2_m}$.

Now, we continue the proof by showing that these mappings f and g are bijections in case we want to prove that $([\![C_1]\!] \otimes_P \cdots \otimes_P [\![C_n]\!])$ is isomorphic to $([\![C_1|| \ldots ||C_n]\!])$, provided $\{C_1, \ldots, C_n\}$ are fully compatible.

- So, on the other hand, any transition outgoing from q_{2_m} should fire in at least one component C_i. We discern two cases: either $i \leq k$ or $i = k + 1$.

First case: If C_i belongs to the first k components, then by hypothesis, the subsystem $([\![C_1|| \ldots ||C_k]\!])$ contains as well t. If the label "a" of this transition does not belong to the set of actions of $[\![C_{k+1}]\!]$ then the whole graph G_2 can also perform it. On the other hand, if "a" belongs also to $[\![C_{k+1}]\!]$ actions then graphs of the two components C_i and C_{k+1} can synchronize to perform "a" without problem since they are fully compatible. Hence, $f(q_{1_m}) = q_{2_m}$.

Second case: If $i = k + 1$ then the transition is performed by the last component $k + 1$. If the label of this transition "a" belongs only to the set of actions of C_{k+1} then following Definition 16, this transition will be also added in the parallel product between C_{k+1} graph and $([\![C_1]\!] \otimes_P \cdots \otimes_P [\![C_k]\!])$. On the other hand, if "a" is a common action between C_{k+1} and some component $C_{j \leq k}$ then their graphs are also able to synchronize for "a" because they are compatible. Consequently, we deduce that the property remains true at the $m + 1$ step.

■

Example: We presented in Subsection 5.3.1, a train control system which aims to provide a safe and smooth train traffic across some borders (see Fig.5.2(a) and Fig.5.3). Using our modular approach, we try now to check whether the two components always evolve without leading to irregular or deadlock situations.

The analysis of the partial graph related to the train (see Fig.5.6(a)), shows a path containing thick red colored arcs labeled with causally linked actions from barrier component. These actions have been added, by our partial exploration algorithm, to allow the train component to progress. Similarly, we add in the partial graph

related to the barrier controller (in Fig.5.6(b)) all available train actions that make it evolve . However, when combining the two partial graphs by means of the parallel operator (see Fig.5.7(a)), we discover that an execution path of the merged graph leads to deadlock statuses (e.g., "*Approach!; Approach?; CrossingSecured!; Crossing!; Crossing?; CrossingPassed!; CrossingPassed?*"). Hence, we deduce that these two components are not compatible and not able to correctly cooperate. Namely, they are unable to handle the event "*CrossingSecured?*" at the end of the previous sequence. In Subsection 5.4.5, we will see how to discover such irregular situations when comparing the partial graph of any component with its related sequence diagrams (see Fig.5.7(b)).

Furthermore, on comparing the synchronized graph of Fig.5.7(a) with the behavioral graph of Fig.5.5 related to the whole system, we see that though the former is a subgraph of the latter, it is not deadlock-free. Hence the two components cannot cooperate to safely achieve their intended "controlled" sequences of interactions.

5.4.3 Partial interaction model

To allow using automated tools for analysis and verification of sequence diagrams, we have defined in [48] a branching time semantics for these diagrams, which record faithfully the intended traces of events related to exchanged messages between the participants, together with branching structures. Now, we easily make use of this method within our component based framework thanks to the compositional style we used to provide combined interaction fragments with formal semantics by means of recursive definitions of interaction operators and using few algebraic operations over resulting graphs [48].

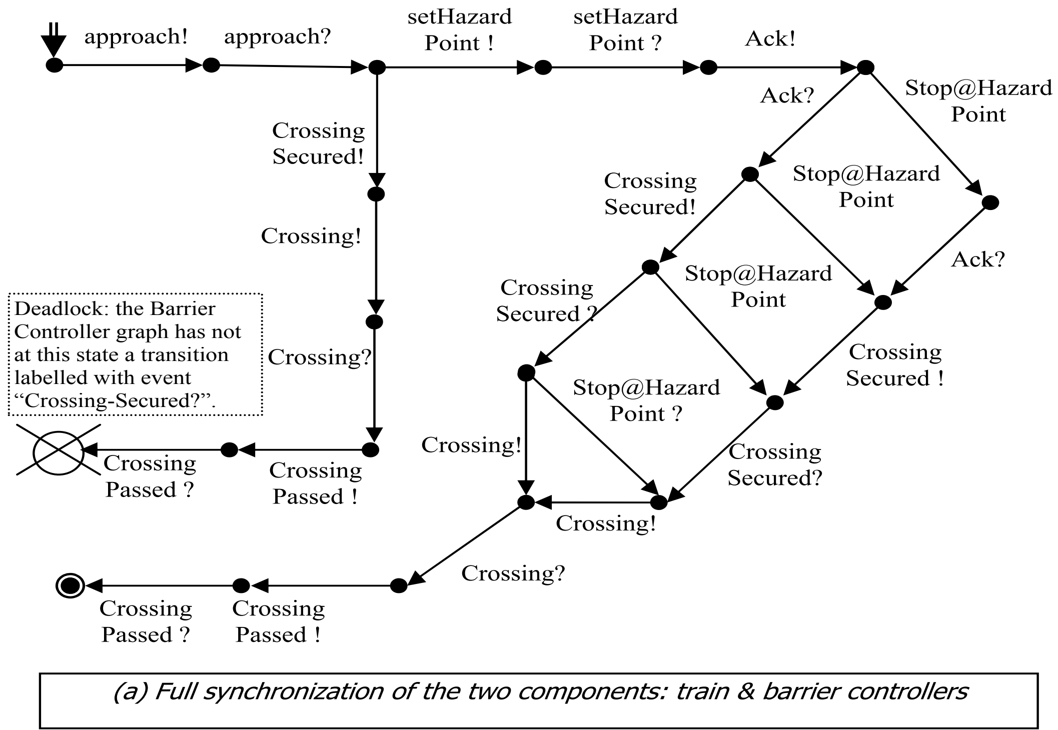
Since we are interested in checking one component at a time (note it C_0), we select only a subset of sequences diagrams (denoted by $SD[C_0]$), namely those where the main component C_0 is involved as a participant.

Each sequence diagram $sd \in SD[C_0]$ is built up by combining its interaction fragments by means of a set of interactions operators. Therefore, we build the transition system of sd by combining, recursively, the graphs related to its enclosed fragments by means of the algebraic operations over graphs we have defined in [48] (see Subsection 4.5 summarizing this process).

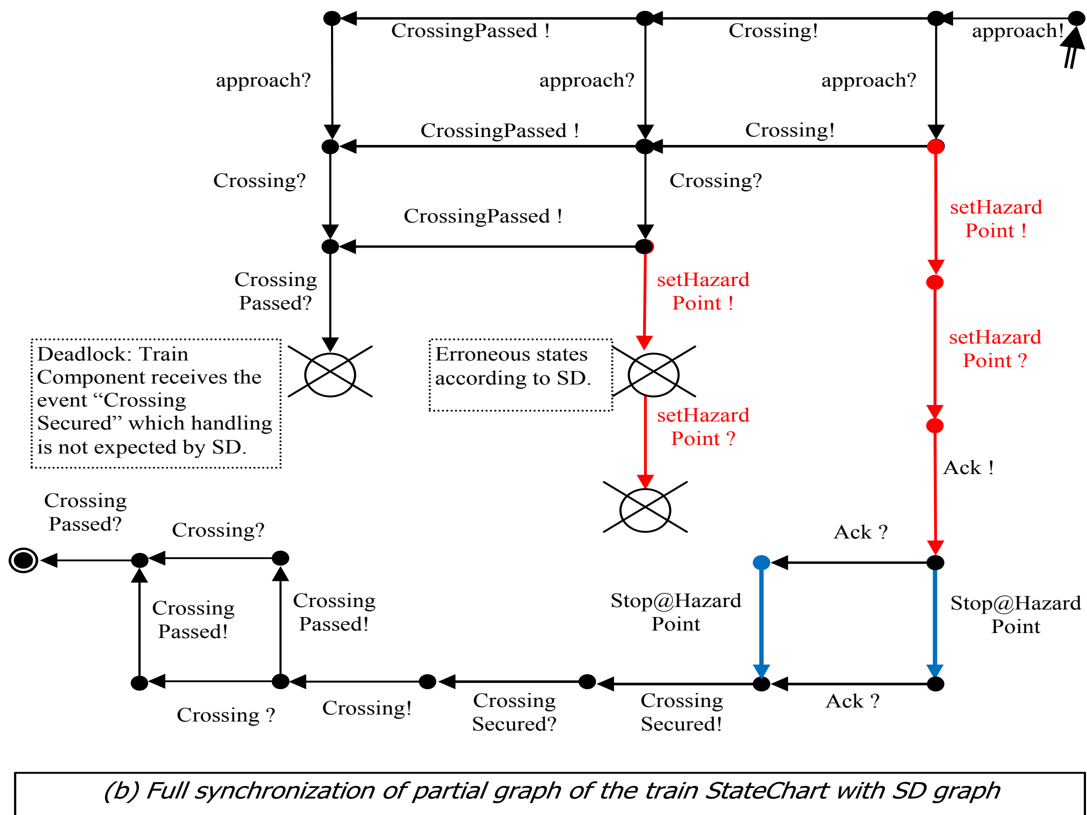
For instance, Figure 5.2(b) illustrates the behavioral graph of the sequence diagram of Figure 5.2(a) where our main component (train controller) is involved.

5.4.4 Comparison criterion

In order to enable the comparison between partial behavioral graphs derived from sequence and state diagrams, we need to take into account, besides the visible actions



(a) Full synchronization of the two components: train & barrier controllers



(b) Full synchronization of partial graph of the train StateChart with SD graph

Figure 5.7: Modular compatibility and consistency checking.

of Σ, the concept of τ-actions modeling interactions that are unobservable from side to side. The set of transitions becomes thus $\hookrightarrow \subseteq Q \times \Sigma \cup \{\tau\} \times Q$.

We reuse, next, the full synchronized product between graphs $G_i = \langle Q_i, \hookrightarrow_i, \Sigma_i, q_{0_i}, \mathcal{F}_i, \mathcal{E}_i \rangle_{|i=1,2}$ (according to Definition 16) which forces them to synchronize for all common actions but allows them to progress in an interleaving way for the performance of uncommon actions.

In the current approach, we need to define a simulation relation which fits for the modular consistency checking of dynamic diagrams. It should consider τ actions resulting from the hiding of specific actions brought by the partial interaction model (i.e., behavioral graph of the sequence diagram) in the sense that these actions are neither performed by the component itself, nor causally linked to the component actions. That is why these must be made unobservable to the component implementation graph although they are often interleaved with the other common actions.

However, these actions have to not interfere with the execution of the scenarios where the component is involved. In other terms, we have to ensure that such actions do not prevent the component from achieving some or all authorized sequences among those depicted by the sequence diagram. For instance, suppose that the component, when involved in some scenario, can reach some state s_1 by performing an action a which should be followed by an action b. Now, consider the state s_1 in the partial interaction graph, from which some unrelated actions may be performed and make the graph reach another state s_2. Such a new state should still offer the second action b. Consequently, we have to use a branching simulation that takes care of the unrelated actions we need to hide with regards to the component implementation graph.

Definition 19 *A branching simulation is a binary relation $\mathcal{R} \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2$, $p\mathcal{R}q$ iff:*

- *if $p \in \mathcal{F}_1$ then $q \in \mathcal{F}_2$.*
- *if $p \in \mathcal{E}_1$ then $q \in \mathcal{E}_2$.*
- $\forall a \in \Sigma, \forall p', p_1 \in Q_1 : p \stackrel{\tau}{\Longrightarrow} p_1 \stackrel{a}{\longrightarrow} p'$ *then* $\exists q' \in Q_2 : q \stackrel{a}{\longrightarrow} q' \wedge p_1\mathcal{R}q \wedge p'\mathcal{R}q'$.

A graph G_2 simulates another graph G_1 ($G_1 \preceq_{br} G_2$) when $q_{0_1} \preceq_{br} q_{0_2}$.

Note that two states p and q are related by $\preceq_{br}$ if we can perform from the latter state the same observable actions we can carry out from the former state in order to reach, respectively in both sides, states that still be related. Moreover, if some state p' is reached by a τ-step from p then it should be also simulated from q. This strong constraint is due to the fact that unobservable actions are considered, in our framework, as internal actions under the sole control of its owning graph and do

not need any cooperation with the other graph (they actually represent concurrent interactions between some components other than the one we are currently handling and are not causally linked to its actions).

Next, we give and prove an important proposition stating that our branching simulation is preserved under our composition operation of behavioral graphs. In other words, related components by $\preceq_{br}$, remain related once plugged into their environment.

Proposition 20 *Let G_1, G_2 and G_3 be behavioral graphs.*
$G_1 \preceq_{br} G_2 \implies (G_1 \otimes_P G_3) \preceq_{br} (G_2 \otimes_P G_3)$.

Proof.

We need to prove that the relation $\mathcal{R} = \{((p,q),r) \in (Q_1 \times Q_2) \times Q_3 | r \in Q_3 \wedge p \preceq_{br} q\}$ is a branching simulation $\preceq_{br}$.

Let $(p,q) \in Q_1 \times Q_2$ such that $p \preceq_{br} q$ and let $r \in Q_3$. We show that $((p,q),r) \in \mathcal{R}$. In other terms, $((p,q),r)$ satisfies Def.19 requirements of $\mathcal{R}$.

Case 1: According to Def.19, if $p \in \mathcal{E}_1$ then $q \in \mathcal{E}_2$. By Def.16, we get whatever the status r, $(p,r) \in \mathcal{E}_{1\times 3}$ and $(q,r) \in \mathcal{E}_{2\times 3}$.

Case 2: According to Def.19, if $p \in \mathcal{F}_1$ then $q \in \mathcal{F}_2$. According to Def.16, we get two subcases following the status of r :

SubCase 21: if $r \in \mathcal{F}_3$ then by Def.16, $(p,r) \in \mathcal{F}_{1\times 3}$ and $(q,r) \in \mathcal{F}_{2\times 3}$.

SubCase 22: Otherwise, $r \notin \mathcal{F}_3 \implies \exists r' \in Q_3, \exists b \in \Sigma_3 : r \xrightarrow{b} r'$. According to Def.16, p and q can have two statuses:

1. p and q can be final states from which no outgoing internal transition exists. In such a case, we deduce that b is an uncommon (no synchronized) action in both $(G_1 \otimes_P G_3)$ and $(G_2 \otimes_P G_3)$. By Def.16, this implies that: $(p,r) \xrightarrow{b} (p,r')$, and $(q,r) \xrightarrow{b} (q,r')$. We can deduce that $(p,r')\mathcal{R}(q,r')$ because (p',r') and (q',r') have the same status which depends only on the status of r' (i.e., final or not) since p and q are both final. Hence, if r' is not final then we get both $(p',r') \notin \mathcal{F}_{1\times 3}$ and $(q',r') \notin \mathcal{F}_{2\times 3}$, otherwise, if r' is final then we get both $(p',r') \in \mathcal{F}_{1\times 3}$ and $(q',r') \in \mathcal{F}_{2\times 3}$.

2. p and q can be final states which lead to the previous final states throughout a sequence ω of internal transitions (by Def.16). So, since action b cannot be a common action in both $(G_1 \otimes_P G_3)$ and $(G_2 \otimes_P G_3)$, we get either:

   - b is an uncommon action in both $(G_1 \otimes_P G_3)$ and $(G_2 \otimes_P G_3)$. Such a case is similar to point (1).
   - b is a common action in only $(G_2 \otimes_P G_3)$ and not in $(G_1 \otimes_P G_3)$. This means that: $(p,r) \xrightarrow{b} (p,r')$ and $(q,r) \xrightarrow{b} (q',r')$. We deduce $(p,r')\mathcal{R}(q',r')$

because if r' is not final then we get both $(p, r') \notin \mathcal{F}_{1\times3}$ and $(q', r') \notin \mathcal{F}_{2\times3}$, otherwise, if r' is final then we get both $(p, r') \in \mathcal{F}_{1\times3}$ and $(q', r') \in \mathcal{F}_{2\times3}$ (both p and q' are final).

- b is a common action in only $(G_1 \otimes_P G_3)$ and not in $(G_2 \otimes_P G_3)$. This means that: $(p, r) \xrightarrow{b} (p', r')$ and $(q, r) \xrightarrow{b} (q, r')$. We deduce $(p', r')\mathcal{R}(q, r')$ because if r' is not final then we get both $(p', r') \notin \mathcal{F}_{1\times3}$ and $(q, r') \notin \mathcal{F}_{2\times3}$, otherwise, if r' is final then we get both $(p', r') \in \mathcal{F}_{1\times3}$ and $(q, r') \in \mathcal{F}_{2\times3}$ (both p' and q are final).

Case 3: $\forall a \in \Sigma_1, \forall p', p_1 \in Q_1 : p \overset{\tau}{\Longrightarrow} p_1 \xrightarrow{a} p'$ then $\exists q' \in Q_2 : q \xrightarrow{a} q' \wedge p_1\mathcal{R}q \wedge p'\mathcal{R}q'$.

SubCase 31: If $\exists r' \in Q_3 : r \xrightarrow{a} r'$. This implies, $(p, r) \overset{\tau}{\Longrightarrow} (p_1, r) \xrightarrow{a} (p', r') \wedge (q, r) \xrightarrow{a} (q, r') \wedge (p_1, r)\mathcal{R}(q, r) \wedge (p', r')\mathcal{R}(q, r')$.

SubCase 32:

If $\exists b \in (\Sigma_3 \setminus \Sigma_1), \exists r' \in Q_3 : r \xrightarrow{b} r'$. This implies, $(p, r) \overset{\tau}{\Longrightarrow} (p_1, r) \xrightarrow{b} (p_1, r')$.

If $\exists q' \in Q_2, q \xrightarrow{b} q'$ then we get a synchronization for b performance in $(G_2 \otimes_P G_3)$ such that $(q, r) \xrightarrow{b} (q', r') \wedge (p_1, r)\mathcal{R}(q, r) \wedge (p_1, r')\mathcal{R}(q', r')$.

Otherwise, $\nexists q' \in Q_2, q \xrightarrow{b} q'$ then we get an interleaving for b performance in $(G_2 \otimes_P G_3)$ such that $(q, r) \xrightarrow{b} (q, r') \wedge (p_1, r)\mathcal{R}(q, r) \wedge (p_1, r')\mathcal{R}(q, r')$.

Accordingly, we deduce that $\mathcal{R}$ is indeed a branching simulation $\preceq_{br}$. ■

5.4.5 Fulfillment relation within interleaving semantics

Given a graph G and a set Γ of actions. $H_\Gamma(G)$ denotes the graph G where all actions in Γ are hidden.

We say that the state machine of a component C *fulfills* a sequence diagram sd (denoted $C \vDash sd$) if the graph resulting from the full synchronization of the derived graph of C with that of sd, and after the hiding of uncommon actions in $\Gamma = \Sigma([\![sd]\!]) \setminus \Sigma([\![C]\!])$, can be simulated by the graph $([\![C]\!])$. Γ is the set of interactions from sd in which $[\![C]\!]$ is not involved or which are not causally related to actions of C.

This definition should be seen as an *implementation* relation where C is allowed to implement not all but only a subset of the specified sequences in the corresponding sequence diagram.

