Deriving Distributed Design Models from Global Requirements Models

by

Mohammad Fawzi Ahmad Al-Hammouri

Thesis submitted to the
Faculty of Engineering
in partial fulfillment of the requirements for the
Doctorate in Philosophy degree in Electrical and Computer Engineering

School of Electrical Engineering and Computer Science (EECS)
Faculty of Engineering
University of Ottawa

Abstract

During the system and software development process for distributed systems, the development of the overall system design is critical for correctness, performance, and reliability. The objective of this thesis is the improvement of methods and tools that can be used to obtain a correct design model for distributed system components automatically by deriving the design model from the global system requirements. Mainly, we are concerned with the transformation from a global requirements model to a distributed design model. The global requirements model describes the behavior of a distributed system in an abstract manner by defining the local actions to be performed by different roles which represent actors in the different system components. The distributed design model defines the behavior of each actor separately, including its local actions plus the exchange of coordination messages, which are necessary to assure that the actions are performed in the required order.

In this work, we first consider a global requirements model in the form of partially ordered actions similar to High-level Message Sequence Chart (HMSC). We study the realizability of the global requirements, which is said to be directly realizable if a design model can be constructed without any coordination messages. We study some problems which prevent direct realizability, such as strict sequence, non-local choice, non-deterministic choice, termination race, and others, and show under which conditions these problems are absent and the global model is directly realizable. For the other cases, we show how a conforming design model can be obtained by introducing a minimal number of coordination messages. In this context, we also show under which conditions sequence numbers are required in the messages of a weak while loop.

Then we study the automatic derivation of a distributed design model using a tool. In order to obtain an easily readable notation for the global requirements model, we adapt the HMSC notation to the UML Hierarchical State Machine (HSM) notation and extend this notation to describe the roles that participate in the actions of each state of the global behavior. A simple state represents some local actions of a single role, while a hierarchical state usually represents a collaboration between several roles. Then we describe a derivation algorithm that can be applied to a global model written in this proposed HSM notation and

generates a distributable UML HSM model, which contains a hierarchical state machine for each role of the application.

We implemented this derivation algorithm as a tool in the context of the Umple UML development environment. This tool takes a global requirements model written in the extended HSM notation as input and automatically generates a UML HSM model. The distributed implementation environment described in Zakariapour's thesis is used for generating a distributed Java implementation, where each distributed component contains one Java run-time environment and realizes the behavior of one or several of the roles of the application. A Travel Management System illustrative example has been discussed to illustrate the representation of the global model using the extended HSM notation and to demonstrate the correctness of the generated design models by the tool.

Acknowledgements

First of all, I thank ALLAH, the almighty, for helping me to complete this work.

I would like to express my sincere gratitude to my supervisor, Dr. Gregor v. Bochmann, whom I had the luck to work with during my Ph.D. study. His inspirational thoughts, guidance and assistance, patience, careful review, and financial support, helped me in all the time of research and writing of this thesis. He provided me with valuable support and gave his time and experience generously. Thank you Dr. Bochmann!

Many thanks to the Hashemite University in Jordan for its financial support. And many thanks to the University of Ottawa for giving me the opportunity to complete my Ph.D. degree in such a great environment.

My grateful thanks to my father Fawzi and mother Haifa for their love and support. They have always encouraged me toward success. I also want to thank my brothers and sister for their support.

Working far from home in a foreign country can be difficult sometimes. Fortunately, I have been lucky to have a wonderful group of friends that make me feel like at home. Special thanks to all my friends here in Canada for their support.

My family deserves the biggest of the acknowledgements. Grateful thanks to my great wife, Sara, my daughter Bailasan, and my son Zaid. They are always standing beside me with their love and unconditional support. They make my life more meaningful.

Dedication

This thesis is dedicated to my beloved parents, and my lovely family, Sara, Bailasan, and Zaid.

Table of Contents

List of Figures

List of Tables

List of Acronyms

BPEL	Business Process Execution Language
CIM	Choice Indication Message
CIMC	Choice Indication Message for the Composite States
EPP	End-Point Projection
FIFO	First In First Out
FM	Flow Message
HMSC	Hierarchical (also called High-Level) Message Sequence Chart
HSM	Hierarchical State Machine
IR	Initiating Roles
LPOSET	Labelled Partially Ordered Set
MO	Message Overtaking
MSC	Message Sequence Chart
OMG	Object Management Group
POMSET	Partially Ordered Multi-set
PR	Participating Roles
SOC	Service Oriented Computing

TR	Termination Race
UML	Unified Modeling Language
WS-CDL	Web Services Choreography Description Language

Chapter 1

Introduction

During the system and software development process for distributed systems, the development of the overall system design is critical for correctness, performance, and reliability. The objective of this research is the improvement of methods and tools that can be used to obtain a correct design specification for a distributed system automatically by deriving the design specification from the global system requirement, and this is called protocol derivation. Mainly, we are concerned with the transformation from a global requirements model, which describes the behavior of a distributed system in an abstract manner by defining the local actions to be performed by different roles which represent actors in the different system components, to a distributed design model, which defines the behavior of each role separately, including its local actions plus the exchange of coordination messages which are necessary to assure that the actions of the different roles are performed in the required order.

Different notations were proposed for describing requirements models. Alur [1] used Message Sequence Chart (MSC), and Hierarchical Message Sequence Chart (HMSC) (also called high-level MSC), Castejón [2] proposed the concept of Collaborations and Activity Diagrams, and PO-Charts were used in [3].

Bochmann [4] proposed a derivation algorithm for semi-automatically deriving a correct distributed system design with different system roles that conforms to the global specification (written in partial order notation (POS)). Coordination messages between system

roles are introduced to avoid any conflicts and ensure synchronization. F. Laamarti, in her master thesis [5], implemented this algorithm in a software tool which generates the behavior (design model) for each role from a global specification in the form of an Activity Diagram. The translation of the generated distributed design model into BPEL (Business Process Execution Language) was experimented by [6].

In this thesis, first, we consider a global requirements model in the form of partially ordered actions represented by collaborations [7], or HMSC. A HMSC determines different roles participating in the distributed system and different actions executed by these roles; also, there are messages between roles that indicate the partial order between the actions. We study the realizability of the global requirements model, which is said to be directly realizable if a design model can be constructed without any coordination messages. We study some problems which prevent direct realizability, like strict sequence [8], non-local choice [9], non-deterministic choice [10], and race conditions [11], and show under which conditions these problems are absent, and the specification is directly realizable. For the other cases, we show how a conforming design model can be obtained by introducing minimal changes to the basic implementation. Overall, this is an important improvement of the previous work in [4].

Secondly, we study what kind of notations would be suitable for modeling the behavior of distributed applications (global model), and which one will be the easiest to use as input for a tool that derives the role design model automatically. In this context, we adopt the UML Hierarchical State Machine (HSM) and then introduce certain extensions to this notation in order to describe the different roles which participate in the global model. The local action which is executed by a given role is indicated as a do-action in a simple state which corresponds to that role. This proposed notation is similar to HMSC; however, there is no message exchange, and the partial order is indicated as a completion transition between the states.

Then we follow the improved derivation algorithm and show how a design model can be derived from a global model written in the extended HSM notation. The derived design model for each role is a UML HSM, which contains the local actions of that role and the

required coordination messages which are generated by our algorithm.

Finally, we build a tool that automatically derives a role design model from a global requirements model written in our notation of extended Hierarchical State Machine. This tool implements the derivation algorithm and does a model transformation from our extended HSM, representing the global requirements model, into UML HSM, representing the design models for the different roles. In this context, we extend Umple [12] to support our notation for the global model and to derive distributed design models based on our derivation algorithm. Then, a Travel Management System case study has been discussed to illustrate the representation of the global model using the extended HSM notation and to demonstrate the correctness of the generated design models by the tool.