Definition 21 *$C \vDash sd$ iff $H_\Gamma([\![C]\!] \otimes_P [\![sd]\!]) \preceq_{br} [\![C]\!]$ where $\Gamma = \Sigma([\![sd]\!]) \setminus \Sigma([\![C]\!])$. Let SD be a set of sequence diagrams. $C \vDash SD$ iff $\forall sd \in SD, C \vDash sd$.*

This definition ensures that any component C is able, at least, to achieve the interactions in which it is involved. It is worth noting that some of the related events

carried out by other components, may be causally linked to the events of the main component C. Thus, these events should have been inserted in the behavioral graph $[\![C]\!]$ through our process of partial exploration. Consequently, we can deduce that all components involved in the sequence diagram will correctly cooperate to achieve the intended interaction. This result is given in theorem 23. However, we can only deduce they are partially compatible with regard to this specification fragment.

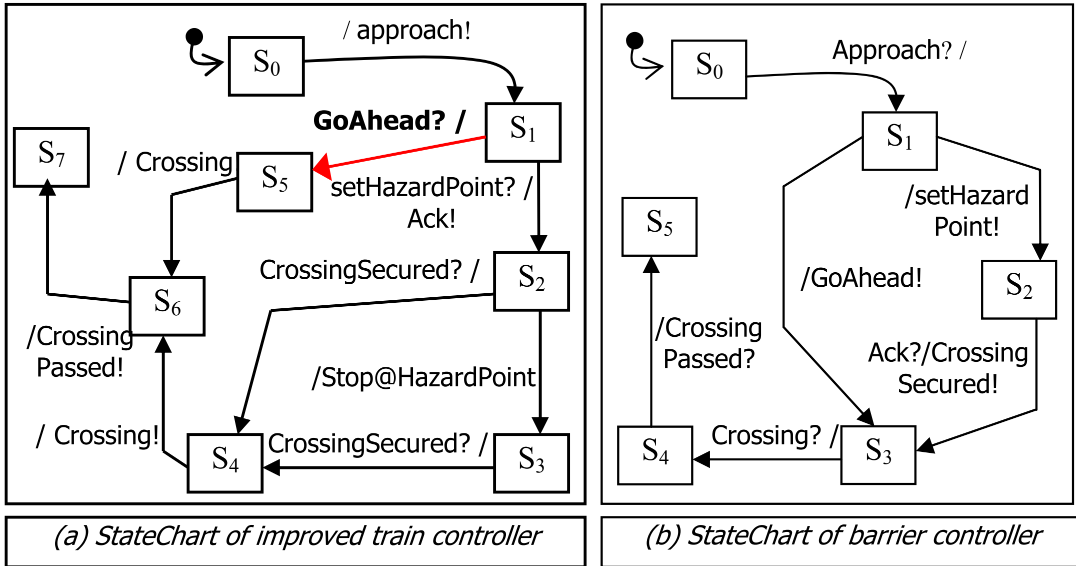
The previous definition does as well, fit for the sequence diagrams including negative fragments which depict forbidden scenarios thanks to the use of a branching simulation along with the merging operator over the augmented behavioral graphs. Besides the fact that it allows to check wether the component graph is able to carry out some or all of the allowed sequences, the definition checks that the component graph does not offer any of its actions in order to achieve a negative sequence. This statement is given in the next proposition which just asserts that the validity definition ensures that erroneous and deadlock states are not reachable in the graph $[\![C]\!] \otimes_P [\![sd]\!]$.

Proposition 22 *If* $C \models sd$ *then* $\mathcal{L}_{\mathcal{E}}\{[\![sd]\!]\} \cap Prefixes(\mathcal{L}\{[\![C]\!]\}) = \emptyset$ *and* $[\![C]\!] \otimes_P [\![sd]\!]$ *is deadlock-free.*

Proof. Similarly to the proof of proposition 12, we assume that there is a sequence in the graph $[\![C]\!] \otimes_P [\![sd]\!]$ that leads to an erroneous or deadlock state. Even though this sequence may contain $\tau-$actions, the component graph can simulate such a sequence since the two graphs are related by $\preceq_{br}$. This means that this graph may contain such an irregular state, which is absurd. ■

Example: To check the train control system, we compare the state machine of the train controller (Fig.5.3(a)) with the interaction diagram (Fig.5.2(a)). For this purpose, we realize the full synchronized product of their behavioral graphs illustrated, respectively, by Fig.5.6(a) and Fig.5.2(b). The combined graph is given in Fig.5.7(b). We notice that the train controller component does not correctly cooperate to achieve the interaction diagram because its scenario-oriented execution leads to a deadlock situation where the barrier controller graph is not able to synchronize with train controller graph on the event *CrossingSecured*? as shown in Fig.5.7(a). Moreover, the combined graph in Fig.5.7(b) contains a forbidden sequence leading to an erroneous state.

So, we have to add some control information to both the state and sequence diagrams (as depicted in Fig.5.8) to make the two controllers safely cooperate to realize the intended interaction. In the enhanced sequence diagram of Fig.5.8(c), when a train approaches a crossing border, the barrier controller sends now an acknowledgment message *GoAhead* to the train controller in order to let it pass. Otherwise, if there is any



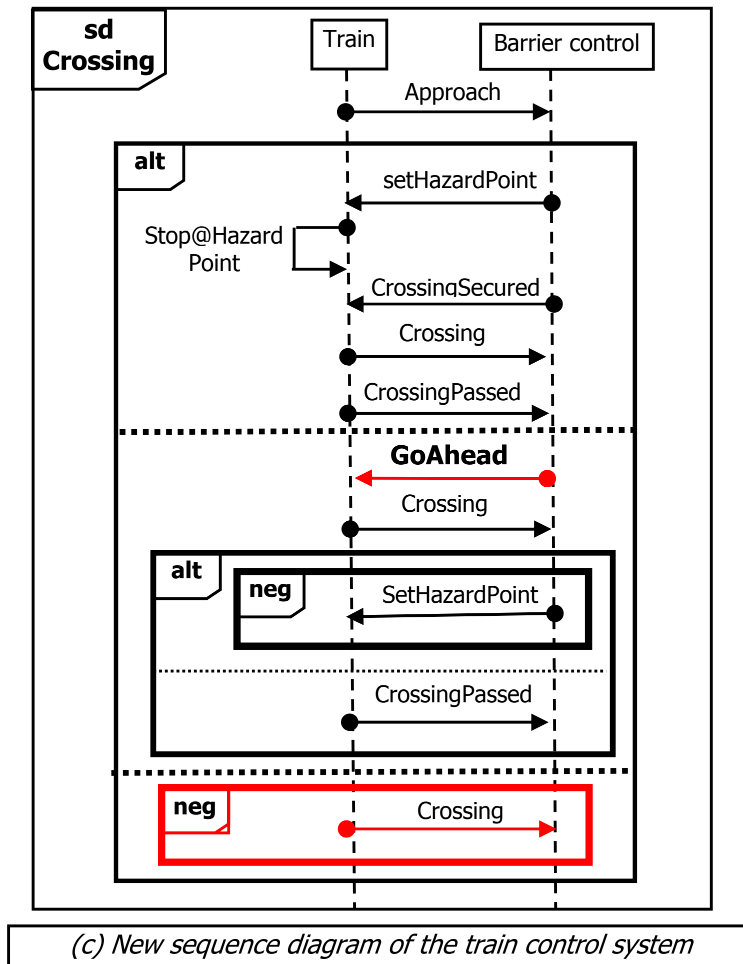


Figure 5.8: Enhanced state and sequence diagrams.

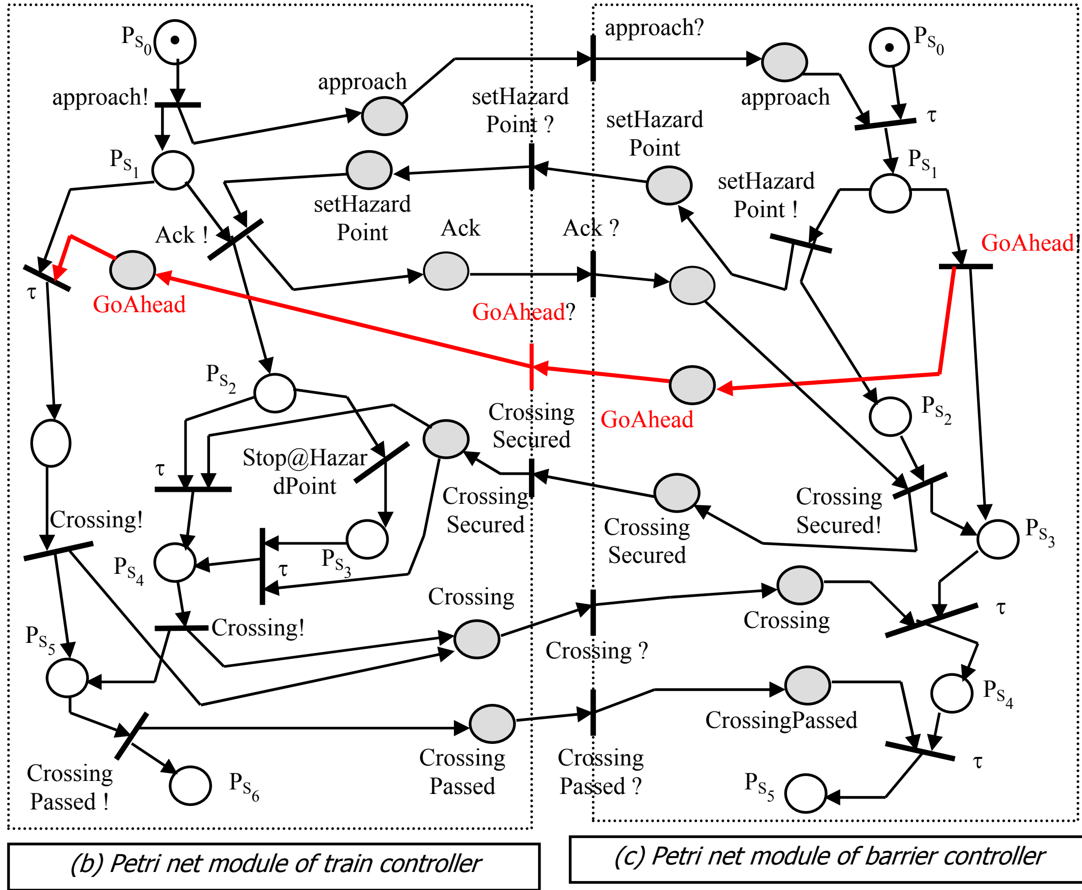


Figure 5.9: Petri nets of improved state machines of the train control system.

problem, the barrier controller can send the stop command *setHazardPoint* to the train. Similarly, the state machines of train and barrier controllers are improved as follows (see Fig.5.8(a) and (b)):

- In the train state machine (Fig.5.8(a)), we put the trigger event *goAhead*? on the arc from S_1 to S_5 which lets the train state machine continue its crossing without stopping.

- In the barrier state machine (Fig.5.8(b)), we rename the triggered event on the transition from S_1 to S_3 with *GoAhead*! instead of *CrossingSecured*!. This helps avoid the confusion between event *GoAhead*! which lets the train continue its crossing without stopping and event *CrossingSecured*! which lets the train resume its crossing after it has been asked to stop at a hazard point.

For the consistency checking of these new diagrams, we use our exploration algorithm to derive from the Petri net of Fig.5.9 the partial behavioral graph of the new train controller (given in Fig.5.10(a)). Next, this is combined with the specification

graph (of Fig.5.11) related to the new sequence diagram. Notice that we obtain a deadlock-free and error-free graph (given in Fig.5.12) which is branching simulated by the partial graph of the train controller of Fig.5.10(a)).

Moreover, Fig.5.13(a) represents the full synchronization graph of the train system components, which is deadlock-free and erroneous-free subgraph of that of the whole state space in Fig.5.13(b). To get a *full compatibility*, we have to add more control events to both the two components of the train system to compel them once combined, to carry out their interactions in a way similar to their individual progress within a controlled environment (i.e., according to our approach for partial state exploration of components).

The theorem below, asserts that the whole system realizes any scenario which is fulfilled by all of its involved compatible components and vice versa. Indeed, when two components fulfill some sequence diagram, it means that they cooperate correctly at least in respect of this scenario.

Theorem 23 *Let SD denote available sequence diagrams and let $SD[C_i]$ denote the subset of sequence diagrams having the component C_i as a participant.*

1. *If all components C_i of Sys are fully compatible, two by two, then:$\forall C_i \in Sys, C_i \models SD[C_i] \Leftrightarrow Sys \models SD$.*
2. *Moreover, If the components are compatible, two by two, then: $\forall C_i \in Sys, C_i \not\models SD[C_i] \Rightarrow Sys \not\models SD$.*

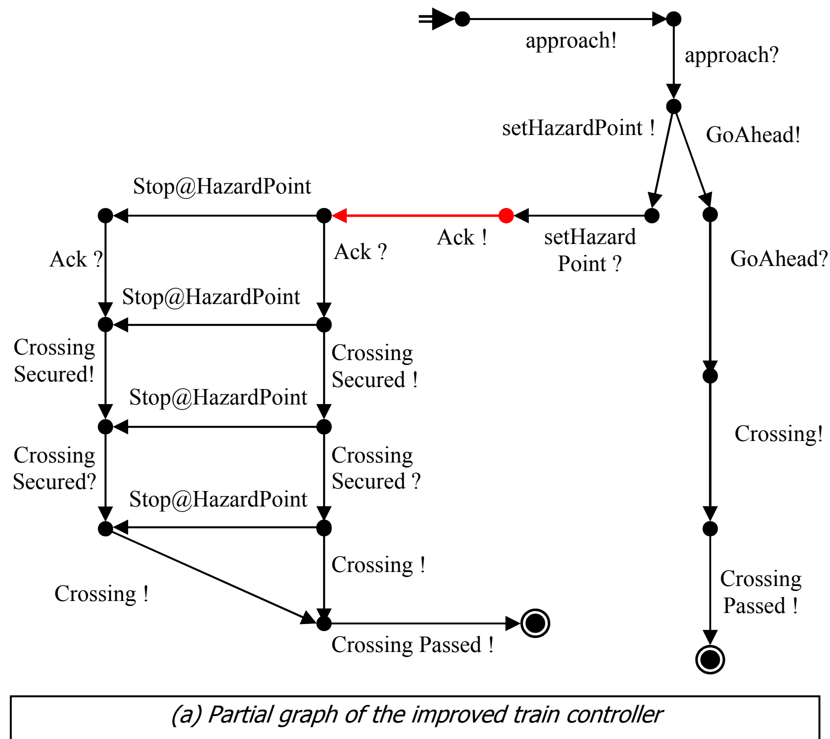
Proof.

(1) ($\Rightarrow$) We prove the theorem for a subsystem Sys consisting of two fully compatible components C_1 and C_2 against one sequence diagram sd in which these are involved. Let $C_1 \models sd$ and $C_2 \models sd$. Hence by Def.21, $H_{\Gamma_1}(\llbracket C_1 \rrbracket \otimes_P \llbracket sd \rrbracket) \preceq_{br} \llbracket C_1 \rrbracket$ where $\Gamma_1 = \Sigma(\llbracket sd \rrbracket) \setminus \Sigma(\llbracket C_1 \rrbracket)$, and $H_{\Gamma_2}(\llbracket C_2 \rrbracket \otimes_P \llbracket sd \rrbracket) \preceq_{br} \llbracket C_2 \rrbracket$ where $\Gamma_2 = \Sigma(\llbracket sd \rrbracket) \setminus \Sigma(\llbracket C_2 \rrbracket)$.

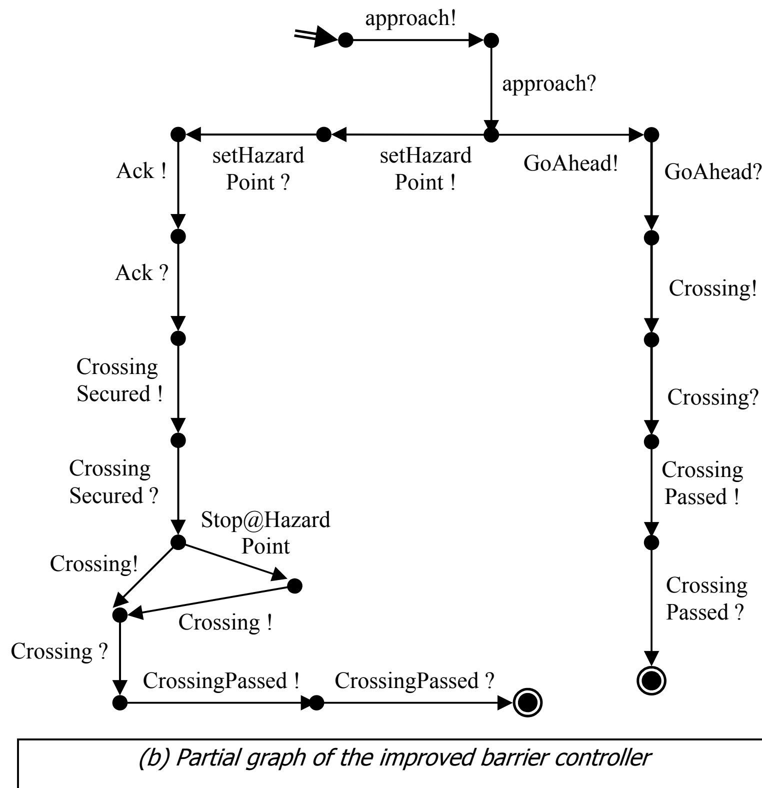
By Proposition 20, $H_{\Gamma_2}(\llbracket C_2 \rrbracket \otimes_P \llbracket sd \rrbracket) \preceq_{br} \llbracket C_2 \rrbracket \implies \llbracket C_1 \rrbracket \otimes_P H_{\Gamma_2}(\llbracket C_2 \rrbracket \otimes_P \llbracket sd \rrbracket) \preceq_{br} (\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket) \implies H_{\Gamma_1 \cup \Gamma_2}((\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket) \otimes_P \llbracket sd \rrbracket) \preceq_{br} (\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket) \implies H_{\Gamma_3}((\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket) \otimes_P \llbracket sd \rrbracket) \preceq_{br} (\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket)$ where $\Gamma_3 = \Gamma_1 \cup \Gamma_2 = \Sigma(\llbracket sd \rrbracket) \setminus \Sigma((\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket)) = \Sigma(\llbracket sd \rrbracket) \setminus \Sigma(\llbracket C_1 || C_2 \rrbracket)$.

Since the subsystem Sys is consisting of two fully compatible components C_1 and C_2 then $(\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket)$ is isomorphic to $(\llbracket C_1 || C_2 \rrbracket)$. We easily deduce thus, $H_{\Gamma_3}((\llbracket C_1 || C_2 \rrbracket) \otimes_P \llbracket sd \rrbracket) \preceq_{br} (\llbracket C_1 || C_2 \rrbracket)$.

($\Leftarrow$) Vice versa, we can deduce that when the subsystem Sys consisting of fully compatible components fulfills sd then $H_{\Gamma_3}((\llbracket C_1 || C_2 \rrbracket) \otimes_P \llbracket sd \rrbracket) \preceq_{br} (\llbracket C_1 || C_2 \rrbracket) \implies H_{\Gamma_3}((\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket) \otimes_P \llbracket sd \rrbracket) \preceq_{br} (\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket)$ where $\Gamma_3 = \Gamma_1 \cup \Gamma_2 = \Sigma(\llbracket sd \rrbracket) \setminus \Sigma((\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket)) = \Sigma(\llbracket sd \rrbracket) \setminus \Sigma(\llbracket C_1 || C_2 \rrbracket)$. Hence, each component C_i should satisfy



(a) Partial graph of the improved train controller



(b) Partial graph of the improved barrier controller

Figure 5.10: Partial graphs of improved state machines of Fig.5.9.