1.1 Motivation

Modeling a set of distributed system components such that their behavior conforms to a given global specification is really a challenging task. Many problems prevent the realization of global requirements; such problems include race conditions, deadlocks, non-local choice, non-deterministic choice, and others. One solution is to derive the design model for each role separately, then integrate these models together and adding additional coordination messages if necessary.

Synthesis methods have been used to automatically derive a distributed design model (also called protocol specification) from a given global model specification. Derivation methods have been used to specify and derive the required coordination messages in order to get a correct design model.

The improvement of methods and tools that can be used to automatically obtain correct system design models (for the participating roles) from the global requirements are really needed in the field of distributed communication modeling. Our work is important in the context of distributed system design, where designers and developers should consider the distributed application problems and know how to solve them. This work is also important for the construction of tools that generate code for distributed applications in

order to generate code without design flaws.

1.2 Objective

Our objective in this research is to improve the design methods for distributed applications and, in particular, review the protocol derivation algorithm, which derives from a given requirements for the global specifications of the system, a distributed design that is correct by construction. We study what kind of coordination messages are necessary for the generation of correct design models and improve the derivation algorithm in [4] by minimizing the required coordination messages to get a realizable requirements model. In addition, we discuss the problem of non-local choice, and more specifically, with competing initiatives. This work is important in the context of the automatic derivation of correct distributed design models from global requirements.

The second objective is finding a suitable, abstract modeling notation for the behavior of distributed applications and then developing an algorithm that generates a model of the behavior for each role of the distributed application, including the exchange of messages necessary for the coordination of the activities. In this context, we study what kind of notations would be suitable for modeling the global model and which one will be easiest to use as input to the tool in order to generate a design model for each role in the global model.

The third objective is the creation of such a tool that is useful for the development of distributed computing applications. This tool supports the proposed modeling notation, and it is used for an automatic generation of a distributed design models from a given global requirements model.

1.3 Research Questions

This thesis will address the following research questions:

- **RQ1**: *What kind of coordination messages are necessary for distributed design?*
- **RQ2**: *Which kind of notations would be suitable for modeling the behavior of distributed applications, and which one will be the easiest to use as input for a tool?*
- **RQ3**: *In a distributed system, if more than one role take uncoordinated choice initiatives concurrently, how can this problem be solved?*
- **RQ4**: *Which existing tool can be extended to support the proposed notation for the global model, and be extended to automatically derive distributed design models from the global specification?*

1.4 Thesis Contributions

By addressing the research questions (RQ1 to RQ4), this thesis provides the following **major** contributions:

- Extension of a hierarchical state machine (HSM) language used for global requirements models.
- Improved version for the derivation algorithm of the design models in [4], which derives a roles design model from a global requirements model by minimizing the required coordination messages to get a correct design model. Then we adapt this algorithm to support the proposed global model. In this context, we provide the following:
  - A reduction in the required coordination messages to get a correct design model.
  - Butter support for the derivation of the weak while loop, choices, and competing initiatives.
  - Application of the algorithm can be applied to the global model written in the proposed HSMs notation.

- A prototype tool that extends Umple HSM, and transforms a HSM global model into a distributed design consisting of a collection of UML HSMs, one per role.

The thesis also provides the following **minor** contributions:

- Illustrative example to show the representation of the global model using the extended HSM and to demonstrate the correctness of the generated design models by the tool.
- A study of competing initiatives in distributed applications and the discussion of some distributed design choices.
- Improvement of the existing Umple self generator, for generation of a model written in Umple syntax from an Umple internal representation.

1.5 Thesis Outline

The rest of this thesis is organized as follows:

- Chapter 2 introduces the basic concepts and notations that are important as background for this thesis.
- Chapter 3 is a literature review.
- Chapter 4 discusses the modeling using partial order and the realizability of the global requirements model.
- Chapter 5 discusses a role based modeling paradigm based on HSM. It describes an algorithm for the derivation of design models from a global requirements model written in extended HSM notation.
- Chapter 6 discusses resolving competing initiatives in distributed applications.
- Chapter 7 presents the overall architecture of the derivation software tool, which derives a distributed design model automatically.

- Chapter 8 presents a case study and the testing of the tool.
- Chapter 9 presents the conclusion and future work.

Chapter 2

Background

In this chapter, we cover some important concepts and modelling paradigms which were used for describing the behavior of a distributed system in a global view.

2.1 Strict and Weak Sequences

The strict sequence between two activities A and B (written as $A ;_s B$) means: the actions of A must be completed before any actions of B may start [13]. The weak sequence between two activities A and B (written as $A ;_w B$) means: for any role r, all actions of A at r must be completed before any action of B at r may start [8]. This allows any role to start action in B as soon as it has finished with A without waiting for the other roles to finish as well. Figure 2.1 (a) shows an example of strict sequence A ;s B where the solid arrows represent activity A and the dashed arrows represent B. The actions in B can't start until receiving a coordination message (dotted arrow), which indicates the end of activity A. The same example is represented using the weak sequence in Figure 2.1 (b), where the coordination messages are not needed.

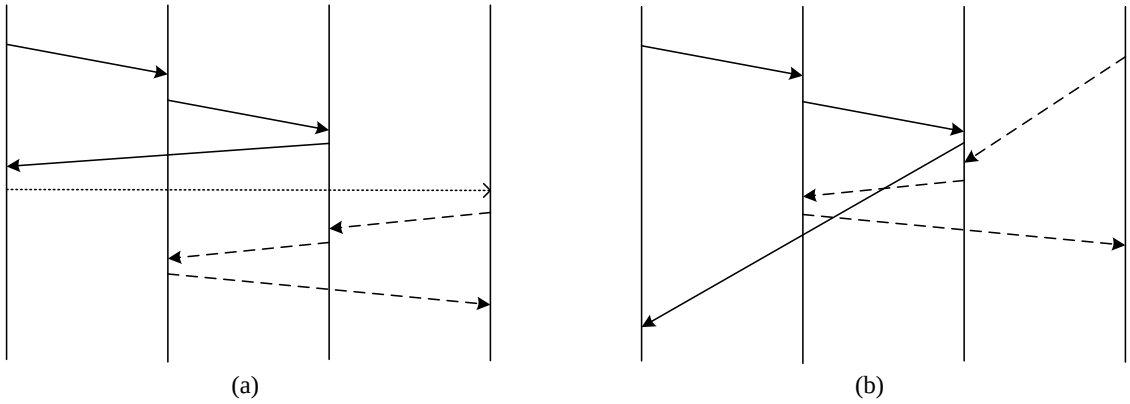


Figure 2.1: Example of (a) strict sequence, (b) weak sequence

2.2 The Concept of Collaboration

The UML collaborations concept is proposed by [2] for modeling the global requirements (behavior of the distributed system from a global point of view). A collaboration determines different roles in a distributed environment and different actions executed by those roles. In the requirements model, a certain number of roles are identified, and each action is performed by one of these roles such that their execution satisfies a given partial order [14]. One of the interesting features of collaboration is: it allows the composition/decomposition into sub-collaborations using collaboration uses [2], one distinguishes between a composite collaboration (a collaboration that is decomposed of sub-collaborations), and an elementary collaboration (a collaboration that is not further decomposed into sub-collaborations) which is often simple and reusable. Figure 2.2 shows an example for collaboration C which contains three roles x, y, and z, and six actions, a_i ($i = 1, 2, ...6$). The actions located on the collaboration border are initiating or final actions.

In any collaboration, there are initial and final actions and initiating and terminating roles:

- Initiating Action: An action is *initial* if no other action precedes it in that collaboration. The action a_1 in Figure 2.2 is an example.
- Final Action: An action is $final$ if no other action succeeds it in that collaboration.

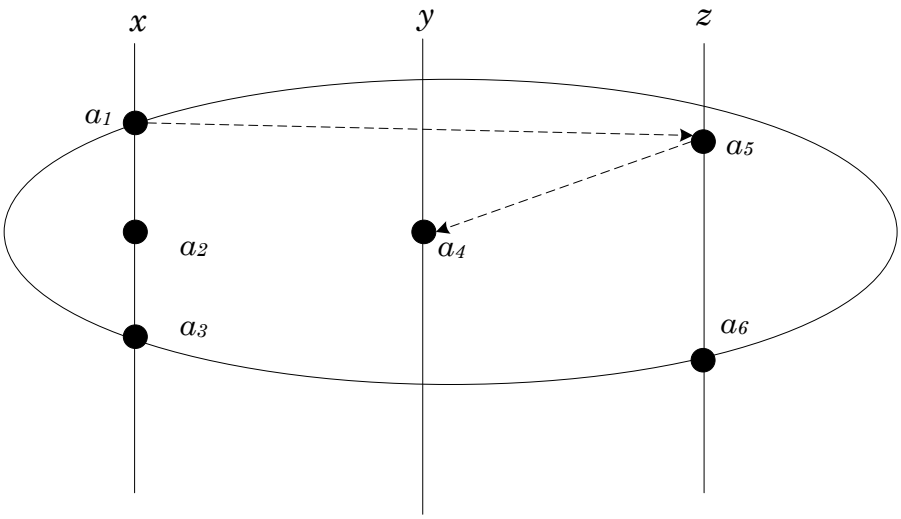


Figure 2.2: Example of collaboration C.

a_3 and a_6 are examples.

- Initiating role is a role that performs an initiating action in the collaboration or sub-collaboration. The role x is an example.

- Terminating role is a role that performs a final action in the collaboration or sub-collaboration. The roles x and z are examples.

Collaboration C has one initial action: a_1, and two final actions: a_3 and a_6. We can also figure out some order dependency for the inner actions in Collaboration C. For example, the inner action a_2 happens after a_1 (local dependency). Also the action a_4 happens after the initial action a_1 and before the final actions a_3 and a_6 (since it is not a final action). The partial order (see the dashed arrows in Figure 2.2) can also be used in collaborations [7] to indicate the order constraints between actions. For example, the action a_4 happens after the action a_5.

The concept of collaboration does not describe the dynamic form of the behavior [2]. For describing the behavior dynamically, [2] proposes a decomposition of a collaboration into several sub-collaborations (each may involve two or more roles), and identify their

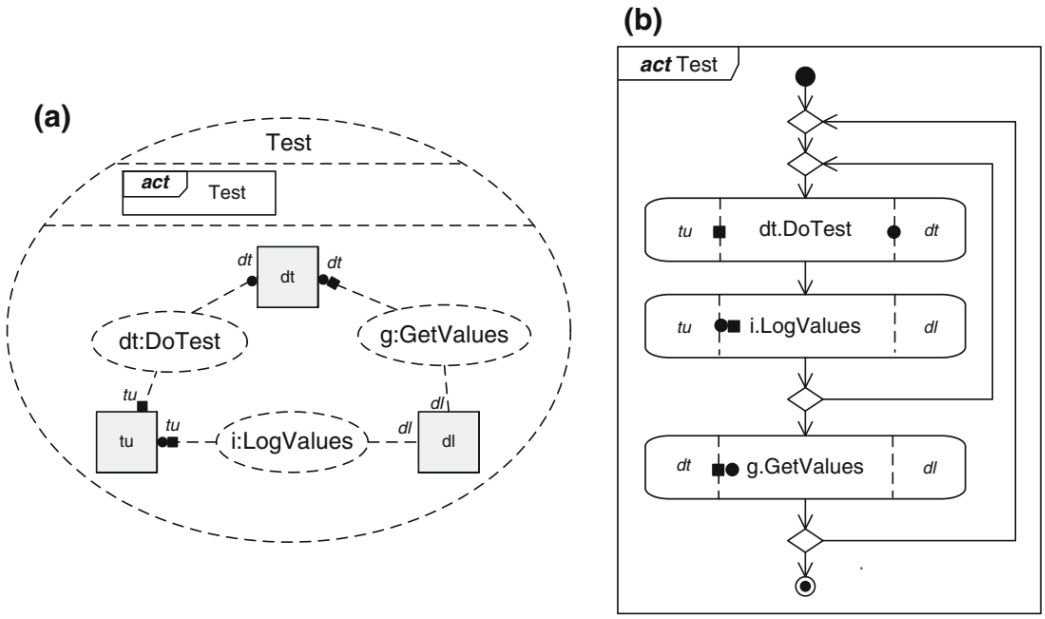


Figure 2.3: Example of a medical test modeled by (a) a UML Collaboration and, (b) an Activity Diagram.

execution order using the sequencing primitives of Activity Diagrams: (1) The dynamic behavior of a collaboration is represented by an Activity Diagram, where each activity represents a sub-collaboration which could involve several roles. (2) For each activity, the initiating roles are indicated by a solid circle, and the terminating roles are indicated by a solid square. (3) Sequencing between activities is either strict or weak.

The example in Figure 2.3 (taken from [2]) presents the model of a medical test. The UML collaboration of Figure 2.3(a) shows three roles: doctor terminal (dt), test unit interfacing the patient (tu), and data logger (dl). The dynamic behavior is shown in Figure 2.3(b). As we see, the test starts with the DoTest sub-collaboration, and each sub-collaboration (activity) is associated with the initiating and terminating roles.

2.3 Message Sequence Chart

Message Sequence Chart (or MSC) is a formal and scenario-based language used to describe the interaction between two or more independent instances (roles) [15, 16]. It contains mes-

sage sending, receiving and internal actions. The MSC can be used to model a collaboration (however, collaboration is more abstract); it concentrates on messages (sending and reception) in addition to internal actions, and there is a standardized graphical notation for it [15]. The messages exchanged between roles are shown by using arrows; where the arrow tail represents the sending of the event, and the head represents the event of reception. Figure 2.4 (a) is an example of a simple MSC with three roles: x, y, and z, and six events: $(x_1, y_1), (y_2, z_1), (x_2, z_2)$, where x_1 and y_1 are the sending and the reception of the message m_1 respectively, y_2 and z_1 for m_2, x_2 and z_2 for m_3. In Figure 2.4 (b) we show the dynamic behavior of MSC X using Collaboration notation (borrowed from [2]) which presents the execution order of the events.

Alur et al. [1] formally defines the semantics of an MSC based on partial orders [14]. It defines the MSC-Graph notation, which is a directed graph and corresponds to the Interaction Overview Diagram in UML [16]. Each node in an MSC-Graph represents an MSC, and each edge represents the sequential execution between two MSCs. It is assumed that all edges in MSC-Graphs represent either a strict sequence (called synchronous concatenation) or a weak sequence (called asynchronous concatenation). The paper also defines Hierarchical MSC (written HMSC) as an extension of MSC-Graphs, where a node contains either an MSC or another MSC-Graph and can be used for modeling complex system specifications. Figure 2.5 (a) shows an example of a loop written in MSC notation, and (b) shows the equivalent MSC-Graphs of the loop.