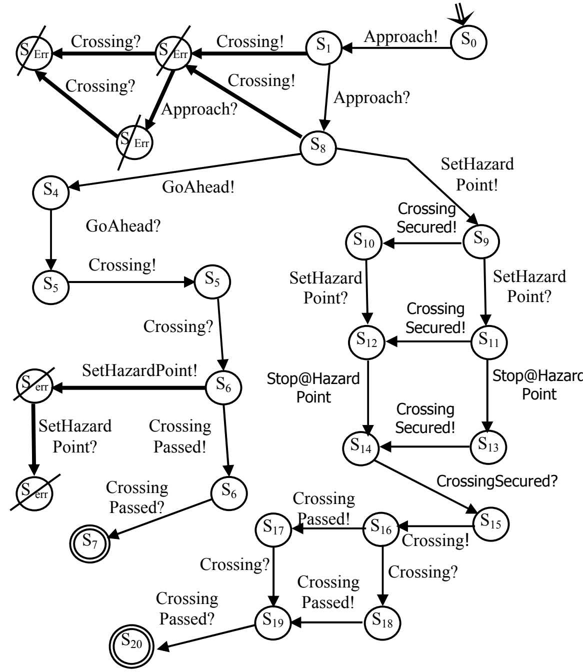


Figure 5.11: The behavioral graph of the improved sequence diagram.

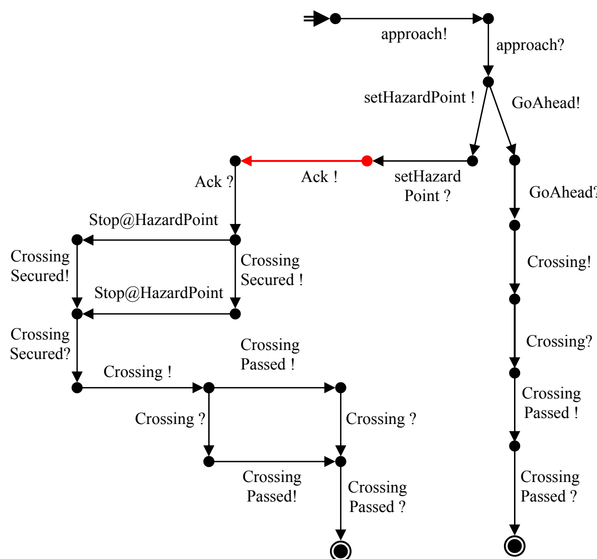


Figure 5.12: The full synchronization of behavioral graphs related to the improved sequence diagram and the Train Controller StateChart.

sd: $H_{\Gamma_i}(\llbracket C_i \rrbracket \otimes_P \llbracket sd \rrbracket) \preceq_{br} \llbracket C_i \rrbracket$, otherwise their combination will not satisfy sd, which is absurd.

(2) We have to prove that when the subsystem Sys, consisting of compatible components, fulfills sd then its components satisfy also this sequence diagram.

Suppose $C_1 \nvDash sd \wedge Sys \vDash sd$. So, we get also $H_{\Gamma_1}((\llbracket C_1 \rrbracket \otimes_P \llbracket sd \rrbracket) \npreceq_{br} (\llbracket C_1 \rrbracket) \implies H_{\Gamma_3}((\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket) \otimes_P \llbracket sd \rrbracket) \npreceq_{br} (\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket)$ where $\Gamma_3 = \Gamma_1 \cup \Gamma_2$. This means, $H_{\Gamma_3}((\llbracket C_1 || C_2 \rrbracket) \otimes_P \llbracket sd \rrbracket)$ would contain erroneous or deadlock states. However, according to Proposition 18, Since the subsystem Sys is consisting of two compatible components C_1 and C_2 then $(\llbracket C_1 \rrbracket \otimes_P \llbracket C_2 \rrbracket)$ is a deadlock-free and error-free subgraph of $(\llbracket C_1 || C_2 \rrbracket)$. Hence, we deduce that also $Sys \nvDash sd$, which is absurd.

■

Remark: In practice, only a fragment of intended behaviors of the system is captured by sequence diagrams. Then, even though the consistency checking is successfully achieved between designed state and sequence diagrams, this does not mean the absence of some inapparent incompatibilities between components. Hence, the choice of *good* scenarios is very important to uncover design flaws.

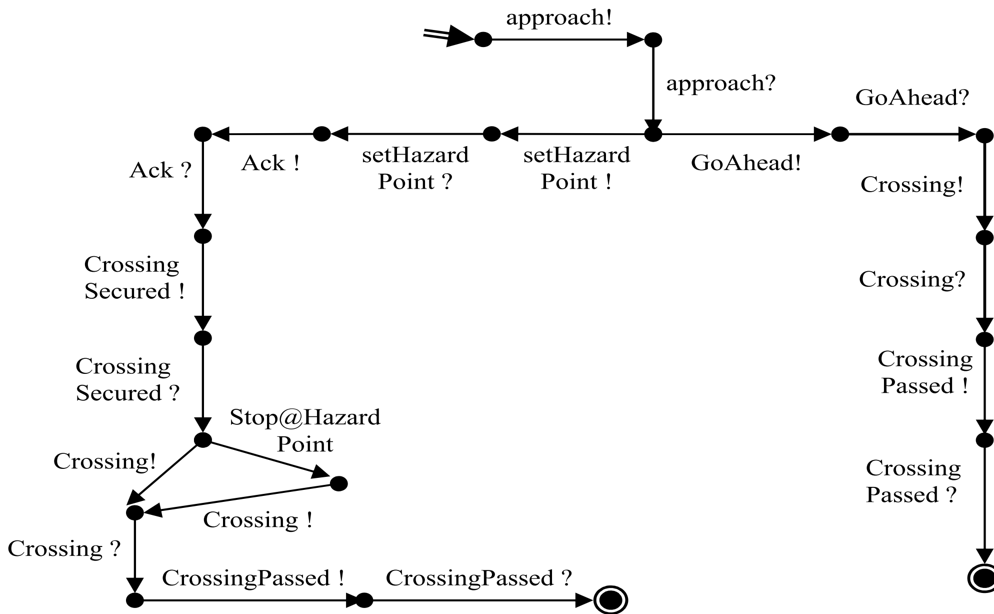
5.4.6 Fulfillment relation within true concurrency semantics

The behavioral model that can capture subtleties about true concurrency, might be a non-interleaving model which would be suitable to capture the concept of independent actions (i.e., absence of causality). We opt for namely asynchronous transition system (denoted ATS) [31] whose formal definition (Definition 1) has been already given in Chapter 3.

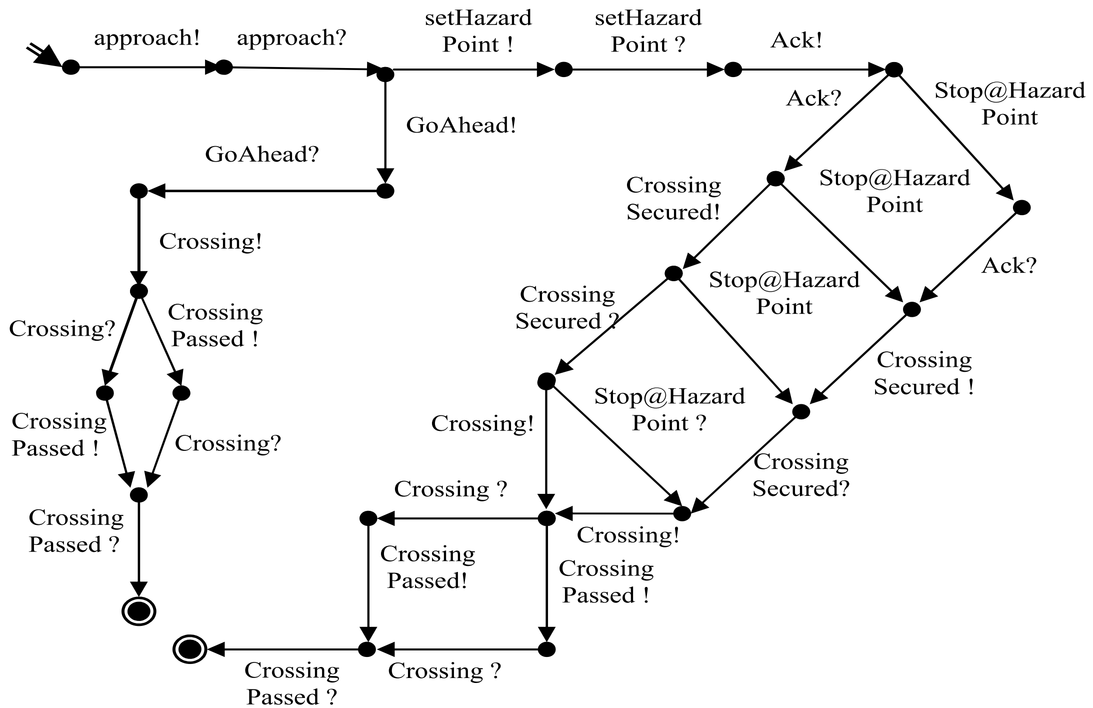
The next question that arises, is about what observation criterion can be used to compare the new graphs. As the classical simulations are based on an interleaving semantics, we need to introduce a new (branching variant of the forth-back) simulation within the setting of true concurrency semantics. It is worth noting that there are two reasons for our choice of this particular model and its observational equivalence rather than other ones; first, these are straightforward and smooth extensions of the concepts introduced before in the interleaving context, and second, the branching forth-back simulation turns out to be a branching simulation when considering only classical transition systems.

Definition 24 *A forth-back simulation is a binary relation* $\preceq_{fb}$ *such that* $p \preceq_{fb} q$ *if and only if :*

- *if* $p \in \mathcal{F}_1$ *then* $q \in \mathcal{F}_2$*.*
- *if* $p \in \mathcal{E}_1$ *then* $q \in \mathcal{E}_2$*.*



(a) The full synchronization of behavioral graphs of the improved barrier and train controllers' StateCharts.



(b) The whole state space of the train control system.

Figure 5.13: Modular consistency checking of improved components.

- $\forall a \in \Sigma, \forall p' \in Q_1 : p \stackrel{\tau}{\Longrightarrow} p_1 \stackrel{a}{\longrightarrow} p'$ *then* $\exists q' \in Q_2 : q \stackrel{a}{\longrightarrow} q' \wedge p_1 R q \wedge p' R q'$.
- $\forall a \in \Sigma, \forall p' \in Q_1 : p' \stackrel{\tau}{\Longrightarrow} p_1 \stackrel{a}{\longrightarrow} p$ *then* $\exists q' \in Q_2 : q' \stackrel{a}{\longrightarrow} q \wedge p_1 R q' \wedge p' R q'$.

This definition ensures that the second graph faithfully simulates the first graph by capturing, as well, the independence relation among events. Next, we give the definition of the validity relation between state machines and sequence diagrams in the context of non-interleaving semantics.

Definition 25 $C \models sd$ *iff* $H_\Gamma(\llbracket C \rrbracket \otimes_P \llbracket sd \rrbracket) \preceq_{fb} \llbracket C \rrbracket$. *Let SD be a set of sequence diagrams.* $C \models \boldsymbol{SD}$ *iff* $\forall sd \in \boldsymbol{SD}, C \models sd$.

Within this context, Proposition 22 remains valid (since $\preceq_{br} \subseteq \preceq_{fb}$) so that the component fulfilling a sequence diagram never achieves its forbidden scenarios.

5.5 Conclusion

This chapter has presented a component-based formal method for compatibility and consistency checking of UML dynamic diagrams. We have given an exhaustive method that takes care of the features of UML 2.x state and sequence diagrams, particularly the forbidden sequences introduced by the interaction operators *neg* and *assert*.

Afterwards, we have tailored this approach to the component-based framework. We have presented a new method for extracting partial state spaces of a system with regard to its interacting components, one at a time, in such a way that each partial behavioral graph contains, in addition to the component actions, those which are causally related to them in order to cope with the asynchronous communication between communicating state machines. We have made use of these graphs to analyze the compatibility and composability between components. Also, we proposed a fitting comparison way, based on branching simulation, to modularly check whether the components state machines fulfill the interaction-based specification. We have also shown that this relation is preserved by our composition operation. In so doing, we can assert that partial checking results could be generalized onto the combined systems.

Chapter 6

State Machine Refinement & Substitutability

6.1 Introduction

According to works on formal semantics of concurrent languages [67, 91, 130, 128, 129, 131, 132], a complete definition of such semantics needs adding some observational criterion to compare processes. As expected, observational relations for active objects are subtyping relations asserting that a new object still behave as the old one in respect of the well-suited observational criteria.

After introducing the computational model for state-based objects in previous sections, we consider now the inheritance/refinement and substitutability issues of state machines, as it arises, in specifying behaviors using UML StateCharts ([50]).

In this setting, UML state machines are always refined and changed to depict intended behaviors of new objects/components linked by inheritance/substitution relationships to their predecessors. However, UML state machines, as well as Petri nets, are high level formalisms that cannot be directly compared. In other words, these diagrams shall be mapped into a more basic and common formalism, namely, the reachability graphs obtained through our translation process. These behavioral graphs of components can thus be compared to each other modulo any adequate observational equivalence (such as branching and back-forth bisimulations [130, 31]).

6.1.1 Liskov behavioral substitution principle

It should be noted that the substitutability of state machines is a key topic to the modeling and verification of object and component based systems. Indeed, software systems always evolve throughout the product life cycle. That is to say; objects

and components are transformed as requirements change, bugs are discovered and fixed. Evolution implies the removal of previous components and addition of new ones refined and augmented with new services. However, besides any potential benefits of inheritance and refinement in terms of implementation reuse, the refinement process may raise two kinds of problems in the new behavior: unavailability of previously provided services and violation of global correctness properties that were previously respected [88, 134, 18].

In view of these risks and in order to find out the accurate criterion to compare new components against removed ones, it is worthy to recall the Liskov Substitution Principle [88]: "*If for each object* o_1 *of the type S there is an object* o_2 *of the type T such that for all programs P defined in terms of T, the behavior of P is unchanged when* o_1 *is substituted for* o_2 *then S is a subtype of T*". Accordingly, the substitutability problem can be defined as the verification of two conditions [88, 18]:

1. The *containment* criterion that requires every behavior of the old object (or component) to be also a behavior of the new one. In other words, a refined object must continue to supply all services (actions) which the basic ancestor was offering.

2. Whereas the second criterion is about *compatibility* requiring that correctness properties which were previously proven must remain valid in the new composite system. In fact, a fragment viewed in isolation, cannot be meaningfully checked and its validity can only be expressed in terms of reasonable assumptions made by users of the entire system. One must thus think about reasonable assumptions related to the intended interactions of the component within its environment.

Accordingly, it becomes clear that the informal refinement policies of UML state machines as defined in terms of syntactic rules within UML specification documents [101, 103] are not adequate in view of previous criteria. Instead, it is more interesting to formally deal with substitutability at the semantics level of state machines rather than the syntactic aspects. We, thus, need to propose new subtyping relations which are based on strong observational equivalences and which cope efficiently with service availability when dealing with refinement and inheritance of state-based objects. Indeed, substitutability checking should go beyond asserting that traces of a previous object are prefixes of some traces of the new one since traces based subtyping relations are too weak and do not handle safety properties. Thus, many efforts focus on use of failures based preorders [37, 57, 133] to deal with this issue. Nevertheless, these relations are not too strong to check faithfully substitutability of active components which concurrent and reactive behaviors may deeply alter the service availability and correctness properties.

In this setting, we discuss in view of previous criteria the shortcomings of informal refinement policies of UML StateCharts as defined in terms of syntactical rules within UML specification documents [103, 102]. Instead, we formally deal with substitutability at the semantics level of StateCharts rather than the syntactical aspects and we accordingly propose new subtyping relations [50] that are based on strong branching bisimulations and which cope efficiently with service availability when dealing with refinement and inheritance of state-based objects. Moreover, we address the issue of substitutability in the non-interleaving context where we propose a fitting relation based on a new variant of back-forth bisimulation. Recall that our objects and components are modeled using StateCharts which semantics are provided with transitions systems, whereas their interaction protocols are depicted by UML 2.0 sequence diagrams [103] as those capture various safety and liveness properties of components in an intuitive and visual way. According to the compatibility requirement, we prove then the correctness of the substitutivity relations over StateCharts in relation to any (fragment based) specifications they fulfill including those given in terms of sequence diagrams [50].

6.1.2 Motivation and related work

According to the substitutability criteria, it is not enough that execution paths of a previous component be preserved in the new component behavior which is often augmented with new services. Substitutability checking should go beyond asserting that traces of the previous component are only prefixes of some traces of the latter component.

Obviously, such simulation and traces based subtyping relations [72, 82, 26, 27] are too weak in the context of reactive systems because they do only comply with the containment requirement and do not deal with the mutable shared objects and active components. Indeed, these relations are not able to preserve correctness properties which may be violated by the new behaviors that these relations entirely ignore. For instance, two components having the same traces may react differently to their environment stimuli when considering their branching structures.

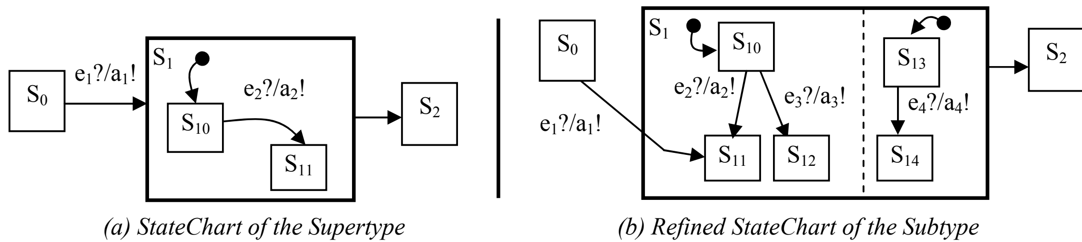
Therefore, much work focuses on using failures based preorders [98, 133, 57, 37] and weak bisimulations [25, 125] where only new behaviors are hidden. However, we deem that these kinds of relations are not strong enough to check faithful substitutivity of active components whose event-driven and time-constrained behaviors may deeply alter the service availability and correctness properties in reactive systems:

A For instance, considering the class of non-optimal subtyping relations defined in [98, 25, 125, 133, 57, 37] (such as implementation, weak, and safe subtyping re-

lations) we notice that only containment requirement is fulfilled and that service availability preservation could be not comprehensively guaranteed because only *weak observational criteria* are used to compare branching structures of components. Consequently, if a previous component was able at some point to react to some stimuli as soon as they occur, then its subtyping - according to relations of this class - can achieve such reactions after or interleaved with its new actions. Unfortunately, because of performance of new activities the enhanced component would ignore (until completion of that new sequence) its reaction to stimuli which the old component used to immediately process. Thus the subtype runs the risk to be unable to react within reasonable deadlines and match expectations of its environment, thereby, leading possibly to deadlocks or violation of safety properties that were previously fulfilled.

B Although the class of optimal subtyping relations [133, 57, 37] strengthened the observational criteria of comparison by using a new hiding operator for computing failures, we notice that these were not able to deal with components, the descriptions of which are given at abstract levels involving non deterministic cases. Moreover, these failures based approaches do not match with severe requirements of substitutability for reactive systems in the sense that service availability is preserved in subtypes at intermediate points within sequences of the new actions but not during their performance. That is, a new component which executes a new action would be unable to react to any stimulus (even if urgent) that its supertype used to accept at an equivalent stage of its progress. This is why we utilize as observational criteria strong branching bisimulations that efficiently deal with silent steps and/or partial order models in order to overcome flaws of existing subtyping relations, at least to make the Liskov Substitution Principle [88] hold for active components in context of reactive, real time and concurrent systems.