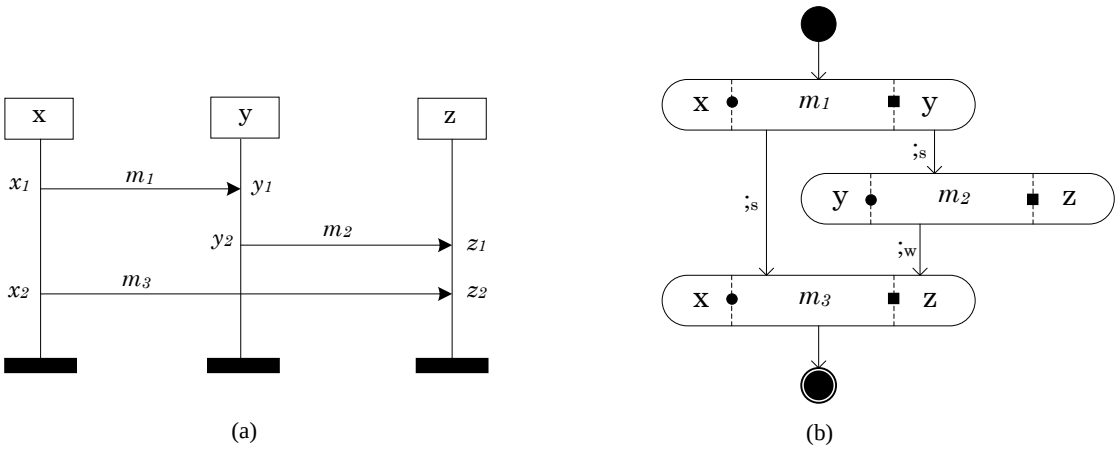


Figure 2.4: Example of (a) MSC X, (b) The dynamic behavior of X.

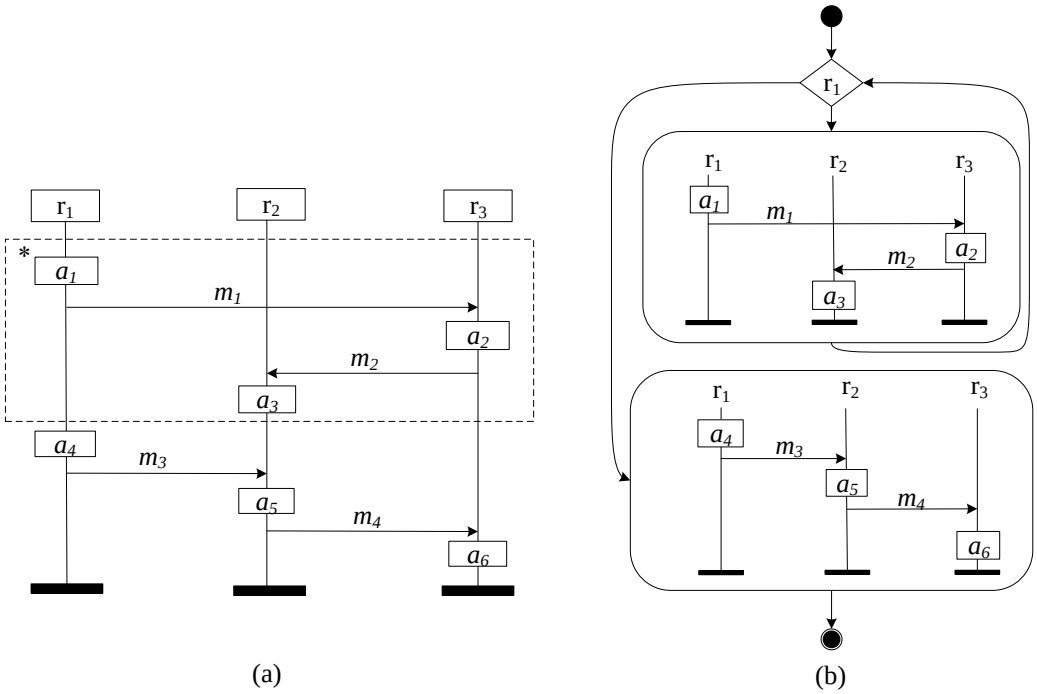


Figure 2.5: Example of (a) MSC, (b) MSC-Graphs

2.4 Partial Order Charts (PO-Charts)

PO-Charts are proposed by [3] for modeling the specification of distributed systems. Each PO-Chart is similar to a HMSC, except that each node of a PO-Chart contains a partial order of events (which we call actions) and the roles that perform these actions. Such partial order is similar to the one in MSC, however, the arrows between events represent a partial order dependency between local actions and not necessarily message exchanges like in MSC. See, for instance, Figure 2.6 which shows a PO-chart that is equivalent to MSC-Graphs in 2.5 (b). The actions (a_1 through a_6) in the MSC-Graphs are executed locally by the roles (r_1, r_2 or r_3) to determine some parameters before messages are sent or to store locally some received information. These actions correspond to the events in the PO-charts where their ordering relationships are defined.

In PO-Charts, both weak and strict sequences can be used for representing the sequential execution between two nodes (each of them is a partial order with roles). Strict sequence ("ss") means the initiating roles of the second node can't start until the last actions of the terminating roles of the first node have occurred. Weak sequence ("ws") enforces the execution order for each role separately. However, in MSC-Graphs and HMSC, only one type of sequencing is allowed in one figure, and therefore, the notation for sequencing was not included. See, for instance, Figure 2.5 (b).

2.5 Labelled Partially Ordered Set (lposet)

A partial order is a natural concept for describing the execution of distributed systems as pointed out by Lamport [17]. Pratt [14] proposed to use labelled partially ordered set (lposet) [18] for this purpose. A (strict) lposet, also called pomset (partially ordered multi-set) is a tuple $(E, \sum, <, l)$, where E is a set of events, $\sum$ is a set of action labels, $< \subseteq E \times E$ is an irreflexive, asymmetric, and transitive order on E (where '$e_1 < e_2$' means that event e_2 is after event e_1, or $e_1 \rightarrow e_2$), and l is a labeling function l: $E \rightarrow \sum$.

The MSC in Figure 2.4 (a) can be described as a lposet as followings: $P = (E, \sum, <, l)$

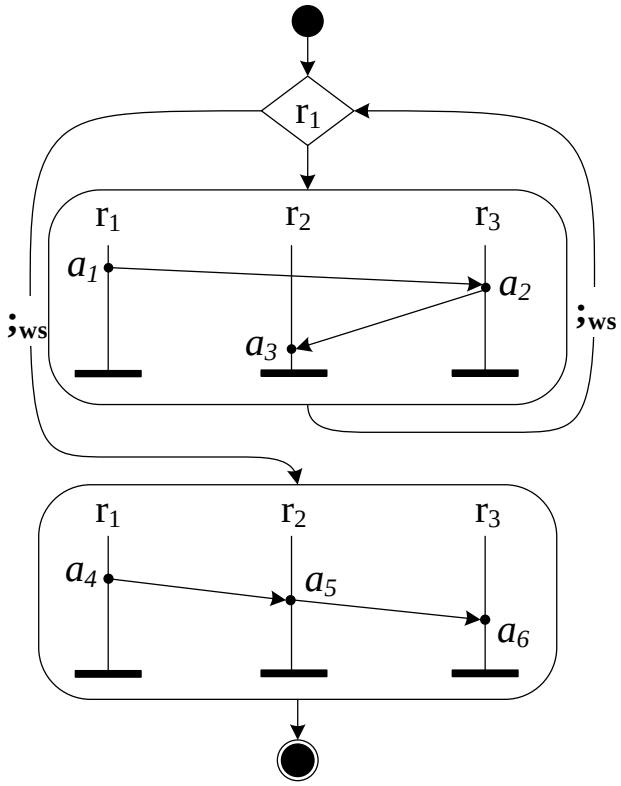


Figure 2.6: Example of PO-Charts.

where $E = \{x_1, x_2, y_1, y_2, z_1, z_2\}$, $\sum = \{!m_1, ?m_1, !m_2, ?m_2, !m_3, ?m_3\}$, $<= \{x_1 < x_2,\ x_1 < y_1,\ y_1 < y_2,\ y_2 < z_1,\ z_1 < z_2,\ x_2 < z_2\}$ and $l = \{\ l(x_1) = !m_1,\ l(x_2) = ?m_1\ ...\ etc\}$, where $!m$, $?m$ represent the sending and receiving of the message m respectively.

The simple collaboration (or MSC) can be described using a single pomset, however in general, the behavioral of a composition collaboration (including a decision or loop) consists of several pomsets. Gischer [19] uses the term "process" to specify a behavioral model (such as a collaboration) and the term "behavior" for a set of pomsets that are allowed for the execution of that process. Gischer defines the following compositions for the behavior of a collaboration (process):

The **strict sequence** of two processes C_1 and C_2, written $C_1\ ;_s\ C_2$, has the following behavior: the set of all strict concatenations of one pomset in the behavior of C_1 with

one pomset in the behavior of C_2 (where the strict combination of two pomsets P_1 and P_2 means that all events of P_2 are after all events of P_1).

The **concurrent** execution of two processes C_1 and C_2, written $C_1 \mid\mid C_2$, has the following behavior: the set of all concurrent combinations of one pomset in the behavior of C_1 with one pomset in the behavior of C_2 (where the concurrent combinations of two pomsets P_1 and P_2 is the pomset that contains the union of events and actions, and no order dependencies between the events of P_1 and P_2).

A **choice** between two alternative processes C_1 and C_2, written $C_1 + C_2$, has the following behavior: it is the union of the pomsets in the behavior C_1 and the behavior of C_2.

For an arbitrary number of repetitions of a process C, the Kleene star operator is defined as usual. The behavior of C^{*s} is defined to be strict sequence of zero, one or more repetitions of C.

2.6 Activity Diagram

An Activity Diagram is a behavioral UML diagram used to model the dynamic behavior of the system [16]. It is used to model the workflow, business process, or procedural logic using graphical representations of activities and actions. An Activity diagram consists of a number of activities linked together by control flows or object flows with support for choice, iteration, and concurrency. The flow directions are represented by arrows.

An Activity diagram can be constructed from the following nodes:

- Initial node: a black circle represents the start of the flow.
- Final node: an encircled black circle represents the ending of all activities.
- Action node: a rounded rectangle represents a sub-activity, simple action or sub-activity diagram to be executed.

- Decision node: a control node represented by a diamond shape, it is used to model the choice between alternatives, it has multiples output flows and only one of them gets executed.
- Flow/edge: represented by an arrow to indicate the flow direction.
- Fork node: a control node represented by a black bar, it has one input flow and multiples output flows. It allows several activities to be executed concurrently (represents the beginning of concurrency).
- Join node: a control node represented by a black bar, it has multiple inputs and one output flows. It represents the ending of concurrency.
- Merge node: a control node represented by a diamond shape, it is used for the merging of various alternate flows. It has multiple inputs and one output.

Figure 2.7 shows an example of activity diagram. In this diagram there is a choice between the activity A_1 and A_2, and after A_2 there is a fork node and then two concurrent activities A_3 and A_4. A_3 and A_4 are joined then merged with the node A_1.

2.6.1 Well-Structured vs Non-Well-Structured Activity Diagrams

We call an activity diagram well-structured if it can be easily converted to a hierarchical state machine diagram. An example of a non-well-structured activity diagram is shown in Figure 2.8.

In addition, the well-structured activity diagram can be represented using the Gischer [19] rules. The Gischer representation for the activity diagram in Figure 2.7 is:

$$C_0 \ ;_s \ (C_1 + C_2 \ ;_s \ (C_4 \ || \ C_3)) \ ;_s \ C_5 \tag{2.1}$$

Here each activity A is replaced by a collaboration C, and we consider the strict sequence ';$_s$' between collaboration. The symbols "+" and "||" refer to choice and concurrent operations, respectively (see Section 2.5).

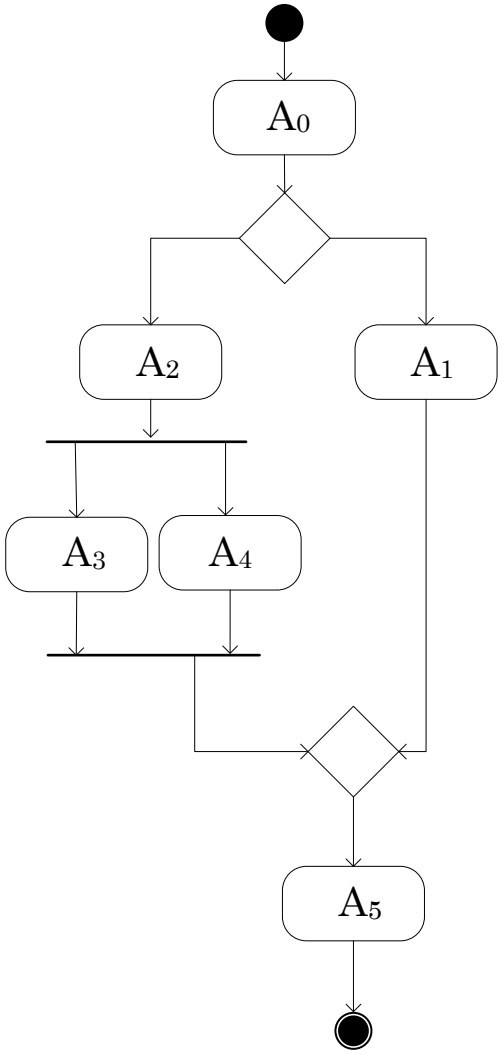


Figure 2.7: Example of activity diagram.