In view of the previous discussion, we still use transition systems as basic behavioral formalism of state machines in the context of interleaving and true concurrency semantics, and we propose new subtyping relations that are based on *strong* branching bisimulation which takes into account the hiding of new *added* services and possibly some old *removed* required services ([50]). Note that we propose to use two categories of silent actions to deal with removal of old services that are deprecated in new components in such a way these be differently handled from silent actions depicting new services. Accordingly, observational criteria and subtyping relations had to be updated therein.



(a) StateChart of the Supertype

(b) Refined StateChart of the Subtype

Figure 6.1: UML state machine refinement.

6.2 State machine refinement in UML specifications

State machine refinement as defined in UML specification [101, 103] does not specify or favor any specific policy of refinement. Instead, it simply provides syntactical rules which allow subtyping (behavioral compatibility), inheritance (implementation reuse), or general refinement policies. These rules are discussed below with regard to Liskov substitutability principle.

Strict Inheritance: The rationale behind inheritance policy is to promote reuse of implementation rather than preserving behavior. States and transitions can be added. A refined state has some of the same incoming transitions but a bigger set of outgoing transitions. It may have more substates, and may change its concurrency attribute. Moreover, a refined transition may go to a new target state but should have the same source and a refined guard may have a different guard condition. Similarly, the rules of this weaker version of refinement lead to violation of both containment and compatibility criteria of Liskov Substitutability Principle.

General Refinement: States and transitions, herein, can be added and deleted without formal requirements on the properties and relationships of the refined state machine element and the refining element. For instance, a refined state may have different outgoing and incoming transitions. Moreover a refined transition may leave from a different source and go to a new target state and refined guards may have different guard conditions. Consequently, this open policy obviously transgresses behavioral subtyping principles.

Subtyping: The refinement policy for subtyping [101, 103] aims at guarantying the substitutability principle through the preservation in the subtype of pre/post condition relationships of applying events / operations on the supertype. Hence, states and transitions are only added, not deleted. A refined state (see Fig.6.1) has the same outgoing transitions, but may add others, and a different set of

incoming transitions. It may have a bigger set of substates, and it may change its concurrency property from *false* to *true*. A refined transition may go to a new target state which is a substate of the state specified in the base class. This comes to guarantee the post-condition specified by the base class. A refined guard has the same guard condition, but may add disjunctions so that pre-conditions are weakened rather than strengthened.

Let us now check whether these subtyping rules ensure containment criterion. In the base StateChart (see Fig.6.1(a)), we can get the computational path $(s_0 \stackrel{(e_1?,a_1!)}{\longrightarrow} (s_1, s_{10}) \stackrel{(e_2?,a_2!)}{\longrightarrow} (s_1, s_{11}))$. Such a path cannot be taken in the refined StateChart (after concealing new arcs) (see Fig.6.1(b)) because receipt of the event e_1 causes the control to be shifted from s_0 into the substate s_{10}, in such a way that any dispatched instance of event e_2 can never be processed. Recall that an event receipt represents a service request whose handling may be crucial in safety-critical systems. Similarly, above subtyping rules may lead to violation of the compatibility criterion. For instance even if we keep in the refined StateChart the arc labeled with $(e_1?/a_1!)$ from s_0 to the edge of s_1, old services availability may be lost. In fact, when the control is in substate s_{10} and the event e_3 is dispatched then the arc to substate s_{12} will be taken. So, while the control is in substate s_{12}, the event e_2 may be received but can never be processed although the old StateChart was able to process it at this status through the same computational path. These examples clearly show that above syntactic rules are not efficient enough to ensure behavioral conformity in respect of Liskov substitutability principle.

Given that all above informal and syntactical rules over StateCharts do not cope efficiently with the substitutability issue for reactive objects, we next present and discuss new behavioral subtyping relations among StateCharts modeling behaviors of active objects [50]. Below wherever there is no ambiguity, the terms of component and StateChart (modeling its behavior) are confused, while their computational model is given as a transition system.

We address as well the issue of substitutability in the case of true concurrency semantics. Since previous subtyping relations are no longer suitable in this context, we propose therein an adequate relation based on a new branching variant of forth and back bisimulation upon asynchronous transitions systems and discuss its properties.

6.3 Preliminary concepts: strong branching bisimulation

As mentioned before, the semantics function (note it $[\![.]\!]$) maps the behavioral model of any component C (given in terms of an UML StateChart) into a labeled transition

system G as follows: $[\![C]\!] = G = \langle Q, \hookrightarrow, \Sigma, q_0 \rangle$ where:

- Q is the set of all reachable component configurations (or status) with the starting node q_0 as its initial configuration. Note that for sake of simplicity, configurations are referred to as states of component C, although configurations are thoroughly different from the concept of states used in high level languages (as StateCharts).

- $\hookrightarrow \subseteq Q \times \Sigma \times Q$ are transitions between nodes labeled with events of Σ. This relation can be obviously extended as follows: a compound transition ($q_0 \stackrel{\omega}{\Longrightarrow} q_n \mid \omega = (a_1..a_n) \in \Sigma^*$) is introduced to denote a sequence of basic transitions labeled with actions names of Σ such that: $\exists q_1, ..., q_{n-1} \in Q \wedge q_0 \stackrel{a_1}{\longrightarrow} q_1 \stackrel{a_2}{\longrightarrow} ...q_{n-1} \stackrel{a_n}{\longrightarrow} q_n$.

- Σ is a set of events. Note that the interface signature of C is simply the subset $\Sigma_I \subseteq \Sigma$ that contains interactions of C with its environment, each of which is related either to a provided service or a required service. However, actions from the set $\Sigma \setminus \Sigma_I$ are internal operations which the component carries out without any cooperation from its environment.

We recall that when substituting any component, its execution paths should be preserved by the new component behavior which can be augmented with new actions. Thus, traces of the previous component are prefixes of some traces of the latter component. Nevertheless, a subtyping relation based on traces is too weak because it is not able to take into account deadlock behaviors. Many works, hence, focus on using failures based preorders. However, we think that these kinds of relations are not strong enough to check faithfully substitutivity of active components which behaviors may change deeply w.r.t. old ones. Indeed, such preorders are not the most discriminatory powerful and do not fit well with many cases where new component signature does not preserve some specific required services since those ones do not affect in any way the capabilities of the new component in terms of provided services.

We recall below the definitions of existing bisimulation variants [91, 128, 129, 130, 132] and introduce a new one we use in defining our substitutability relations.

Let G_1 and G_2 be two labeled transition systems (always referred to as graphs for more conciseness): $G_i = \langle Q_i, \hookrightarrow_i, \Sigma_i, q_0^i \rangle$ for $i = 1, 2$ and let Σ denote the set $\Sigma_1 \cup \Sigma_2$.

Definition 26 *A (strong) bisimulation is a binary relation $\mathcal{R} \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2$, $p\mathcal{R}q$ iff:*

- $\forall a \in \Sigma, \forall p' \in Q_1 : p \stackrel{a}{\longrightarrow} p'$ *implies* $\exists q' \in Q_2 : q \stackrel{a}{\longrightarrow} q'$ *and* $p'\mathcal{R}q'$,

- *and conversely,* $\forall a \in \Sigma, \forall q' \in Q_2 : q \stackrel{a}{\longrightarrow} q'$ *implies* $\exists p' \in Q_1 : p \stackrel{a}{\longrightarrow} p'$ *and* $p'\mathcal{R}q'$.

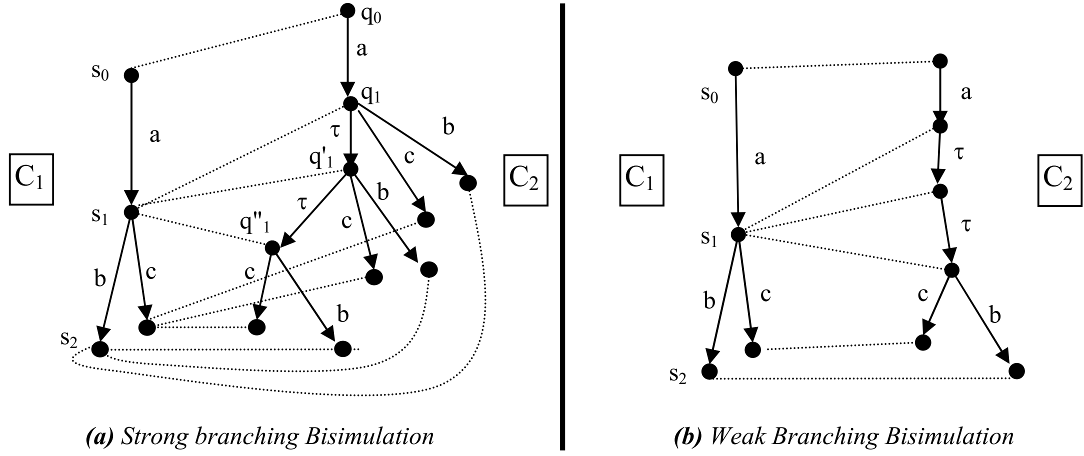


Figure 6.2: Strong branching bisimulation versus weak branching bisimulation.

Two graphs G_1 and G_2 are bisimilar (written $G_1 \approx G_2$) when their initial nodes are related by some strong bisimulation $\mathcal{R}$ ($q_0^1 \mathcal{R} q_0^2$).

This relation [91] is too strong and does not take into account the unobservable action τ which is important to model new services to be hidden. So, we make use of the branching variant of bisimulation [128, 129, 130, 132] considering the branching structure even in the presence of τ-actions in such a way that internal decisions are preserved.

Definition 27 *A branching bisimulation is a binary relation $\mathcal{R} \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2$, $p\mathcal{R}q$ if and only if:*

- *$\forall a \in \Sigma \cup \{\tau\}, \forall p' \in Q_1 : p \xrightarrow{a} p'$ then either: $a = \tau$ and $p'\mathcal{R}q$, or $\exists q_1, q' \in Q_2 : q \overset{\tau}{\Longrightarrow} q_1 \xrightarrow{a} q'$ and $p\mathcal{R}q_1$ and $p'\mathcal{R}q'$,*
- *and conversely, $\forall a \in \Sigma \cup \{\tau\}$, $\forall q' \in Q_2 : q \xrightarrow{a} q'$ then either: $a = \tau$ and $p\mathcal{R}q'$, or $\exists p_1, p' \in Q_1 : p \overset{\tau}{\Longrightarrow} p_1 \xrightarrow{a} p'$ and $p_1\mathcal{R}q$ and $p'\mathcal{R}q'$.*

Two graphs G_1 and G_2 are branching bisimilar (written $G_1 \approx_{br} G_2$) when their initial nodes are related by some branching bisimulation $\mathcal{R}$ ($q_0^1 \mathcal{R} q_0^2$).

Two states p and q are equivalent if they can perform the same observable actions even though interleaved with τ-actions. If some state p' can be reached via τ-steps from p (see Fig.6.2(a)), then it should be also equivalent to the state q, i.e. p' remains able to achieve all actions the state q can do either directly or via τ-steps. So, the capabilities (services) of an old component may sometimes be only preserved indirectly in the new component such that some of these services become unavailable at some intermediate points along a chain of τ-steps.

Note that τ-action is not under the sole control of the component and requires the cooperation of its environment. When plugging the new component instead of the old one in a composite system, new actions are used to yield new services. As these new interactions may be synchronous, they may lead to unavailability of old services until they finish.

Accordingly, this relation does not comply well with the substitutability constraints between active components by allowing previous services to be interleaved with τ-actions without compelling intermediate states to directly provide the previous actions. To overcome this drawback in a context of mutable shared objects, we introduce our new bisimulation which is stronger than the classical branching bisimulation in respect of services availability even if throughout τ-progress [50].

Definition 28 *A strong branching bisimulation is a binary relation $\mathcal{R} \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2$, pRq iff:*

- $\forall a \in \Sigma \cup \{\tau\}, \forall p' \in Q_1 : p \xrightarrow{a} p'$ *then either:* $a = \tau$ *and* $p'\mathcal{R}q$, *or* $\exists q' \in Q_2 : q \xrightarrow{a} q'$ *and* $p'\mathcal{R}q'$,
- *and conversely,* $\forall a \in \Sigma \cup \{\tau\}$, $\forall q' \in Q_2 : q \xrightarrow{a} q'$ *then either:* $a = \tau$ *and* $p\mathcal{R}q'$, *or* $\exists p' \in Q_1 : p \xrightarrow{a} p'$ *and* $p'\mathcal{R}q'$.

Two graphs G_1 and G_2 are strongly branching bisimilar (written $G_1 \approx_{sbr} G_2$) when their initial nodes are related by some strong branching bisimulation $\mathcal{R}$ ($q_0^1 \mathcal{R} q_0^2$).

Note that two states p and q are equivalent if they can perform *directly* the same observable actions in order to reach pairs of states always equivalent. Furthermore, if some state p' can be attained by one τ-step from p then it should be also equivalent to the state q and preserve the same capabilities (see Fig.6.2(b)). This strong constraint is due to the fact that the unobservable action is not an internal one under the control of the only involved component. Indeed, the unobservable action in our setting represents some new services that are not visible to clients of the previous component. So, even though τ is performed these clients should always be able to use the same services as those which were available before the τ-progress.

As the strong branching bisimulation is weaker than the strong bisimulation and stronger than the classical branching one, we deduce from equivalence hierarchies given in [91, 129] the following one (which proof is similar to that of proposition 40):

Proposition 29 $\approx \ \subseteq \ \approx_{sbr} \ \subseteq \ \approx_{br}$.

6.4 Behavioral subtyping relations within interleaving semantics

The rationale behind behavioral substitutability is that (reactivity) capabilities of a previous state machine should be always preserved and often augmented in the refined one. As a result, the state machine signature changes raising a question about how this modification would be done. In our opinion, in spite of the way the internal structure of a state machine is refined, the main concern is to preserve and augment provided services (reactions to triggering events) making it possible to continue to fulfill any request related to previous services from its environment (i.e. old clients). Thus, substitutability checking amounts first to asserting that behaviors of old and new state machines are equivalent modulo an adequate subtyping relation provided that new added services (added actions) are suitably concealed to old users of the refined state machine.

On the other hand, enhancements operated on the internal implementation may lead to the removal of some required services which become unnecessary as well as the addition of new ones needed to realize these improvements.

Besides this, it is also possible to include internal (and unobservable) improvements on how these services are carried out. Thus, some required services can be removed as well as other required services can be appended in the new component. Indeed, some improvements can be operated on the internal behavior of this component such that some previous required services are no longer required in the new component. In addition, a number of new services would be claimed from its environment to achieve these improvements.

So, what are the consequences of such assertions on the definition of substitutability? Obviously, the observable behavior of the new component would include new paths containing additional provided and required services and discard all previous paths encompassing required services that have been removed.

Consequently, we distinguish two main cases:

6.4.1 Case 1: The new component adds new provided/required services without removing any required services

We obviously have to discard the strong bisimulation because it does not take into consideration the unobservable action τ which models all new added services we have to hide. Note that actions are under control of both its owner state machine and its environment because these actions require the cooperation of both the two parts. Instead, we use others kinds of bisimulation taking care of the unobservable actions

like branching bisimulation variants.

Let C_1, C_2 be the state machines of two components. $\Sigma(C_i)$ denotes the set of events raised by a state machine C_i (i=1,2).

Some events of $\Sigma(C_i)$ are internal actions, whereas some others depict interactions with its environment (to provide or require services).

$[\![C_i]\!]$ is the transition system modeling the basic behavior of C_i which we produce thanks to our translation method given in previous chapters. We use a hiding operator (note it H) as defined in process algebras to hide actions related to new services, making them unobservable to some clients which still use old services of the refined state machine. Moreover, new unobservable events are different from internal actions because the latter ones are completely under control of the state machine, whilst the former events are under control of both the involved state machine and its environment (i.e. cooperating state machines). New services are hence unobservable only by some clients who view the enhanced component as if it was the old one and thus still use only its old services. Consequently, we consider here only hidden actions depicting claimed old services and discard internal controlled events of a component which do not affect its external behavior w.r.t. availability of services in view of its users.

Let $N = \Sigma(C_2) - \Sigma(C_1)$ be the set of new services. $H_N([\![C]\!])$ denotes the behavior model of C where all events of N are relabeled to τ.

$$H_N(e) = \begin{cases} e & \text{if e} \notin \text{N} \\ \tau & \text{if e} \in \text{N} \end{cases}$$

The basic idea of our comparison criteria is that old services which were available at some states in the old component should not be affected by the performance of new (hidden) actions in the new component and so have to remain available independently from them (i.e., before and after the τ-action).

Hence, we give below the optimal substitutability relation which is faithful enough in terms of preservation of capabilities for active components:

Definition 30 *$C_2 \preceq_{sbr} C_1$ if and only if $H_N([\![C_2]\!]) \approx_{sbr} [\![C_1]\!]$.*

The main target of this definition is to ensure the state machine C_2 to behave as the state machine C_1 (in respect of an external client) whichever the state that C_2 has reached.

Every time C_1 is able from a state s_1 (see Fig.6.2(b)) to interact with its environment (by offering or demanding a service a) and reach some stable state s_2, C_2 should be also capable to do the same interaction a and to reach an equivalent state of s_2 even though after performing an unobservable activity. In this case, the intermediate states $\{q_1', q_1''\}$ in C_2 should be strongly equivalent to the starting point s_1 of C_1 by offering the same actions as s_1. Indeed, unobservable events in C_2 do not affect its

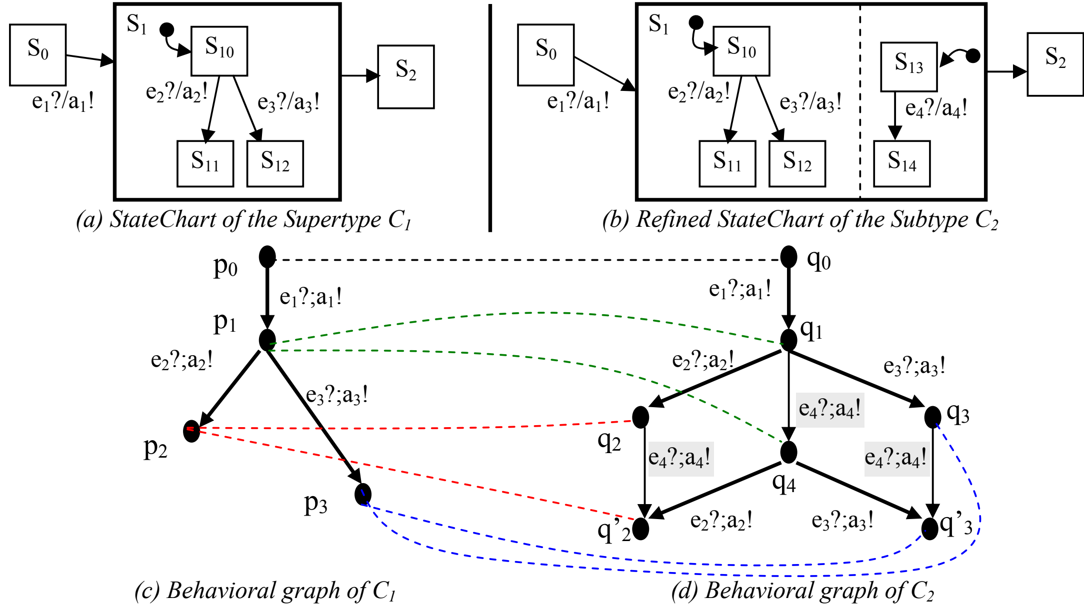


Figure 6.3: State machine refinement using strong subtyping relation.

capability at state q_1' or $q_1^{"}$ to simulate the oldest component C_1 which remains in its starting state s_1. So, the intermediate states (namely, q_1, q_1' and $q_1^{"}$) should preserve directly all services offered or required at the state s_1.

Besides the above strong bisimulation, the weaker version of our strong substitutability relation is the following one which agrees with the safe subtyping relations proposed in [57, 133].

Definition 31 $C_2 \preceq_{br} C_1$ *if and only if* $H_N(\llbracket C_2 \rrbracket) \approx_{br} \llbracket C_1 \rrbracket$.

Hence, C_2 behaves as C_1 in respect of C_1 clients with intermittent loss of service availability at some intermediate states (see Fig.6.2(a)).

Every time C_1 is able from a state s_1 to interact with its environment (by offering or demanding a service a) and reach some stable state s_2, C_2 is also capable to do the same interaction a and to reach an equivalent state of s_2 even though after performing unobservable activities. In this case, the intermediate states in C_2 are equivalent to the starting point s_1 of C_1 by offering the same actions but interleaved with τ-actions. As these states are stable, this means that C_2 may refuse at these points to provide old services to a client which still requires them.

Example: In order to illustrate the differences between the two aforementioned subtyping relations, a simple embedded system will now be considered. A first version of this system may consist of a component **T** (Temperature Controller) that takes readings from a set of thermocouples (via an analogue-to-digital converter, ADC) and

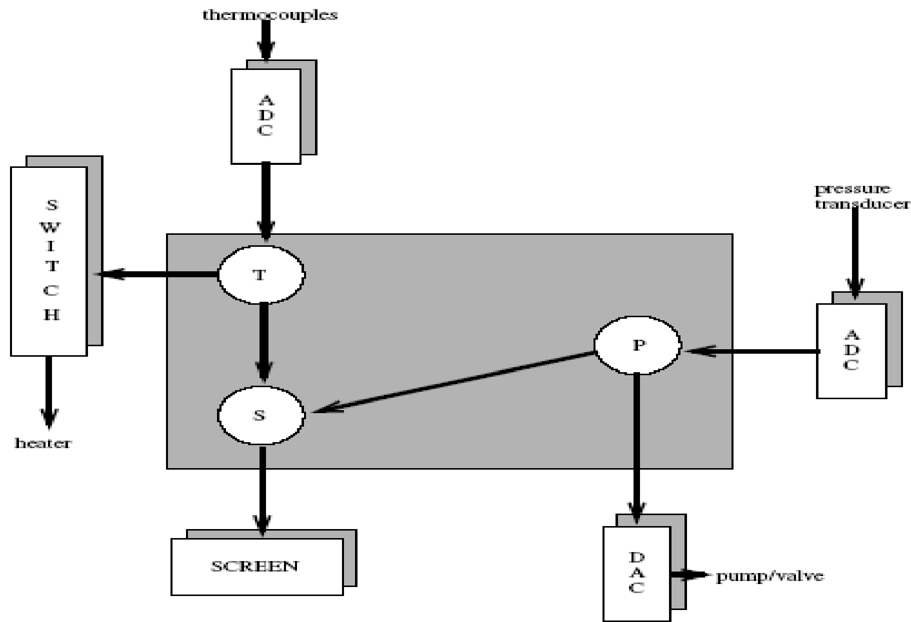


Figure 6.4: Simple embedded system.

converts them into temperature setting to make appropriate changes to a heater (via a digitally controlled switch). Thereafter, the data is communicated to the screen of the operator. The StateChart given in Figure 6.5(a) illustrates how such a system interacts with its environment and Figure 6.5(b) gives its behavioral model as a transition system.

We would later enhance the previous component as outlined in Figure 6.4, with an additional process **P** (Pressure Controller) for handling pressure measurement from pressure transducer and converting them into pressure setting to be send to some pump/valve device (via a digital to analogue converter, DAC). Both **T** and **P** must communicate data to **S**, which presents measurements to an operator via a screen. Note that **P** and **T** are active entities and the overall objective of this embedded system is to keep the temperature and pressure of some chemical process within defined limits.

A real system of this type would clearly be more complex allowing, for example, the operator to change the limits. However, even for this simple system, its implementation could take one of two forms:

(1) We may use an alternating single program which ignores the logical concurrency of **T** and **P** (see Fig.6.5(c)). Its related transition system, is illustrated by Fig.6.5(d). Note that this one with new actions $\{d, e, f\}$ hidden, is only weakly branching bisimilar to the transition system of the initial component depicted by Fig.6.5(b). Hence, the

implementation model has the immediate handicap that temperature and pressure readings can be taken only in alternating way, which may not be in accordance with requirements. It may still be necessary to split the computationally intensive sections into a number of distinct actions, and interleave these actions so as to meet a required balance of work. Even if this was done, there remains a serious drawback with this program structure: while waiting to read a temperature, no attention can be given to pressure (and vice versa). Moreover, if there is a system failure that results in, say, control never returning from the temperature *Read*, then in addition to this problem no further pressure *Reads* would be taken.

(2) A second solution, illustrated by Fig.6.5(e), consists in letting processes Temp Controller and Pressure Controller execute concurrently and each contains an indefinite loop within which the control cycle is defined. While one orthogonal region is suspended waiting for a read, the other may be executing. As a result, the transition system of the enhanced component (given in Figure 6.5(f)) is strongly branching bisimilar to the transition system of the old component of Figure 6.5(b) when the new actions $\{d, e, f\}$ are concealed; Even though any action of the process **P** is forwardly executed, process **T** remains able to initiate its loop and achieve its actions.

This kind of bisimulation is naturally based on an interleaving semantics and hence does not take into consideration the true concurrency between the two processes **T** and **P**. For instance, with this bisimulation we are not able to state that if there is a system failure on a temperature *Read*, pressure *Reads* could concurrently be taken. That is why we introduce later another kind of bisimulation supporting true concurrency (as explained in Section 6.6).

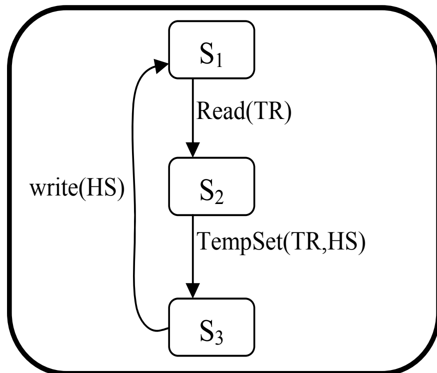
6.4.2 Case 2: The new component adds new provided/required services with removing some previous required services

Herein, we distinguish in Σ(C) two subsets:

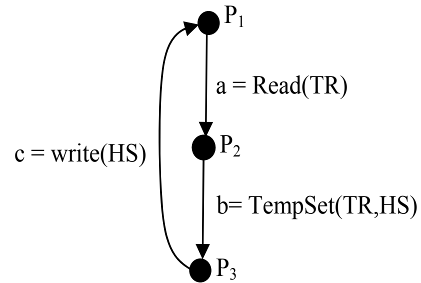
- Σ_P(C) denotes the subset of events related to the provided services of C.
- Σ_R(C) denotes the subset of events related to the required services of C.

Let C_2 be the new component added instead of C_1. Let α_P , α_R^1 , α_R^2 denote the following subsets:

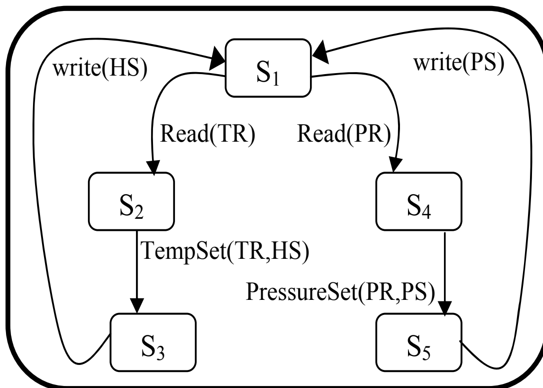
- $\alpha_P = \Sigma_P(C_2) \setminus \Sigma_P(C_1)$, including new provided services.
- $\alpha_R^1 = \Sigma_R(C_2) \setminus \Sigma_R(C_1)$, including required services added in the new component.



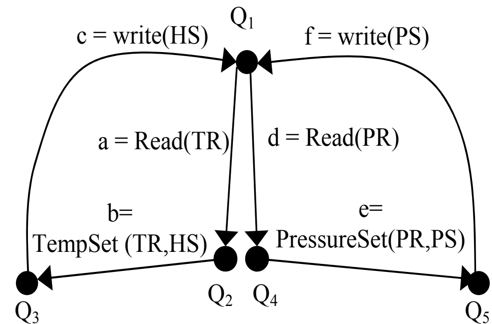
(a) The StateChart of a simple embedded system



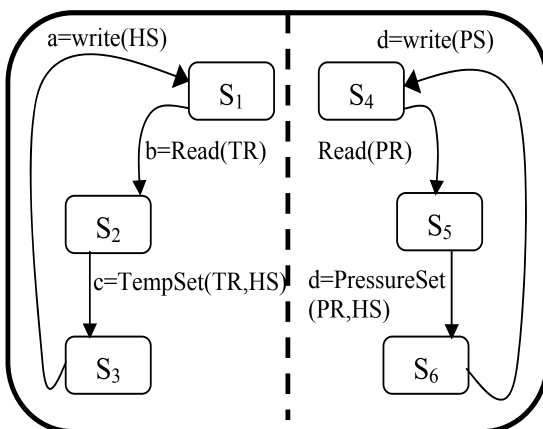
(b) The related Behavioural Model of (a)



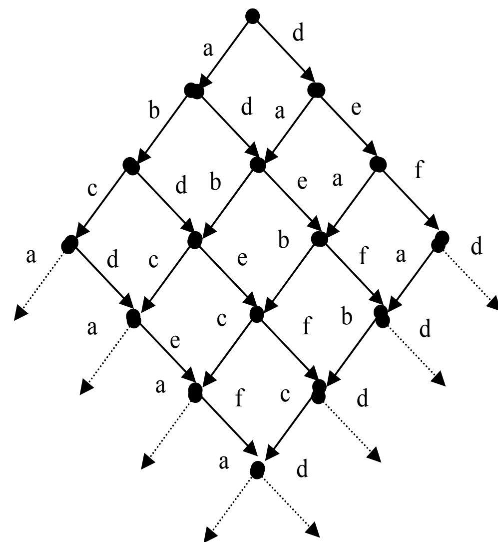
(c) The StateChart of the new embedded system



(d) The related Behavioural Model of (c)



(e) Another StateChart of the new embedded system



(f) The Related Behavioural Model of (e)

Figure 6.5: Behavioral models for the simple embedded system.

- $\alpha_R^2 = \Sigma_R(C_1) \setminus \Sigma_R(C_2)$, including required services removed from the new component interface [1].

Firstly, we propose to give the weak definition of substitutability in this case:

Definition 32 *$C_2 \preceq_{ibr} C_1$ if and only if $H_{\alpha_R^2}([\![C_1]\!]) \approx_{br} H_{\alpha_P \cup \alpha_R^1}([\![C_2]\!])$.*

In the first component C_1 model, we hide the required services removed from C_2. However in the second component model, we conceal the new provided and required services. The hiding operation in the two components is the same one, generating thus only τ-actions as invisible actions. However, the hiding actually should not be the same in the two components because we are interested in the availability of old services only in the new component. In fact, since removed services from the old component C_1 are required services which are no more used by its clients, we can deduce that these actions are under the exclusive control of C_1 when considering the new framework. Therefore, these removed actions have to be hidden into internal actions (i-action). Whereas in C_2, hidden services are only concealed to clients of the old component and remain needed by new component clients which cooperate to carry out them. For this purpose, we define a new branching bisimulation which behaves strongly with respect to τ-actions but behaves as a classical branching bisimulation with respect to i-actions.

Definition 33 *A strong i-branching bisimulation is a binary and symmetric relation $\mathcal{R} \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2$, $p\mathcal{R}q$ iff:*
$\forall a \in \Sigma \cup \{\tau, i\}\, \forall p' \in Q_1 : p \xrightarrow{a} p'$ then either

- *$a = \tau$ or $a = i$ and $p'\mathcal{R}q$,*
- *or $a = \tau$ and $\exists q' \in Q_2 : q \stackrel{i}{\Longrightarrow} q' \wedge p'\mathcal{R}q'$,*
- *or either at least one of the following cases holds:*
  - *$\exists q', q_1 \in Q_2 : q \stackrel{i}{\Longrightarrow} q_1 \xrightarrow{a} q'$ and $p\mathcal{R}q_1$ and $p'\mathcal{R}q'$,*
  - *$\exists q' \in Q_2 : q \xrightarrow{a} q'$ and $p'\mathcal{R}q'$.*

Two graphs G_1 and G_2 are strongly i-branching bisimilar (written $G_1 \approx_{sibr} G_2$) when their initial nodes are related by some strong i-branching bisimulation $\mathcal{R}$.

Two states are equivalent if they lead to new equivalent states via a same action even if interleaved with i-actions, but when one component achieves a τ-step then its source and target states should be able to do the same actions as the equivalent state on the other component.

Hence, we give the new strong definition of substitutability in this second case:

[1] Removed actions may be some bugs or deprecated operations.

Definition 34 $C_2 \preceq_{sibr} C_1$ *if and only if* $H'_{\alpha_R^2}([\![C_1]\!]) \approx_{sibr} H_{\alpha_P \cup \alpha_R^1}([\![C_2]\!])$.

Note that $H'_{\alpha_R^2}([\![C_1]\!])$ renames into i-action all actions of α_R^2 in C_1 whereas $H_{\alpha_P \cup \alpha_R^1}([\![C_2]\!])$ renames into τ-action all actions of $(\alpha_P \cup \alpha_R^1)$ in C_2.

Recall that we use two symbols τ and i to denote two kinds of unobservable actions in order to be able to differently handle the classes of actions they represent. $\tau-$actions result from hiding (by H) the new services in the updated component within all cases whereas $i-$actions are related to only the definition of $\approx_{sibr}$ and result from hiding (by H') some services of the old component removed from its subtype. The idea behind using these two symbols for depicting silent steps is the following:

The updated component is compared with its supertype by observing its behavior with relation to the context where the old component was used. Thus, only interactions of the subtype with clients of the old component are considered even if it interacts at the same time with new clients claiming new services. This is why, new actions are abstracted away into $\tau-$actions which are not under control of the testing environment; that is, clients of old component do not contribute in performing these actions which may occur independently from them. Therefore, we compel new actions to do not interfere with old actions in any way.

On the other hand, when some old services are removed from the new component these are relabeled into i-actions in the old component so that they become distinguishable from the new actions relabeled into $\tau-$actions in the new component. If we compare or test the two components, the $i-$actions would be under the control of the testing environment because this consists of the old component clients which know about and could invoke these actions in C_1 whereas C_2 would be unaware of their performance since these $i-$actions lead to equivalent states offering the same visible actions at both the two sides. Therefore, our observational equivalence for comparing old with updated components strengthens the conditions upon $\tau-$actions and weakens those upon $i-$actions since removed actions would not be invoked on the new component. In other terms, we use the criterion of a branching bisimulation for $i-$actions while we use a stronger criterion for $\tau-$actions.

Example: in Figure 6.6(a), we hide the actions that do not exist both in the two graphs. The obtained graphs G_1 and G_2 are branching bisimilar but are not strongly branching bisimilar. To make the two graphs strongly branching bisimilar, the node q_2 in G'_2 should preserve the same capabilities (dashed lines) as the equivalent state p_0 in G_1 (see Fig.6.6(b)).

Remark: Decidability and Complexity of Substitutability Checking

The decidability and complexity analysis of substitutability checking straightforwardly goes back to that of the underlying branching bisimulations which decidability and

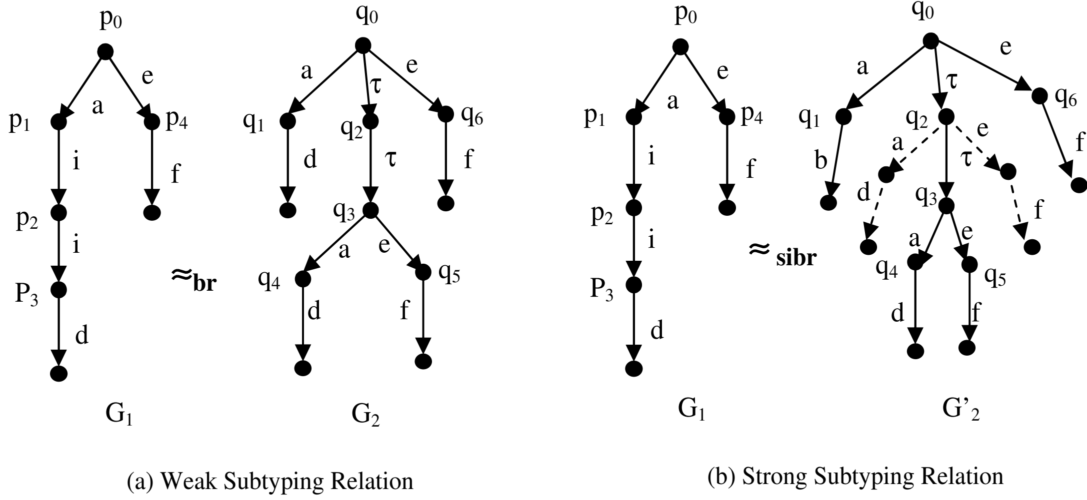


Figure 6.6: Strong and weak subtyping relations with considering of i-actions.

complexity issues have been analyzed by various characterization algorithms. All of them are based on the algorithm of Groote and Vaandrager [45] which solves the Relation Coarsest Partition with Stuttering problem (RCPS) for deciding branching bisimulation equivalence on finite labeled transition systems in $O(n(m + n))$ time and using $O(m + n)$ space, where n is the number of states, and m is the number of transitions.

6.5 Compatibility checking

A formal description of any system is given both in terms of detailed descriptions of its components and assumptions about their intended interactions. We would then perform adequacy checking between components behaviors as described in implementation models and interaction assumptions given as a specification model.

For this purpose, we adapt to our component-based framework a verification approach given in [73] where properties are seen as processes which are combined with the system model by a classical parallel operator. Then, the result has to remain equivalent to the system. However, our approach is more close to the refinement orientation and then takes a component specification as a global model pattern which any implementation should always match. This approach seems less complex to deal with than seeing the specification as some individual fragments of the intended behavior which we want the implementation to embed.

Accordingly, we introduce a definition of a more fitting synchronization product

between graphs[2] which can progress in an interleaving manner for the performance of the actions of some subset Γ but should synchronize for the other actions ($\in \overline{\Gamma} = \Sigma \backslash \Gamma$). Let $G_i = \langle Q_i, \hookrightarrow_i, \Sigma_i, q_{0_i} \rangle_{|i=1,2}$ be two transition systems.