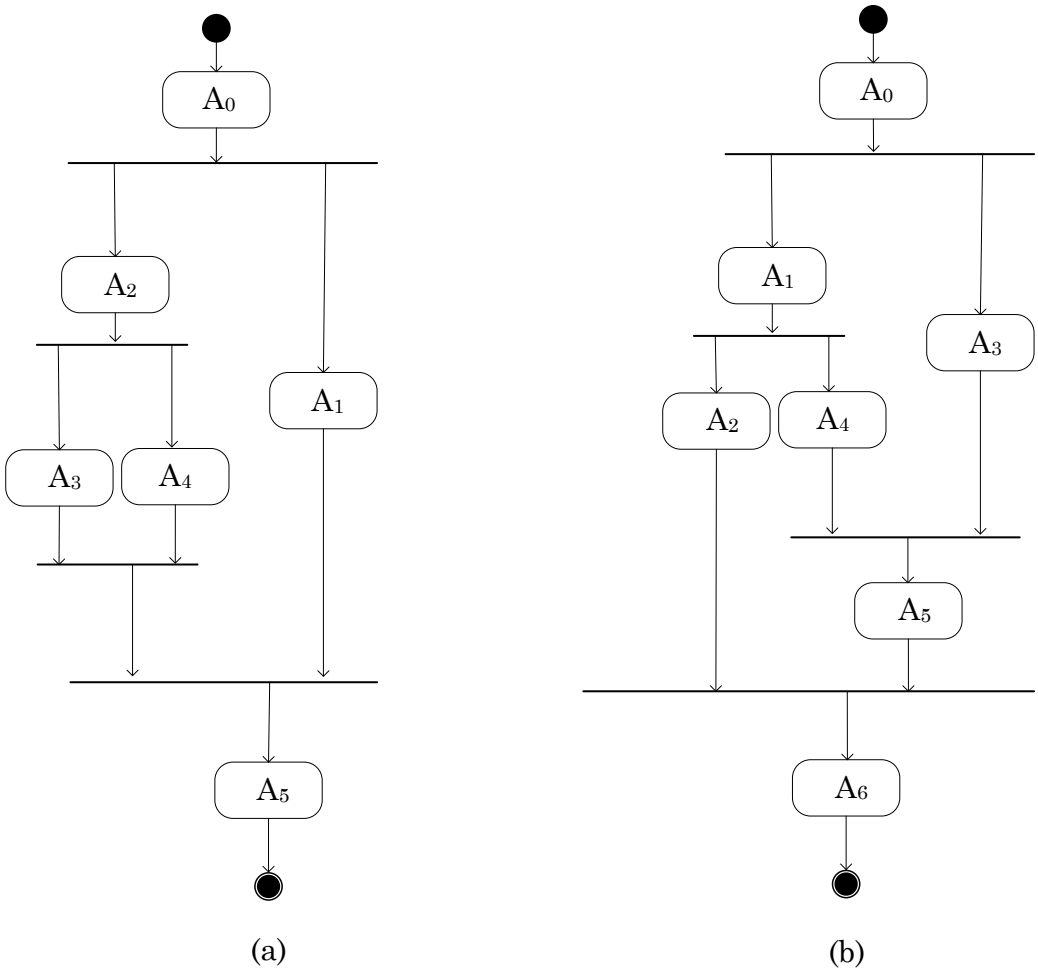


Figure 2.8: Example of (a) well-structured activity diagram, (b) non-well-structured activity diagram

2.7 Modeling State Machines in Umple

In this section, we illustrate the modeling with Hierarchical State Machine (HSM) in Umple. We present the syntax of the Umple HSMs, their meta-model, features, and examples.

2.7.1 Umple: A Modeling and Development Environment

Umple is a technology for Model-Oriented Programming [12]. Umple has the same concept as UML modeling [16], however, it can represent the models in textual syntax. Umple is used for the following purposes:

- Creating UML class and state machine models using a textual representation: In addition to drawing graphical UML models, Umple allows user to create class and state machine diagrams textually, just like programming language code.
- Generate high-quality code from UML models: Umple generates code in languages like Java, C++ and PHP corresponding to the UML class diagrams or state machines.

Umple can be accessed using Umple Online [20]. Figure 2.9 illustrates how to use Umple Online to model a simple state machine textually; it shows both textual and graphical representations for the model. In addition, Umple can be accessed using a command-line compiler, and as Eclipse plugin. More details can be found at the Umple website [12].

2.7.2 Layered Architecture of Umple

Umple architecture can be viewed in three meta-modeling layers like UML (see Figure 2.10), and these layers correspond to the OMG M_2, M_1, and M_0 modeling layers. The Umple meta-model (corresponding to the OMG layer M_2) can be viewed at [21], and it corresponds to only a subset of the UML meta-model. A Umple model (corresponding to the OMG layer M_1) is an instance of the Umple meta-model. It is either a Umple class or a state machine and can be used to describe arbitrary systems. The bottom layer contains the instances (correspond to the OMG layer M_0) of the Umple model; it represents the real objects, states, names, etc., which are found in the running system.

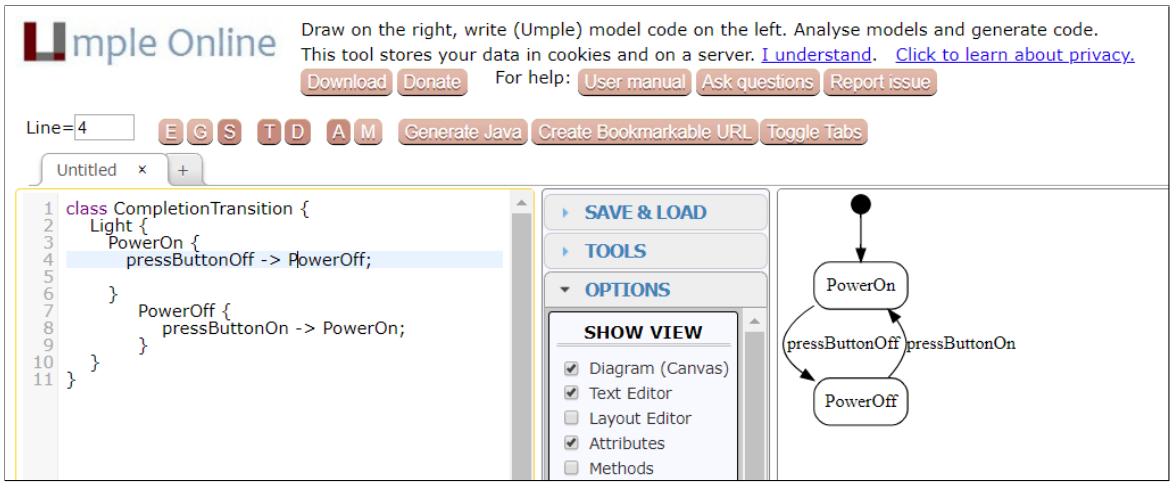


Figure 2.9: Textual and graphical views of an example using Umple Online.

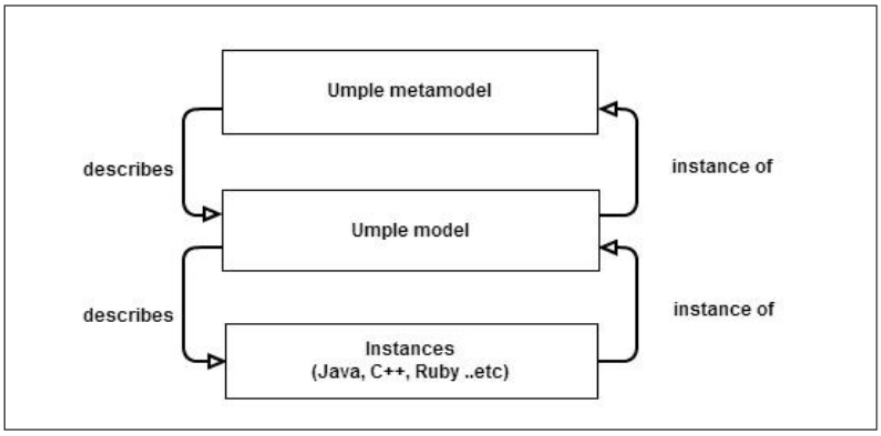


Figure 2.10: Umple meta-modeling architecture [22]

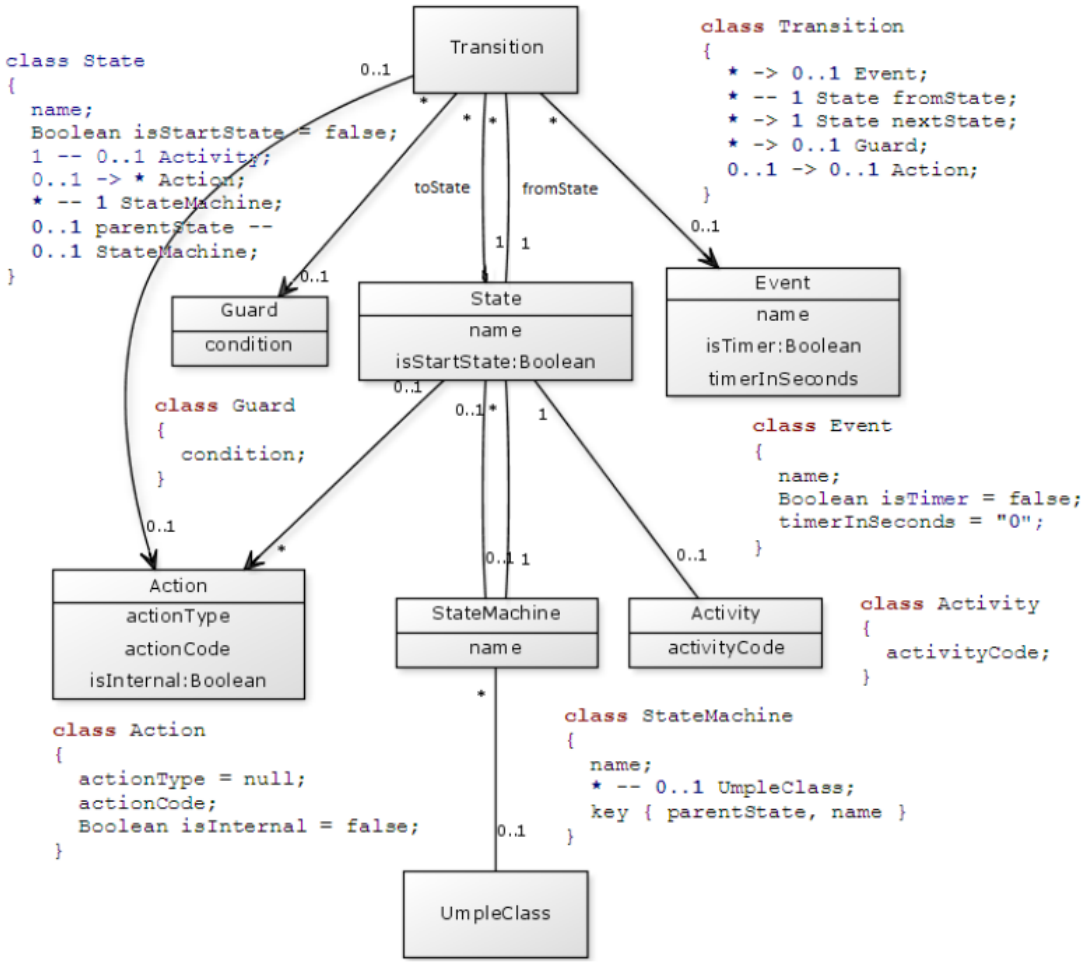


Figure 2.11: Umple Hierarchical State Machine (HSM) meta-model [23]

2.7.3 The Umple State Machine Meta-Model

The Umple state machine meta-model, as shown in Figure 2.11, is similar to the UML state machine meta-model [16]. The main difference is the existence of the UmpleClass in the Umple meta-model. The UmpleClass is associated with 0 or 1, or many state machines. However, a state machine, may or may not be associated with a UmpleClass (Umple supports standalone state machines).

The similarities and differences between Umple and UML state machines in terms of the state machine attributes (state, transition, event, action, etc.) are summarized in Chapter 3 in [23].

2.7.4 State Machines Syntax in Umple

This section illustrates the basic syntax of Umple state machines, which are, in general, hierarchical state machine. Figure 2.12 (taken from [24]) presents a state machine for a car automatic transmission; it is a two-level hierarchical state machine, where the state *Driving* is a composite state, and the other states are simple. The state machine starts in the *Neutral* state, which is defined as the initial state. During this state, the state machine responds to four events: *selectReverse*, *selectFirst*, *selectSecond*, and *selectDrive*. These events trigger transitions to new states; the event, *selectSecond*, for example, triggers a transition to the *Second* state.

In the *Second* state, there are two events with a guards: the event "reachThirdSpeed[driveSelected]" that triggers a transition to the *Third* state, and the event "dropBelowSecondSpeed[driveSelected]" triggers a transition to the *First* state. Both transitions require the guard "*driveSelected*" to be true to take effect.

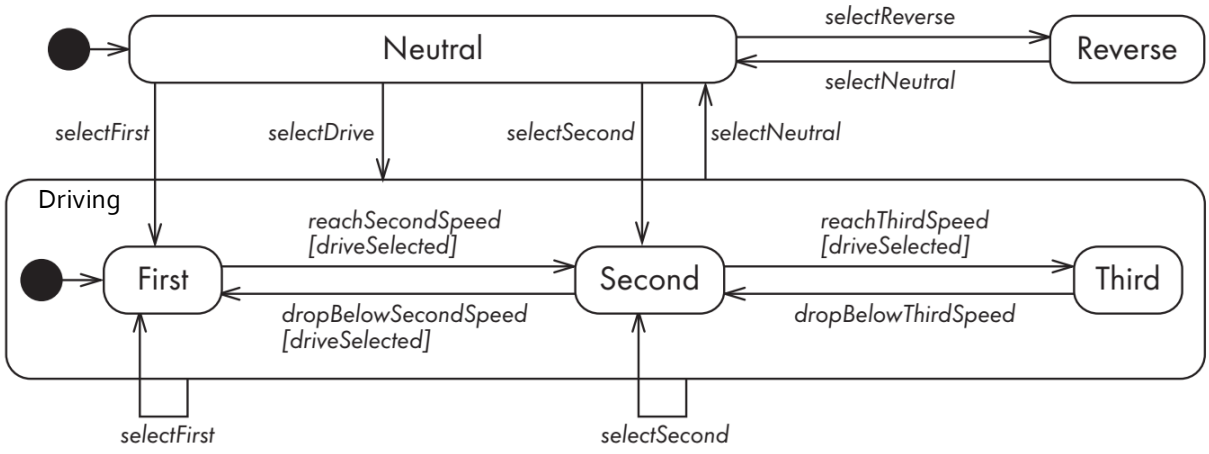


Figure 2.12: State machine for an automatic car transmission (taken from [24]).

Listing 2.1 illustrates how the state machine in Figure 2.12 can be represented in textual Umple state machine syntax:

- **Line 3**: declares a class named "CarTrasnmission".
- **Line 5**: declares the state machine named "smCarTrasnmission".

```
// UML state machine diagram in Umple representing a car automatic
   transmission
//  From Book by Lethbridge and Laganiere, McGraw Hill 2004
class CarTransmission {
  Boolean driveSelected;
  smCarTransmission {
    Neutral {
      selectReverse -> Reverse;
      selectDrive -> Driving;
      selectFirst -> First;
      selectSecond -> Second;
    }
    Reverse {
      selectNeutral/{sendAck();} -> Neutral;
    }
    Driving {
        selectNeutral/{sendAck();} -> Neutral;
        selectFirst -> First;
        selectSecond -> Second;
            First {
                reachSecondSpeed [driveSelected] -> Second;
            }
            Second {
                reachThirdSpeed [driveSelected] -> third;
                dropBelowSecondSpeed [driveSelected] -> first;
            }
            Third {
                dropBelowThirdSpeed-> second;
            }
    }
    public void sendAck()
    {
        System.out.println("Neutral is selected, press the brake");
    }
  }
}
```

Listing 2.1: State machine (written in Umple) representing a car automatic transmission (available on Umple Online [20])

- **Lines 6 to 11**: declare the initial state $Neutral$, and it has four transitions (without guards) to the states: $Reverse$, $Driving$, $First$, and $Second$.
- **Lines 12 to 14**: declare the state $Reverse$. The transition "selectNeutral" (line 13) has an action $sendAck()$. The function $sendAck()$ of the class is defined on the lines 30 through 33.
- **Lines 15 to 29**: declare the composite state $Driving$, and it is comprised of three nested sub-states; $First$, $Second$ and $Third$.