Definition 35 *Let Γ be a subset of Σ. A $\overline{\Gamma}$-synchronization product $\otimes_{\overline{\Gamma}}$ of two graphs G_1 and G_2 yields a new transition system $G = \langle Q, \hookrightarrow, \Sigma, q_0 \rangle$ such that:*

- $Q = Q_1 \times Q_2$, $q_0 = (q_{0_1}, q_{0_2})$, $\Sigma = \Sigma_1 \cup \Sigma_2$
- *and* $\hookrightarrow = \{(p_1, p_2) \xrightarrow{a} (q_1, q_2) \mid a \notin \Gamma \wedge (p_1 \xrightarrow{a} q_1) \in \hookrightarrow_1 \wedge (p_2 \xrightarrow{a} q_2) \in \hookrightarrow_2\}$
  $\cup \{(p_1, p_2) \xrightarrow{a} (q_1, q_2) \mid a \in \Gamma \wedge (p_1 \xrightarrow{a} q_1) \in \hookrightarrow_1 \vee (p_2 \xrightarrow{a} q_2) \in \hookrightarrow_2\}$

We can derive from $\overline{\Gamma}$-Synchronization product two operators as follows:

- A full synchronization product ($\otimes_f$) when $\Gamma = \emptyset$. This means that the two graphs have to synchronize for all actions.
- A $\overline{\{\tau\}}$-synchronization product ($\otimes_{\overline{\tau}}$) when $\Gamma = \{\tau\}$. Herein, we do not require the synchronization of the two graphs for the special action τ.

The next proposition states the compositionality of branching bisimulation variants w.r.t. the product operator. Thus, when plugging an updated component instead of the old one, the new composite system is equivalent to the previous one.

Proposition 36 $\forall C$, $[\![C_1]\!] \sim [\![C_2]\!] \implies [\![C]\!] \otimes_{\overline{\Gamma}} [\![C_1]\!] \sim [\![C]\!] \otimes_{\overline{\Gamma}} [\![C_2]\!]$ *for all equivalence relations* $\sim \in \{\approx, \approx_{sbr}, \approx_{br}\}$

Proof. We prove the proposition for the case of $\approx_{br}$ which is the weaker variant of the branching bisimulations. Nevertheless, the proposition can be proved similarly for the strongest ones by employing the same method.

We should prove that the relation $\mathcal{R} = \{((p,q),(p,r)) \in ([\![C]\!] \otimes_{\overline{\Gamma}} [\![C_1]\!]) \times ([\![C]\!] \otimes_{\overline{\Gamma}} [\![C_2]\!]) : q \approx_{br} r\}$ is a branching bisimulation. Assume: $(p,q) \xrightarrow{a} (p',q')$ with $(p,q)\mathcal{R}(p,r)$. Hence, $q \approx_{br} r$ by definition of $\mathcal{R}$. We distinguish three cases:

case 1: $a \notin \Gamma$. Then, $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$. Since $q \approx_{br} r$, we obtain: $\exists r_1, r' \in [\![C_2]\!] : r \xRightarrow{\tau} r_1 \xrightarrow{a} r'$ and $q \approx_{br} r_1$ and $q' \approx_{br} r'$. Hence, $(p,r) \xRightarrow{\tau} (p,r_1) \xrightarrow{a} (p,r')$ and $q \approx_{br} r_1$ and $q' \approx_{br} r'$. Therefore, $(p,q)\mathcal{R}(p,r_1)$ and $(p',q')\mathcal{R}(p',r')$.

case 2: $a \in \Gamma \setminus \{\tau\}$. Then, $p \xrightarrow{a} p'$ and $q = q'$ or $q \xrightarrow{a} q'$ and $p = p'$. When $q = q'$, we get $(p',q)\mathcal{R}(p',r)$ because $q \approx_{br} r$. When $q \xrightarrow{a} q'$ and $p = p'$, we obtain: $\exists r_1, r' \in [\![C_2]\!] : r \xRightarrow{\tau} r_1 \xrightarrow{a} r'$ and $q \approx_{br} r_1$ and $q' \approx_{br} r'$. Hence, $(p,r) \xRightarrow{\tau} (p,r_1) \xrightarrow{a} (p,r')$ such that $q \approx_{br} r_1$ and $q' \approx_{br} r'$. Therefore, $(p,q)\mathcal{R}(p,r_1)$ and $(p,q')\mathcal{R}(p,r')$.

[2] Similar to the parallel operator of CSP [67]

case 3: $a = \tau$. Then, $p \stackrel{\tau}{\Longrightarrow} p'$ and $q = q'$ or $q \stackrel{\tau}{\Longrightarrow} q'$ and $p = p'$. When $q = q'$, we get $(p', q)\mathcal{R}(p', r)$ since $q \approx_{br} r$. When $q \stackrel{\tau}{\Longrightarrow} q'$ and $p = p'$, we obtain: $q' \approx_{br} r$ (by definition of $\approx_{br}$). Hence, we deduce that: $(p, q')\mathcal{R}(p, r)$.

Symmetrically, for any transition $(p, r) \stackrel{a}{\longrightarrow} (p', r')$ with $(p, q)\mathcal{R}(p, r)$, we can prove the above three cases.

Therefore, we deduce that $\mathcal{R}$ is a branching bisimulation. ■

We say that a component C implementation *strongly fulfills* ($\models$) its specification (S_{pec}) if the full synchronization product of their behavioral models (resp. $[\![C]\!], [\![S_{pec}]\!]$) is strongly bisimilar to the implementation model ($[\![C]\!]$).

Definition 37 $C \models S_{pec}$ *iff* $\Sigma(C) = \Sigma(S_{pec}) \wedge [\![S_{pec}]\!] \otimes_f [\![C]\!] \approx [\![C]\!]$.

We say also that a component C implementation *fulfills* (resp. weakly fulfills) its specification (S_{pec}) if the $\overline{\{\tau\}}$-synchronization product of their behavioral models is strongly branching bisimilar (resp. simply branching bisimilar) to the C model.

Definition 38 $C \models_{sbr} S_{pec}$ *iff* $\Sigma(C) = \Sigma(S_{pec}) \cup \{\tau\} \wedge [\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!] \approx_{sbr} [\![C]\!]$.

Definition 39 $C \models_{br} S_{pec}$ *iff* $\Sigma(C) = \Sigma(S_{pec}) \cup \{\tau\} \wedge [\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!] \approx_{br} [\![C]\!]$.

The following proposition tells us that when the full synchronization product of two graphs is *strongly bisimilar* to the first graph then their $\overline{\{\tau\}}$-synchronization product is always *strongly branching* bisimilar to the first graph. Formally,

Proposition 40 $C \models S_{pec}$ *implies* $C \models_{sbr} S_{pec}$ *implies* $C \models_{br} S_{pec}$.

Proof. We have to prove that: $[\![S_{pec}]\!] \otimes_f [\![C]\!] \approx [\![C]\!]$ implies $[\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!] \approx_{sbr} [\![C]\!]$ implies $[\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!] \approx_{br} [\![C]\!]$.

First implication: The bisimulation $\approx$ can be given as the relation $\mathcal{R}_0 = \{((p, q), q) \in ([\![S_{pec}]\!] \otimes_f [\![C]\!]) \times [\![C]\!] : (p, q) \approx q\}$. We prove now that the relation $\mathcal{R}_1 = \{((p, q), q) \in ([\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!]) \times [\![C]\!] : ((p, q), q) \in \mathcal{R}_0\}$ is a strong branching bisimulation.

Assume that: $(p, q) \stackrel{a}{\longrightarrow} (p', q')$ with $(p, q)\mathcal{R}_1 q$. By definition of $\mathcal{R}_0$ and $\mathcal{R}_1$, we get: $(p, q) \in ([\![S_{pec}]\!] \otimes_f [\![C]\!])$ and $(p, q) \approx q$. Hence, $\forall a \in \Sigma \cup \{\tau\}$, $p \stackrel{a}{\longrightarrow} p'$ and $q \stackrel{a}{\longrightarrow} q'$. Since $\exists q' \in [\![C]\!] : q \stackrel{a}{\longrightarrow} q'$ and since $(p, q) \approx q$ we obtain $(p', q') \approx q'$. By Definition 35, $([\![S_{pec}]\!] \otimes_f [\![C]\!]) \subseteq ([\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!])$. So, $(p', q') \in ([\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!])$, and thus, we obtain $(p', q')\mathcal{R}_1 q'$.

Conversely, for any transition $q \stackrel{a}{\longrightarrow} q'$ with $(p, q)\mathcal{R}_1 q$, we can similarly deduce $\exists (p', q') \in ([\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!]) : (p, q) \stackrel{a}{\longrightarrow} (p', q')$ and $(p', q')\mathcal{R}_1 q'$.

Therefore, $\mathcal{R}_1$ is a strong branching bisimulation.

Second implication: The bisimulation $\approx_{sbr}$ can be given as the relation $\mathcal{R}_2 = \{((p,q),q) \in ([\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!]) \times [\![C]\!] : (p,q) \approx_{sbr} q\}$. We prove now that the relation $\mathcal{R}_2$ is also a branching bisimulation.

Assume that: $(p,q) \xrightarrow{a} (p',q')$ with $(p,q)\mathcal{R}_2 q$. Hence, $(p,q) \approx_{sbr} q$ by definition of $\mathcal{R}_2$. We explore the two cases related to the branching bisimulation:

case 1: $a \in \Sigma \setminus \{\tau\}$. Then, $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$. Since $(p,q) \approx_{sbr} q$, we obtain: $(p',q') \approx_{sbr} q'$. Hence, $(p',q')\mathcal{R}_2 q'$.

case 2: $a = \tau$. Following Definition 35, we get either $p \xrightarrow{\tau} p'$ or $q \xrightarrow{\tau} q'$. If $q = q'$ then we deduce from $(p,q) \approx_{sbr} q$ that $(p',q) \approx_{sbr} q$ and thus, $(p',q)\mathcal{R}_2 q$. On the other hand, if $p = p'$ then we deduce from $(p,q) \approx_{sbr} q$ that $(p,q') \approx_{sbr} q'$ and thus, $(p,q')\mathcal{R}_2 q'$.

Conversely, for any transition $q \xrightarrow{a} q'$ with $(p,q)\mathcal{R}_2 q$, we prove the two cases related to the branching bisimulation:

case 1: $a \in \Sigma\setminus\{\tau\}$. Since $(p,q) \approx_{sbr} q$, then, $\exists (p',q') \in ([\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C]\!]) : (p,q) \xrightarrow{a} (p',q')$ such that $(p',q') \approx_{sbr} q'$. Thus $(p',q')\mathcal{R}_2 q'$.

case 2: $a = \tau$. Following Definition 35, we get $(p,q) \xrightarrow{\tau} (p,q')$. Since $(p,q) \approx_{sbr} q$ then $(p,q') \approx_{sbr} q'$. Thus, $(p,q')\mathcal{R}_2 q'$.

Therefore, we deduce that $\mathcal{R}_2$ is a branching bisimulation. ■

Below we present some results about compatibility checking between old and new components w.r.t. interaction assumptions with their environment.

Case 1: without removing any old required services.

When C_2 is a strong subtype of C_1 then whenever C_1 strongly fulfills any specification then C_2 also does. However when C_2 is a weak subtype of C_1 then the fulfillment of the specification by C_2 is weaken.

Theorem 41 *If $C_2 \preceq_{br} C_1$ then $C_1 \models S_{pec} \implies H_N(C_2) \models_{br} S_{pec}$.*

Proof. Following Definition 37, $C_1 \models S_{pec}$ implies $[\![S_{pec}]\!] \otimes_f [\![C_1]\!] \approx [\![C_1]\!]$. With regards to Definition 39, we have to prove that: $[\![S_{pec}]\!] \otimes_\tau H_N([\![C_2]\!]) \approx_{br} H_N([\![C_2]\!])$. Following Proposition 40, we obtain: $[\![S_{pec}]\!] \otimes_f [\![C_1]\!] \approx [\![C_1]\!]$ implies $[\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C_1]\!] \approx_{sb} [\![C_1]\!]$. However, $C_2 \preceq_{br} C_1$. This means that $H_N([\![C_2]\!]) \approx_{br} [\![C_1]\!]$. Since $H_N([\![C_2]\!]) \approx_{br} [\![C_1]\!]$, we have also: $[\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C_1]\!] \approx_{br} [\![C_1]\!] \approx_{br} H_N([\![C_2]\!])$. Therefore, to prove $[\![S_{pec}]\!] \otimes_\tau H_N([\![C_2]\!]) \approx_{br} H_N([\![C_2]\!])$, it suffices to prove : $[\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C_1]\!] \approx_{br} [\![S_{pec}]\!] \otimes_\tau H_N([\![C_2]\!])$. This property is straightforwardly deduced from Proposition 36 stating the compositionality of $\approx_{br}$. ■

Theorem 42 *If $C_2 \preceq_{sbr} C_1$ then $C_1 \models S_{pec} \implies H_N(C_2) \models S_{pec}$.*

Proof. Following Definition 37, $C_1 \vDash S_{pec}$ implies $[\![S_{pec}]\!] \otimes_f [\![C_1]\!] \approx [\![C_1]\!]$. With regards to Definition 38, we have to prove that: $[\![S_{pec}]\!] \otimes_{\overline{\tau}} H_N([\![C_2]\!]) \approx_{sbr} H_N([\![C_2]\!])$. Following Proposition 40, we obtain: $[\![S_{pec}]\!] \otimes_f [\![C_1]\!] \approx [\![C_1]\!]$ implies $[\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C_1]\!] \approx_{sbr} [\![C_1]\!]$. However, $C_2 \preceq_{sbr} C_1$ which means: $H_N([\![C_2]\!]) \approx_{sbr} [\![C_1]\!]$. Since $H_N([\![C_2]\!]) \approx_{sbr} [\![C_1]\!]$, we have also: $[\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C_1]\!] \approx_{sbr} [\![C_1]\!] \approx_{sbr} H_N([\![C_2]\!])$. Therefore, to prove $[\![S_{pec}]\!] \otimes_\tau H_N([\![C_2]\!]) \approx_{sbr} H_N([\![C_2]\!])$, it suffices to prove : $[\![S_{pec}]\!] \otimes_{\overline{\tau}} [\![C_1]\!] \approx_{sbr} [\![S_{pec}]\!] \otimes_\tau H_N([\![C_2]\!])$. This property is straightforwardly deduced from Proposition 36 stating the compositionality of $\approx_{sbr}$. ■

Case 2: with removing of some old required services.

Theorem 43 *If* $C_2 \preceq_{ibr} C_1$ *then* $H_{\alpha_R^2}([\![C_1]\!]) \vDash_{sbr} S_{pec} \implies H_{\alpha_P \cup \alpha_R^1}([\![C_2]\!]) \vDash_{br} S_{pec}$.

Theorem 44 *If* $C_2 \preceq_{sibr} C_1$ *then* $H'_{\alpha_R^2}([\![C_1]\!]) \vDash_{sbr} S_{pec} \implies H_{\alpha_P \cup \alpha_R^1}([\![C_2]\!]) \vDash_{sbr} S_{pec}$.

The same remarks hold in this case but here we require the fulfillment of S_{pec} by $H'_{\alpha_R^2}([\![C_1]\!])$ and not C_1 because some of required services will no more be used in C_2. The proofs are similar to those of theorems 41 and 42.

6.6 Substitutability relations in true concurrency context

When we handle the substitutivity of components with regards to true concurrency semantics, we should use a behavioral model that is suitable for this purpose [50].

6.6.1 Preliminary concepts

On comparing the two StateCharts C_1 and C_2 of Figure 6.7 with regards to an interleaving semantics, we can say they are equivalent even modulo the strong bisimulation. However, let us consider the case when a is a synchronous action. Thereafter, if the first StateChart is blocked on execution of a because of an unprepared cooperating component then it remains able to execute b once it receives its trigger event.

Indeed, according to the semantics of StateCharts [103] the actions a and b are independent and thus achieved concurrently, each of which within its orthogonal region. However, within the second StateChart, these two actions are causally ordered. Consequently, in the first case, the component C_1 is capable to react to the two stimuli at the same time whereas the second one can only handle them sequentially so that if it is blocked on a first synchronous action, it cannot immediately handle the second stimulus even though this is critical.

As any substitutability relation within true concurrency semantics shall be an non-interleaving model along with adequate observational comparison criteria, we naturally

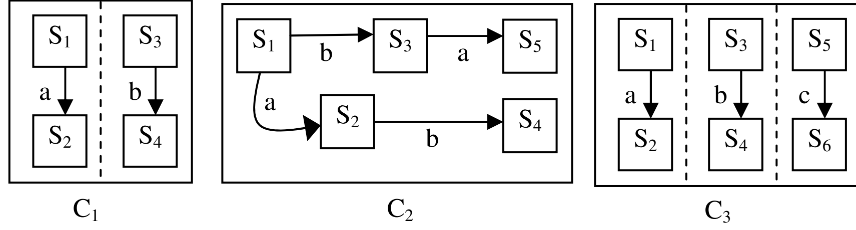


Figure 6.7: Interleaving Vs. true concurrency.

make use of asynchronous transition systems (yet given in Definition 1 of Chapter 3) underlying behavioral model.

The other remaining issue is about the observation criterion we could use to distinguish between the two aforementioned graphs. As the classical bisimulations are not suitable, we recall the forth-back bisimulation and then introduce a new variant thereof which is more appropriate for our setting.

Definition 45 *A forth-back bisimulation [31] is a symmetric relation $\approx_{fb}$ such that $p \approx_{fb} q$ if and only if $\forall p', a : p \xrightarrow{a} p'$ implies $\exists q', a : q \xrightarrow{a} q'$ and $p' \approx_{fb} q'$, and $\forall p', a : p' \xrightarrow{a} p$ implies $\exists q', a : q' \xrightarrow{a} q$ and $p' \approx_{fb} q'$,*

In order to take into account the unobservable actions, we give our new branching variant of the forth-back bisimulation.

Definition 46 *A branching forth-back bisimulation [50] is a symmetric relation $\mathcal{R} \subseteq Q_1 \times Q_2$, such that for every $(p, q) \in Q_1 \times Q_2, p\mathcal{R}q$ iff:*

- *$\forall a \in \Sigma \cup \{\tau\}, \forall p' : p \xrightarrow{a} p'$ then either $a = \tau$ and $p'\mathcal{R}q$ or $\exists q' : q \overset{\tau}{\Longrightarrow} q_1 \xrightarrow{a} q'$ and $p\mathcal{R}q_1$ and $p'\mathcal{R}q'$,*
- *and $\forall a \in \Sigma \cup \{\tau\}, \forall p' : p' \xrightarrow{a} p$ then either $a = \tau$ and $p'\mathcal{R}q$ or $\exists q' : q' \overset{\tau}{\Longrightarrow} q_1 \xrightarrow{a} q$ and $p'\mathcal{R}q_1$ and $p'\mathcal{R}q'$.*

Two graphs G_1 and G_2 are branching forth-back bisimilar (written $G_1 \approx_{bfb} G_2$) when their initial nodes are related by some branching forth-back bisimilar $\mathcal{R}$.

6.6.2 Substitutability relations

We still use the hiding operator (H) to conceal events related to additional services in new components making them unobservable from outside. We recall always that new unobservable events are different from internal ones because while the latter events are completely under control of the component, the former events are under control of both

the involved component and its environment. Hence, new services are unobservable only by some clients who view the improved component as it was the old one and thus still use only its old services. So, we consider below only hidden actions which depict new services and ensure that they do not affect the availability of old services of the new component in a true concurrency framework.

Let $N = \Sigma(C_2) \setminus \Sigma(C_1)$ be the set of new services. $H_N([\![C]\!])$ denotes the behavior model of C where all events of N are relabeled to τ. Now, we give below an optimal substitutability relation which is faithful enough in terms of preservation of capabilities of active components in a true concurrency framework:

Definition 47 $C_2 \preceq_{bfb} C_1$ *iff* $H_N([\![C_2]\!]) \approx_{bfb} [\![C_1]\!]$.

The main target of this definition is to ensure C_2 to behave as C_1 with preservation of the *independence* relation between concurrent actions whichever the state that C_2 has reached. Every time C_1 is able from a state s_1 to interact with its environment (by offering or demanding a service a) and reach (or return back to) some stable state s_2, C_2 should be also capable to do the same interaction a and to reach (or return back to) an equivalent state of s_2 even though after performing an unobservable activity. In this case, the intermediate states in a τ-sequence in C_2 should be equivalent to the starting point s_1 of C_1 by preserving all actions offered at the state s_1. Indeed, unobservable events in C_2 have not to exclude any action that C_1 can perform from the equivalent state s_1. Note that independent actions in C_1 remain independent in C_2 so that their interleaved paths remain illustrated by diamond-shaped forms even if they are combined with τ-actions (see Fig.6.8).

Example: Let C_3 be a new component we want to substitute to C_1 (see Fig.6.7). The new action c is hidden in the graph ($[\![C_3]\!]$) which we can easily prove that it is branching forth-back bisimilar to the behavioral graph of C_1 (see Fig.6.8).

However, these two graphs ($[\![C_1]\!]$ and $[\![C_3]\!]$) are not branching forth-back bisimilar to the graph obtained from the parallel combination of C_2 and the action c (see Fig.6.8) because C_2 is already not equivalent to C_1 via $\approx_{bfb}$.

6.7 Conclusion

This chapter has presented adequate substitutability relations that are based on strong assumptions of service availability and correctness properties mainly in reactive systems. We have dealt with this issue with respect to the two semantics; interleaving and true concurrency semantics. In all cases, the removed services in new components are concealed. However, our subtyping relations preserve the availability of old services even though the control is in some intermediate states along a chain of τ-steps.

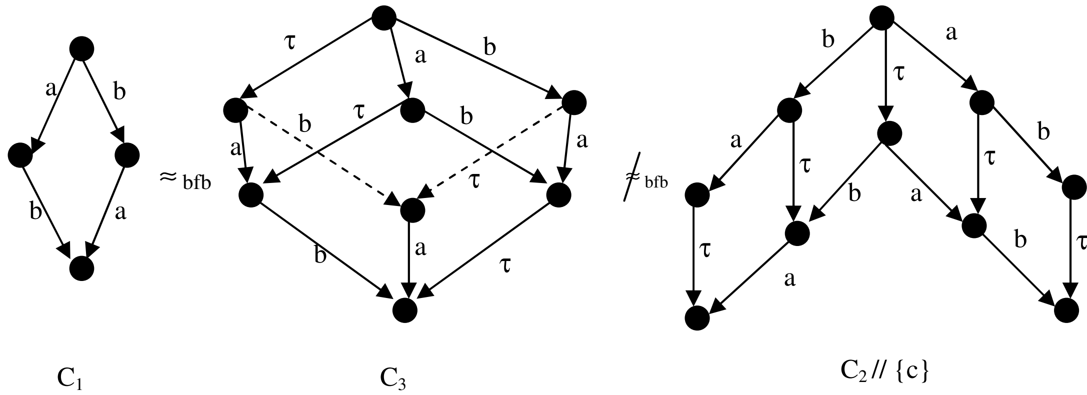


Figure 6.8: Branching forth-back bisimulation.

Furthermore, we have shown that new components continue to preserve properties that old components fulfill when interacting with its environment. In the true concurrency framework, we exploited the history preserving relations to strengthen the service availability in one component even though some of its orthogonal regions (internal tasks) are blocked at doing some synchronous actions.

Chapter 7

Conclusions

This chapter briefly summarizes some important results of this thesis, and gives an outlook on some potential future research directions.

After summarizing the results of our work, we discuss in this section some key topics enabled by our approach which aims at addressing issues of consistency checking, and substitutability and refinement of UML state and sequence diagrams within timed and non-interleaving semantics. We discuss as well, the issue of component composability and compatibility (yet sketched in ours papers [53, 55]).

7.1 Summary

The results in this thesis can be summarized as follows:

A In Chapter 2, we have proposed a concurrent and an object-oriented paradigm promoting a component-based design of reactive systems to manage their complexity and time constraints. Indeed, passive objects are grouped into components, each of which possesses an active object controlling its passive objects. When the active object receives required services from other components (namely, their enclosed active objects) it then invokes related methods of passive objects enclosed in its component. Consequently, the system under design is seen as a collection of active components'state machines evolving in a concurrent way. Any one of them has its own event pool and progresses according to its individual pace of run-to-completion steps. However, from time to time one may require or provide services to the others by means of event instances which are conveyed and queued in event pools of target components (independently of the statuses these components are in). We have also presented, in Chapter 3, a new approach towards the comprehensive formalization of UML state machines for analysis and verification purposes at early stages of software development. The implementation model is

a rigorous formalism, namely modular time Petri nets. We have enhanced that formalism with some suitable concepts (such as use of a modular style, thread places, event flow and control flow places, decomposition of some actions into instantaneous actions, etc.) so that our method succeeds in dealing with both all kinds of composite states and pseudo-states and allows us capturing the temporal annotations on state machine arcs. Once a state diagram has been converted into inter-linked Petri net modules related to state machines of involved components, we could gradually generate a whole reachability graph from those of components. Note that state space exploration can be tailored with respect to any intended run-to-completion policy. Time annotations are also mapped into time constraints to put onto nodes and arcs of produced graphs.

B As inter-objects interactions are given in terms of UML 2 sequence diagrams, we have provided them, in Chapter 4, with a formal semantics by using a faithfully branching time structures rather than the classical trace-based ones. This model records both traces of all interaction components together with branching bifurcations and it can be directly unfolded into a transition system capturing the intended behavior. The graphs related to interaction fragments are equipped with few generalized algebraic operations which help us define the formal semantics of all interaction operations in compositional manner. Moreover, we have proposed a method to extract time properties of UML interactions into time constraints we add to our graph in order to enable timeliness and performance analysis. Hence, resulting graphs modeling valid and invalid behaviors could be now compared to the state diagram to achieve semantically and temporal consistencies checking.

C As the dynamic view of objects and components are given in terms of two kinds of diagrams illustrating respectively their internal behaviors and interactions, we have presented in Chapter 5 a formal method for their compatibility and consistency checking. We have given an exhaustive method that takes care of the features of UML 2.x state and sequence diagrams, particularly the forbidden sequences introduced by the interaction operators *neg* and *assert*. Afterwards, we have tailored this approach to the component-based framework. We have presented a new method for extracting partial state spaces of a system with regard to its interacting components, one at a time, in such a way that each partial behavioral graph contains, in addition to the component actions, those which are causally related to them in order to cope with the asynchronous communication between communicating state machines. We have made use of these graphs to analyze the compatibility and composability between components. Also, we

proposed a fitting comparison way, based on branching simulation, to modularly check whether components state machines fulfill their interaction-based specification. We have also shown that this relation is preserved by our composition operation. In so doing, we can assert that partial checking results could be generalized onto the combined systems.

D Even though inheritance and refinement have proven potential benefits in terms of implementation reuse, the refinement process of objects'state machines may raise two kinds of problems in the new behavior: unavailability of previously provided services and violation of global correctness properties that were previously respected. In order to make this process less hazardous for reactive systems, we have presented in Chapter 6 adequate substitutability relations that are based on strong assumptions of service availability and correctness properties. We have dealt with this issue with respect to the two semantics; interleaving and true concurrency semantics. In all cases, the removed services in new components are concealed. However, our subtyping relations preserve the availability of old services even though the control is in some intermediate states along a chain of τ-steps. Furthermore, we have shown that new components continue to preserve properties that old components fulfill when interacting with their environment. In the true concurrency framework, we exploited the history preserving relations to strengthen the service availability in one component even though some of its orthogonal regions (internal tasks) are blocked at doing some synchronous actions.

7.2 Future research directions

The formalization process of the UML dynamic diagrams as presented in this thesis raises several fundamental questions as well as practical questions that concern, in particular, the applicability of the proposed approach. In the following, selected issues of both are highlighted, and potential for further research discussed.

7.2.1 Consistency checking of dynamic diagrams

The key issue to support the design of reliable software systems is to prove that (scenario-based) requirements depicted as sequence diagrams are faithfully implemented into state machines which correctly realize the intended interactions. Our approach seems practical to modularly and incrementally uncover design flaws of UML dynamic diagrams and to check component substitutability. However, for dealing with real-time systems, the approach need to be extended to find out whether and how tim-

ing constraints of sequence diagrams could be fulfilled in respect of time tags of state diagrams. Further work might, as well, be done with regard to non-interleaving semantics and non-atomic actions.

Consistency checking within non-interleaving semantics

We deem that it is important to make use of our methods which yield asynchronous transition systems to achieve the semantics consistency checking of the dynamic diagrams of any system under design within the true concurrency semantics and situations where actions may take a non-zero duration. The analysis of sequence diagrams remains also an important issue for other systems where threads may be transversal to participants (namely passive objects). For such cases, new methods need to be explored to compare sequence charts depicting only fragments of scenarios based specifications to each other and to timing and state diagrams.

Consistency checking of timed graphs

To allow using automated tools for time checking of UML dynamic diagrams, we shall make use of the underlying formalism in our methods defined in [47, 48] is transition systems augmented with logical clocks and time formulas over values of these clocks to express timing constraints of complex systems.

Timing annotations on the implementation graph are almost concerned with execution and transmission durations whereas the timing constraints in the specification graph describe the requirements in terms of deadlines to be met. When comparing the two graphs, we would verify that they are consistent with each other. That is, the implementation fulfills the specification in a way more suitable than what has been stated in the untimed case.

We could, similarly, check that the composition of timed components does not raise farther incompatibilities. Indeed, the timing constraints have an effect of the reachability of states and deadlines of certain actions may exclude the availability of others.

In this setting, many time consistency checking methods could be defined to implement various timing policies ranging over the spectrum between two extreme timing consistencies [52, 53, 54]: The strong timing consistency checking requires that the produced graph be strongly equivalent to the specification graph in such a way that any action enabled during an interval $[x, y]$ at one node s in the former graph should be also enabled at an equivalent node s' in the latter graph during the same interval $[x, y]$. On the other hand, a weak timing consistency checking turns to check only the reachability of all nodes (statuses) that were previously available in the untimed

specification graph even if the firing durations of actions are not closely preserved.

7.2.2 State machine refinement and substitutability

It should be noted that the substitutability of state machines is a key topic to the modeling and verification of reactive and real-time systems. The behavioral subtyping relations should take care of time aspects, non atomic actions, and true concurrency semantics of active components such that (reactivity) capabilities of any previous state machine should be always preserved in the refined ones. For future work, we deem that it is important to consider quantitative time constraints in substitutability relations and to analyze to what extent the different kinds of interactions may influence the compatibility of constrained components.

7.2.3 Compatibility and composability checking of components

Compatibility and composability checking of components for reactive systems is a critical issue to ensure that composition of components does not lead to risky deadlock and irregular situations. As explained in [53, 55], we advocate the use of automata-based models to depict both specification and implementation details of components where asynchronous as well as synchronous (inter)actions are considered (unlike approaches adopting only synchronous actions [27]). Their combination shall yield a more complete component behavior description where erroneous and correct final states would be emphasized. Composition of such models not only shows whether deadlock states may exist or not but depicts also risky computations leading to erroneous states that are related to forbidden statuses [53, 55]. Therefore, a large weak-strong spectrum of compatibilities could be scanned yielding various criteria for deadlock-free composition of components.

Furthermore, considering the issue of time compatibility, many timing policies could be explored ranging from loose to strong rules for the synchronization of components' time constraints. The definition of new flexible timing policies need to adequately and differently manage the various classes of actions according to their level of urgency and criticality in order to avoid time inconsistencies but without denying safety properties.

7.2.4 Runtime verification and model based testing

Any framework of runtime verification and model based testing of StateCharts would be a valuable means for the detection of failures as well as identification of their causes at early stages of the development process [81]. Moreover, such an approach would be considered efficient for the analysis of reactive systems since the checking process

would not be exhaustive, thereby helping overcome the state explosion problem. The use of the behavioral graphs with time information could be also used for runtime monitoring of systems to make them evolve in such a way that intended properties remain always fulfilled.

Bibliography

[1] R. Alur and D. Dill. *A theory of timed automata*, Theoretical Computer Science, Volume 126, pp. 183-235, Elsevier, 1994.

[2] G. Agha and C. Hewitt. *Concurrent programming using actors*, in A. Yonezawa and M. Tokoro, editors, Object Oriented Concurrent Programming. MIT Press, 1988.

[3] P. America. *Issues in the design of a parallel object oriented language*, Formal Aspects of Computing, 1(4):366-411, October 1989.

[4] P. America. *A parallel object oriented language with inheritance and subtyping*, in OOPSLA/ECOOP '90, pages 161-168, 1990.

[5] P. America. *Pool - design and experience*, OOPS Mesenger, 2(2), 1991.

[6] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. *Towards a theory of actor computation*, in The Third International Conference on Concurrency Theory (CONCUR '92), LNCS 630, pp. 565-579. Springer Verlag, Aug. 1992.

[7] P.C. Attie. *Synthesis of large concurrent programs via pairwise composition*, in Proc. of the 10th International Conference on Concurrency Theory (CONCUR'99), 1999.

[8] L. Baresi. *Some Preliminary Hints on Formalizing UML with Object Petri Nets*, in Proc. of Integrated Design and Process Technology (IDPT-2002), June 2002.

[9] O. Biberstein, D. Buchs, and N. Guelfi. *Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism*, in G. Agha, F. De Cindio and G. Rozenberg, editors, Advances in Petri Nets on Concurrent Object-Oriented Programming and Petri Nets, LNCS 2001, pp. 70-127, Springer-Verlag.

[10] M. Broy, M.L. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic. 2nd *UML 2 Semantics Symposium: Formal Semantics for UML*, MoDELS 2006 Workshops, LNCS 4364, pp. 318-323, Springer, 2007.

[11] C. Bui Thanh and H. Klaudel. *Object-Oriented Modelling with High-Level Modular Petri Nets*, In. Proc. of IFM 2004, E. Boiten, J. Derrick, G. Smith (Eds.), LNCS 2999, pp. 287-306, Springer, 2004.

[12] R. Bastide, and P. Palanque. *Cooperative Objects: a Concurrent, Petri-Net Based Object-Oriented Language*, In Proc. of Systems Engineering in the Service of Humans, IEEE-SMC'93, Le Touquet, France, October 15-20, 1993. IEEE Press (1993)

[13] L. Baresi and M. Pezzè. *Improving UML with Petri Nets*, Electronic Notes in Theoretical Computer Science, Volume 44, Number 4, Elsevier, 2001.

[14] M. Broy. *Formal Description Techniques - how formal and descriptive are they*, in Proc. of FORTE IX, pp. 95-112, Chapman & Hall, 1996.

[15] V. del Bianco, L. Lavazza and M. Mauri. *A Formalization of UML Statecharts for real-time software modeling*, in Proc. of Integrated Design and Process Technology (IDPT-2002).

[16] L. Cardelli. *A semantics of multiple inheritance*, Information and Computation, 76:138-164, 1988.

[17] L.A. Campbell, B.H.C. Cheng, W.E. McUmber, and R.E.K. Stirewalt, *Automatically Detecting and Visualing Errors in UML Diagrams*, Journal of Requirements Engineering, Springer, Volume 7, Number 4, pp.264-287, 2002.

[18] S. Chaki, E. Clarke, N. Sharygina, and N. Sinha. *Dynamic Component Substitutability Analysis*, in Proc. of Formal Methods Conference, 2005.

[19] M.L. Crane and J. Dingel. UML vs. *Classical vs. Rhapsody Statecharts: Not all models are created equal*, Software and Systems Modelling, Volume 6, Number 4, December 2007.

[20] M.L. Crane and J. Dingel. On the Semantics of UML State Machines: Categorization and Comparison. Technical Report 2005-501, Queen' s University, August 2005.

[21] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2011)*, SRI International, USA, January 2011.

[22] S. Christensen and L. Petrucci. *Modular analysis of Petri Nets*, The Computer Journal, Volume 43, Number 3, pp. 224-242, 2000.

[23] R.G. Clark and A.D.Moreira. *Use of E-LOTOS in adding Formality to UML*, Journal of Universal Computer Science, Volume 6, Number 3, pp. 1071-1087, 2000.

[24] J. Cardoso, C. Sibertin-Blanc. *An operational semantics for UML interaction: sequencing of actions and local control*, European Journal of Automatised Systems. APII-JESA 36 P.1015-1028 (ISBN 2-7462-0573-4), Hermés-Lavoisier 2002.

[25] I. Černá, P. Vařeková, B. Zimmerová. *Component Substitutability via Equivalencies of Component-Interaction Automata*, *Proc.* $3^r d$ *International Workshop on Formal Aspects of Component Software (FACS'06)*, Czech Republic, September 20-22, 2006.

[26] L. de Alfaro, , T.A. Henzinger. *Interface Automata*, in Proc. the join 8th European Software Engineering and 9th ACM SIGSOFT Intl Symposium on the Foundations of Software Engineering (ESEC/FSE), 2001.

[27] L. de Alfaro, T.A. Henzinger, M. Stoelinga. *Timed Interfaces*, in Proc. 2nd Intl Workshop on Embedded Software (EMSOFT), LNCS 2491, Springer, pp. 108-122, 2002.

[28] W. Damm and D. Harel. *LSCs: Breathing Life into Message Sequence Charts*, in Proc. of FMOODS'1999.

[29] W. Damm, B. Josko, H. Hungar, and A. Pnueli. *A compositional real-time semantics of STATEMATE designs*, in Proc. of the International Symposium on Compositionality: The Significant Difference, LNCS 1536, pp. 186-238, Springer, September 1998.

[30] D. Drusinsky, J.B. Michael, and M. Shing. *The Three Dimensions of Formal Validation and Verification of Reactive System Behaviors*, Technical Report NPS-CS-07-008, Naval Postgraduate School, Montery, USA, August 2007.

[31] R. de Nicola, U. Montanari, F.W. Vaandrager. *Back and Forth Bisimulations*, Report CS-R9021, Centrum voor Wiskunde en Informatica (CWI), The Netherlands, May 1990.

[32] A. Evans, J.M. Bruel, R. France, and K. Lano. *Making UML Precise*, in Proc. of the OOPSLA'98 Workshop on Formalizing UML, 1998.

[33] G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. *Testing the consistency of dynamic UML diagrams*, in Proc. of the 6th International Conference on Integrated Design and Process Technology (IPDT), Pasadena, USA, 2002.

[34] A. Egyed. *Instant Consistency Checking for the UML*, in Proc. of ICSE06, May 2028, 2006, Shanghai, China.

[35] J. Elamkulam, Z. Glazberg, I. Rabinovitz, G. Kowlali, S.C. Gupta, S. Kohli, S. Dattathrani, and C. P. Macia. *Detecting Design Flaws in UML State Charts for Embedded Software*, in Proc. of HVC'2006, E. Bin, A. Ziv, and S. Ur (Eds.), LNCS 4383, pp. 109-121, 2007.

[36] R. France, A. Evans, K. Lano, and B. Rumpe. *The UML as a Formal Notation*, in Proc. of UML' 98 - Beyond Notation First International Workshop, Mulhouse, France, June 1998.

[37] C. Fischer and H. Wehrheim. *Behavioural Subtyping Relations for Object-Oriented Formalisms*, in Proc. of AMAST' 2000, LNCS 1816, Springer, 2000.

[38] J.L. Garrido and M.Gea. *A Coloured Petri Net Formalisation for a UML-based Notation Applied to Cooperative System Modelling*, in Proc. of DSV-IS 2002, LNCS 2545, pp. 16-28, Springer, 2002.

[39] D. M. Gabbay, C. J. Hogger, J. A. Robinson, and J. H. Siekmann (eds.). *Handbook of Logic in Artificial Intelligence and Logic Programming*, Volume 2, Deduction Methodologies. Oxford University Press, 1994.

[40] A. Gherbi and F. Khendek, *Consistency of UML/SPT Models*, in Proc. of SDL'2007, E. Gaudin, E. Najm, and R. Reed (Eds.), LNCS 4745, pp. 203-224, 2007.

[41] M. Gogolla and F.Parisi-Presicce. *State diagrams in UML: A formal semantics using graph transformations*, in Proc. of the workshop on Precise Semantics for Modeling Techniques (PSMT' 98).

[42] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, 1983.

[43] R. Grosu and S.A. Smolka, *Safety-Liveness Semantics for UML 2.0 sequence diagrams*, in Proc. of ACSD'05, the 5th International Conference on Application of Concurrency to System Design, Saint-Malo, France, June 2005.

[44] G. Gössler, J. Sifakis. *Composition for Component-Based Modeling*, Science of Computer Programming, 55 (1-3), pp. 161-183, March 2005.

[45] J.F. Groote, and F.W. Vaandrager. *An efficient algorithm for branching bisimulation and stuttering equivalence*, in Proc. of ICALP 90, Paterson MS (Ed), LNCS 443, pp. 626-638, Springer, 1990.

[46] Y. Hammal. *Separation of Concurrency Models Using Languages Embeddings*, in Proc. of the International Arab Conference on Information Technology (ACIT'2005), December 6-8, 2005, Al-Isra Private University, Jordan.

[47] Y. Hammal. *A formal Semantics of UML State Charts by means of Timed Petri nets*, in Proc. of 25th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2005), LNCS 3731, Taipei, Taiwan, 2005.

[48] Y. Hammal. *Branching Time Semantics for UML 2.0 Sequence Diagrams*, in Proc. of 26th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2006), LNCS 4229, Paris, France, 2006.

[49] Y. Hammal. *A Modular State Exploration and Compatibility Checking of UML Dynamic Diagrams*, in Proc. of the 6th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-08), Doha, Qatar, 2008.

[50] Y.Hammal. *Substitutability Relations for Active Components*, in Proc. 4th Workshop on Formal Aspects of Component Software (FACS07), Sophia-Antipolis, France, 19-21 September 2007.

[51] Y. Hammal. *A Component-Based Approach for Consistency Checking of UML Dynamic Diagrams*, in IASTED proceedings of the International Conference on Software Engineering and Applications (SEA 2007), Massachusetts, USA, 19-21 November 2007.

[52] Y. Hammal. *A formal Methodology for Semantics and Time Consistency Checking of UML Dynamic Diagrams*, in CCIS/LNCS (59-0078) Proceedings of the International Conference on Advanced Software Engineering & its Applications (ASEA'2009), Jeju, South Korea, Springer, December 10-12, 2009.

[53] Y. Hammal. *Behavioral Compatibility of Active Components*, in Proc. of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM-08), 10-14 November 2008, Cape Town, South Africa.

[54] Y. Hammal. *A Formal Methodology for Semantics and Time Consistency Checking Of UML Dynamic Diagrams*, Journal of the Chinese Institute of Engineers (JCIE), Volume 34, issue 2, March 2011, Taylor & Francis, UK.

[55] Y. Hammal. *Towards Checking Protocol Conformance of Active Components*, in International Journal of Software Engineering and Its Applications (IJSEIA), Volume 5, Number 2, SERSC, Korea, April 2011.

[56] A. Abdelli and Y. Hammal. *A TPN based framework for the specification of real time embedded systems*, in Proc. of the 4th IEEE International Conference on Embedded and Multimedia Computing (EM-Com'2009), Jeju, South Korea, December 10 - 12, 2009.

[57] N. Hameurlain. *On Compatibility and Behavioural Substitutability of Component Protocols*, in Proc. of the 3rd IEEE International Conference Software Engineering and Formal Methods (SEFM' 2005), Germany, September 5-9, 2005.

[58] D. Harel. *A visual formalism for complex systems*, Science of computer programming, Volume 8, Elsevier, 1987.

[59] S. Haddad, J-M Ilié and K. Klai. *An Incremental verification Technique using Decomposition of Petri Nets*, in Proc. of the IEEE Conference on Systems, Man and Cybernetics (SMC' 02), Hammamet, Tunisia, 2002.

[60] D. Harel and E. Gery. *Executable Object Modeling with StateCharts*, IEEE Computer, Volume30, Number 7, pp. 31-42, July 1997.

[61] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde and Ketil Stølen. *STAIRS towards formal design with sequence diagrams*, Software and System Modeling, volume 4, number 4, 2005, pp.355-367.

[62] D. Harel and H. Kugler. *The RHAPSODY Semantics of StateCharts (or, On the Executable Core of the UML)*, in Proc. of Integration of Software Specification Techniques for Application in Engineering, LNCS 3147, pp. 325-354, Springer, 2004.

[63] D. Harel and O. Kupferman. *On the Behavioral Inheritance of State-Based Objects*, in Proc. of 34th International Conference of Technology of Object-Oriented Languages and Systems (TOOLS34), pp. 83-94, 2000.

[64] D. Harel and S. Maoz. *Assert and negate revisited: Modal semantics for UML sequence diagrams*, Software and System Modeling, Springer, Volume 7, Number 2, pp.237-252, 2008.

[65] G. Huszerl, I. Majzik, A. Pataricza, K. Kosmidis, and M. Dal Cin. *Quantitative Analysis of UML Statechart Models of Dependable Systems*, The Computer Journal, Volume 45, Number 3, 2002.

[66] D. Harel and A. Naamad. *The STATEMATE Semantics of Statecharts*, ACM Transactions on Software Engineering and Methodology, Volume 05, Number 4, pp. 293-333, October 1996.

[67] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[68] G.J. Holzmann. *Software Model Checking*, NATO Summer School, Volume 180, pp. 309-355, IOS Press Computer and System Sciences, Marktoberdorf Germany, Aug. 2000.

[69] D. Harel, and A. Pnueli. *On the development of reactive systems*, in Logics and models of concurrent systems. Springer, New York, NY, USA, 477498, 1985.

[70] Øystein Haugen and Ketil Stølen. *STAIRS - Steps to Analyze Interactions with Refinement Semantics*, in Proc. of 6th International Conference on UML- The Unified Modeling Language, Modeling Languages and Applications, San Francisco, CA, USA, October 20-24, 2003, pp. 388-402.

[71] S. E. Keene. *Object Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley Publishing Co., 1989.

[72] I. Keidar, R. Khazan, N. Lynch, A. Shvartsman. *An Inheritance-Based Technique for building Simulation Proofs Incrementally*, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **11(1)**, 2002, pp. 1-29.

[73] J. Kelso and G. Milne. *Properties as Processes: their Specification and Verification*, in Proc. of the 25th International Conference on Formal Techniques for Networked and Distributed Systems, LNCS 3731, Springer, Taiwan, 2-5 Oct. 2005.

[74] Y. Kesten, and A. Pnueli. *Timed and Hybrid Statecharts and Their Textual Representation*, in Proc. of the 2nd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 571, pp. 591-620, Springer, 1992.

[75] K. Korenblat and C. Priami. *Extraction of PI-Calculus Specifications from UML Sequence and State Diagrams*, Technical Report # DIT-03-007, February 2003, University of Trento, Dept. of Information and Communication Technology, Italy.

[76] ITU-T. Z.120. *Message sequence charts (MSC)*, November 1999.

[77] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*, 3rd Ed. Addison-Wesley, 2005.

[78] C. B. Jones. *Process-algebraic foundations for an object-based design notation*, Technical Report UMCS-93-10-1, Manchester University, October 1993.

[79] P. King and R. Pooley. *Derivation of Petri Net Performance Models from UML specifications of Communications Software*, in Proc. of TOOLS 2000, B.R.Haverkort & al.(Eds.), LNCS 1786, pp. 262-276, Springer, 2000.

[80] C. Lakos. *Object-oriented Modelling with Object Petri Nets*, in Proc. of the International conference on Concurrent OOP and PN, G. Agha et al. (Eds.), LNCS 2001, pp. 1-37, Springer, 2001.

[81] L. Lúcio, D. Buchs, and L. Pedro. *Semi-Automatic Test Case Generation from CO-OPN Specifications*, in Proc. of the M-TOOS 2007 Workshop, Portland, Oregon, USA.

[82] G.T. Leavens, K.K. Dhara. *Concepts of Behavioral Subtyping and a Sketch of their Extension to Component-Based Systems*, *Foundations of Component-Based Systems*, (Gary T. Leavens and Murali Sitaraman (editors)), Cambridge University Press,Chapter 6, pages 113-135, 2000.

[83] E. Latronico and P. Koopman. *Representing Embeded System Sequence Diagrams as a Formal Language*, in Proc. of UML'2001 Conference, CityplaceToronto StateOntario, 3-5 Oct.2001.

[84] G. Lüttgen and M. Mendler. *What is in a step: a Fully-Abstract Semantics for Statecharts Macro Steps via Intuitionistic Kripke Models*, Technical report CS-00-04, Dept of Computer Science, University of Sheffield, UK.

[85] V.S.W. Lam and J. Padget, *Consistency checking of sequence diagrams and statechart diagrams using the π-Calculus*, in Proc. of International Conference on Integrated Formal Methods (IFM'2005), LNCS 3771, 2005.

[86] Mass. Soldal Lund and Ketil Stølen. *A Fully General Operational Semantics for UML 2.0 Sequence Diagrams with Potential and Mandatory Choice*, in Proc. of FM'2006, pp.380-395, 2006.

[87] B. Litvak, S. Tyszberowicz, and A. Yehudai. *Behavioral Consistency Validation of UML Diagrams*, in Proc. of the 1st International Conference on Software Engineering and Formal Methods, 2003.

[88] B.H. Liskov and J.M. Wing. *A Behavioral Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, Volume 16, Number 6, pp. 1811-1841, 1994.

[89] P. Merlin. *A study of the recoverability of computer systems*, PhD Thesis, Dept of computer science, University of California, Irvine, 1974.

[90] B. Meyer. *Eiffel: The Language*, Prentice-Hall, 1992.

[91] R. Milner. *Communication and Concurrency*, Prentice-Hall International, 1989.

[92] B. Mitchell. *Inherent Causal Orderings of Partial Order Scenarios*, in Proc. of International Colloquium on Theorical Aspects of Computing, place City Guiyang, country-region China, (LNCS 3407) PP 114-129. September 2004.

[93] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel. *On formal Semantics of StateCharts as Supported by STATEMATE*, in Proc. of the 2nd BCS-FACS Northern Formal Methods Workshop, July 1997.

[94] P-A Masson, H. Mountassir, and J. Julliand. *Modular Verification for a Class of PLTL Properties*, in Proc. of the International Conference of Integrated Formal Methods (IFM' 2000), LNCS 1945, pp. 398-419, Springer, Germany, 1-3 Nov. 2000.

[95] A. Maggiolo-Schettini, A. Peron, and S. Tini. *A comparison of Statecharts step semantics*, Theoretical Computer Science, Volume 290, Number 1, pp. 465-498, Elsevier, 2003.

[96] Zoltán Micskei and Hélène Waeselynck. *The many meanings of UML 2 Sequence Diagrams: a survey*, Software and System Modeling, DOI 10.1007/s10270-010-0157-9, Springer-Verlag, 2010.

[97] I. Ober, S. Graf, and I. Ober. *Validating timed UML models by simulation and verification*, International Journal on Software Tools for Technology (DOI 10.1007/s10009-005-0205-x), Springer, 2005.

[98] O. Nierstrasz. *Regular Types for Active Objects*, in Proc. of OOPSLA, Washington D.C., 1993.

[99] O. Nierstrasz. *Towards an object calculus*, in Proc. of Object Based Concurrent Computing, volume 612 of Springer LNCS, 1992.

[100] O. Niestrasz and M. Papathomas. *Viewing objects as patterns of communicating agents*, in Proc. of OOPSLA, 1990.

[101] Object Management Group, Inc. (OMG). *Unified Modeling Language Specification, Version 1.5*, March 2003. Available from http:// www.omg.org.

[102] Object Management Group, Inc. (OMG). *UML Profile for Schedulability, Performance, and Time Specification, Version 1.0*, Sept. 2003. Available from http:// www.omg.org.

[103] Object Management Group, Inc. (OMG). *Unified Modeling Language: Superstructure version 2.3*, May 2010. Available from http:// www.omg.org.

[104] Object Management Group, Inc. (OMG). *Semantics of a Foundational Subset for Executable UML Models (FTF Beta 2)*, November 2009. Available from http:// www.omg.org.

[105] A. Pnueli. *Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends*, in Current trends in concurrency. Overviews and tutorials. Springer-Verlag, New York, NY, USA, 1986.

[106] A. Pnueli and M. Shalev. *What is in a step: On the semantics of Statecharts*, in Proc. of Theoretical Aspects of Computer Software (TACS' 91), LNCS 526, pp. 244-264, Springer, 1991.

[107] W.Reisig. *Petri Nets. An introduction*, in W. Brauer, G. Rozenberg and A. Salomaa, editors. EATCS Monographs on Theoretical Compute Science, Volume 4, Springer-Verlag, 1985.

[108] C. Rossi, M. Enciso, and I.P. de Guzman. *Formalization of UML state machines using temporal logic*, Software and System Modeling Journal (2004), Volume 3, pp. 31-54, Springer, 2004.

[109] E.Roubtsova, J.van Katwijk, W.J.Toetenel, C.Ponk, and R.C.M de Rooij. *Specification of Real-Time Systems in UML*, Electronic Notes in Theoretical Computer Science, Volume 39, Number 3, Elsevier, 2000.

[110] R.K. Runde, ø. Haugen, and K. Stølen. *How to transform UML neg into a useful construct*, in Proc. of Norsk Informatikkonferanse, pp.55-66, Tapir, 2005, Also Research Report 326, Nov. 2005, Dept. of Informatics, University of Oslo.

[111] A. Refsdal, K.E. Husa and K. Stølen. *Specification and Refinement of Soft Real-Time Requirements Using Sequence Diagrams*, in Proc. of FORMATS'2005, pp. 32-48, LNCS 3829, Springer, Sweden, 2005.

[112] A. Refsdal, R.K. Runde and K. Stølen. *Underspecification, Inherent Nondeterminism and Probability in Sequence Diagrams*, in Proc. of FMOODS'2006, LNCS 4037, pp.138-155, Springer, Italy, 2006.

[113] C.Sibertin-Blanc. *Cooperative Objects: Principles, Use and Implementation*, in Proc. of Concurrent Object-Oriented Programming and Petri Nets, G. Agha , F. De Cindio (Eds.), LNCS 2001, pp. 210-241, Springer, 2000.

[114] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. *Using Description Logic to Maintain Consistency Between UML Models*, in Proc. of UML 2003, LNCS 2863, San Francisco, CA, USA, pp.326-340.

[115] C. Sibertin-Blanc, O. Tahir and J. Cardoso. *Interpretation of UML sequence diagrams as causality flows*, in Proc. of ISSADS'2005, (LNCS 3563), pp. 126-140. 2005.

[116] Harald Störrle. *Assert, Negate and Refinement in UML-2 Interactions*, in Proc. of Workshop on Critical Systems Development with UML (CSDUML'03 Proceedings), pp. 79-94, Technical Report TUM-I0317, Institut für informatik, Technische Universität München, 2003.

[117] Harald Störrle. *Trace Semantics of Interactions in UML2.0*, Technical Report TR0403, University of Munich, Sept. 2004.

[118] B. Stroustrop. *The C++ Programming Language*, Addison-Wesley, 1986.

[119] A. Taleghani and J.M. Atlee. *Semantic Variations among UML State Machines*, in Proc. of MoDELS' 2006, pp. 245-259, 2006.

[120] J. Trowitzsch, D. Jerzynek, and A. Zimmerman. *A toolkit for performability evaluation based on stochastic UML state machines*, in Proc. of the 2nd international conference on Performance evaluation methodologies and tools, 2007.

[121] A.C. Uselton and S.A. Smolka. *A Compositional Semantics for Statecharts using Labeled Transition Systems*, in Proc. of the 5th International Conference on Concurrency Theory, LNCS 836, Springer, Uppsala, August 1994.

[122] D. Varro. A formal semantics of UML statecharts by model transition systems, in Proc. of International Conference on Graph Transformation (ICGT' 02), LNCS 2505, pp. 378-392, Springer, Barcelona, Spain, October 7-12 2002.

[123] A. Valmari. *Stubborn sets for reduced state space generation*, in Proc. of Advances in Petri nets' 90, LNCS 483, Springer, 1990.

[124] K. Varpaaniemi. *On the stubborn set method in reduced state space generation*, Research reports series n51, Hilsinki University of Technology, May 1998.

[125] W. van der Aalst, K. van Hee, and R. van der Toorn. *Component-based software architectures: A framework based on inheritance of behaviour*, Science of Computer Programming, Volume 42, Number 2-3, pp. 129-171, Elsevier, Feb/Mar 2002.

[126] M. van der Beek. *A comparison of StateCharts Variants*, in Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT' 94), LNCS 863, pp.128-148, Springer, 1994.

[127] K. Verschaeve and A. Ek. *Three scenarios for combining UML and SDL' 96*, in Proc. of 8th SDL Forum, Montréal, 1999.

[128] R. van Glabbeek. *The linear time-branching time spectrum*, Report CS-R9021, Centrum voor Wiskunde en Informatica (CWI), The Netherlands, May 1990.

[129] R. van Glabbeek. *The linear time-branching time spectrum II; The Semantics of Sequential Systems with silent moves, (extended abstract)*, in Proc. of CONCUR'93, 4th Intl. Conference on Concurrency Theory, Hildesheim, Germany, LNCS 715, Springer-Verlag, pp. 66-81, August 1993.

[130] R. van Glabbeek. *The linear time-branching time spectrum I; The Semantics of concrete, sequential processes*, Handbook of Process Algebra, pp.3-99, 2001.

[131] R. van Glabbeek and U. Goltz. *Refinement of actions and equivalence notions for concurrent systems*, Acta Informatica, **37**, 229-327, 2001.

[132] R. van Glabbeek and W.P. Weijland. *Refinement in branching time Semantics*, in Proc. of AMAST Conference, Iowa, USA, pp.197-201, May 1989, (also Report CS-R8922, CWI, Netherlands).

[133] H. Wehrheim. *Checking Behavioural subtypes via Refinement*, in Proc. of the 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS' 2002), the Netherlands, 20-22 March 2002.

[134] P. Wegner and S.B. Zdonik. *Inheritance as an Incremental Modification Mechanism or What like Is and Isn't Like*, in Proc. ECOOP'88, Oslo, LNCS 322, Springer, pp. 55-77, August 1988.

[135] Xiaosham Li , Zhiming-Liu and He Jifeng. *A formal semantics of UML sequence Diagram*, in Proc. of Australian Software Engineering Conference 2004, place country-region Australia. April 2004.

[136] S. Yao and S.M. Shatz. *Consistency Checking of UML Dynamic Models Based on Petri Net Techniques*, in Proc. of the 15th International Conference on Computing, 2006.

[137] T. Ziadi, L. Hélouët, and J-M. Jézéquel. *Revisiting statechart synthesis with an Algebraic Approach*, in Proc. of International conference on Software Engineering (ICSE'04). 2004.