2.7.4.1 States, Transitions, Actions and Activities

Here we discuss the state machine elements and how they are represented in Umple:

1. **States**: A state in Umple can be either simple or composite (contain another state machine inside). The state is represented in Umple by a string (which represents the state name) followed by curly brackets where the state attributes are written. The first state in an Umple state machine is considered (by default) its initial state. The final state should be preceded by the keyword "$final$"
2. **Transitions**: A transition in Umple is written as follows where the $Guard$ and $Action$ are optional:
   $eventName[Guard]/\{Action;\} -> nextStateName;$

   A transition in Umple is associated with the following attributes:

   (a) Event: The transition from any state to another state is triggered by an event. The event "selectNeutral" at line 13 in Listing 2.1 in an example.

   (b) Guard: The Guard is written in Umple in square brackets. The transition is triggered only if the guard is true. The transition with the guard [driveSelected] at line 20 (see Listing 2.1) could also be written as:

   ```
   [driveSelected] reachSecondSpeed -> Second;
   ```

   or

```
reachSecondSpeed[driveSelected == true] -> Second;
```

(c) Action: The action is preceded by "/" and written between curly brackets, see the action "sendAck()" at line 13 in Listing 2.1. The action associated with a transition is called a *transition* action.

The action can be a function call (the same action could be used in a different places):

```
selectNeutral/{sendAck();} -> Neutral;
```

or any native Java code :

```
selectNeutral/{ System.out.println(\Neutral is selected, press
the brake"); } -> Neutral;
```

An action could also be associated with a state as an entry or exit action:

```
Neutral {
    entry /{ System.out.println("entry action");}
    exit /{ System.out.println("exit action");}
}
```

The definition above means: whenever the state machine transits to the Neutral state, the entry action is executed, and whenever it transits out of the state Neutral , the exit action is executed.

3. **Do Activity**: While the transition actions and the entry and exit actions in states take a small execution time, a *do* activity (also called *do-action*) is used if a longer running computation time is needed. The activity can be written inside the state using the keyword *do* followed by a code written between curly brackets:

```
do { sendAlert();}
```

The Java implementation of Umple supports this $do-activity$ code; it starts a thread which calls this code. This allows the state machine to "stay live" and be able to respond to other events of concurrent sub-machines while the do-activity is running. When a transition triggers the state machine outside the state where a do-activity is running, the do-activity is aborted.

```
class FinalStateExample {
    finalStates{
        M {
                T3 -> N;
                A {
                        T1 -> AA;
                }
                final AA
                {}
        ||
                B {
                        T2 -> BB;
                }
                final BB
                {}
        }
        final    N {}
    }
}
```

Listing 2.2: The state machine *finalStates* written in Umple syntax

2.7.4.2 Final State

The final state is a special kind of state signifying that the enclosing region has been completed [16]. When all regions (all concurrent state machines of a state) reach the final state, this means the entire hierarchical state is completed. Based on the UML specification, a final state is a state with no outgoing transitions and no actions (entry, exit, or do-activity).

The following state machine, see "finalStates" state machine in Figure 2.13, has a composite state "M" which contains two concurrent sub-state executions. The first concurrent sub-state starts with a state "A" and ends in "AA" which is the final state for this region, the second sub-state starts with the state "B" and ends in "BB" (final state). The entire state machine is completed in the final state "N". The corresponding Umple code for the state machine "finalStates" is shown in Listing 2.2. The final state defined in Umple using the keyword *final* before the state name. We can see that the final states are empty, they did not have any outgoing transitions or actions.

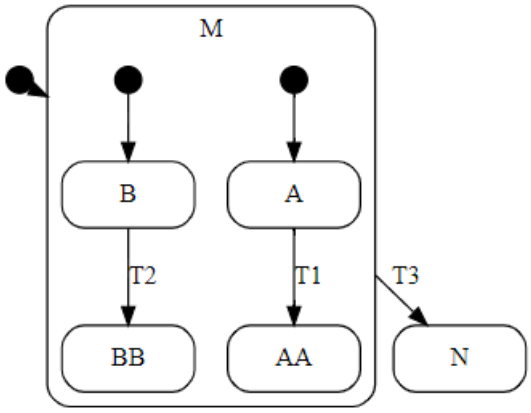


Figure 2.13: The state machine "finalStates".

2.7.4.3 Completion Transition

The *completion* transition is a special kind of transition. The execution of completion means all behaviors associated with the source states have been completed execution (see page 314 in [16]). The completion transition is executed under the following situations:

- In the case of a simple state, when the associated actions and do activity behaviors have completed. If there is no such behavior in a given state, the completion transition is triggered after the state machine enters that state.
- In the case of a composite state, if all regions inside the composite state have reached a final state.

The state machine *completionTransition* (written in Umple syntax) is shown in Listing 2.3, it contains an example of completion transitions. The transitions between the states "A" and "AA" ($->AA$), and the states "B" and "BB" ($->BB$) are examples of completion transitions (see the lines 6 and 13 in Listing 2.3, respectively). The completion transition from a composite to another state is not handled in Umple. We use a transition with a guard [*true*], and this works like a completion transition. The transition ([*true*]$->N$) between the composite state "M" and the state "N" is an example (see line 4 in Listing 2.3).

```
class CompletionTransitionExample {
    completionTransition {
        M {
              [true] -> N;
              A {
                entry /{ System.out.println(" This is A state");}
                -> AA;
              }
              final AA
              {}
        ||
              B {
                entry /{ System.out.println(" This is B state");}
                -> BB;
              }
              final BB
              {}
        }
        final    N {}
    }
}
```

Listing 2.3: Example of a completion transition

2.8 Chapter Summary

In this chapter, we reviewed some modeling paradigms which describe the behavior of a distributed system in an abstract manner. Some of these notations describe the global behavior by defining the local actions to be performed by the different system roles and the partial order between these actions such as Collaboration, Partial Oder Charts (PO-Charts), Labelled Partially Ordered Set (lposet), and Activity Diagrams. The others describe the global behavior in terms of messages exchanged between the participating roles as in the Message Sequence Chart (MSC), MSC-graph, and Hierarchical MSC.

In addition, we explain the modeling of HSM in Umple. We illustrate the syntax and the features of Umple HSM.

Chapter 3

Literature Review

In this chapter, we explore previous works that are related to our research. First, we describe the main problems which are associated with distributed systems, such as *race condition*, *non-local* choice, *choice propagation problem*, and *implied scenarios*. Then we discuss the problem of realizability and present the previous work on the derivation of distributed design models from a global requirements model.

3.1 Distributed System: Coordination Problems

The term distributed system is usually used when different components communicate with each other using messages. These components are located on the same local machine or on physically separated devices interconnected by a network. Transforming the global requirements model, which describes the behavior of a distributed system in an abstract manner by defining the local actions to be performed by different roles which represent actors in the different system components, to a distributed design model, which defines the behavior of each system role separately, leads to many problems. Such problems include race conditions, deadlocks, non-local choice, non-deterministic choice, and others. Most of the work in the literature use MSC [15], UML interaction sequences [25], or the concept of collaboration [25, 26, 2] for the modeling of the global requirements model. In the following, we discuss in detail the above problems and the proposed solutions to solve them.

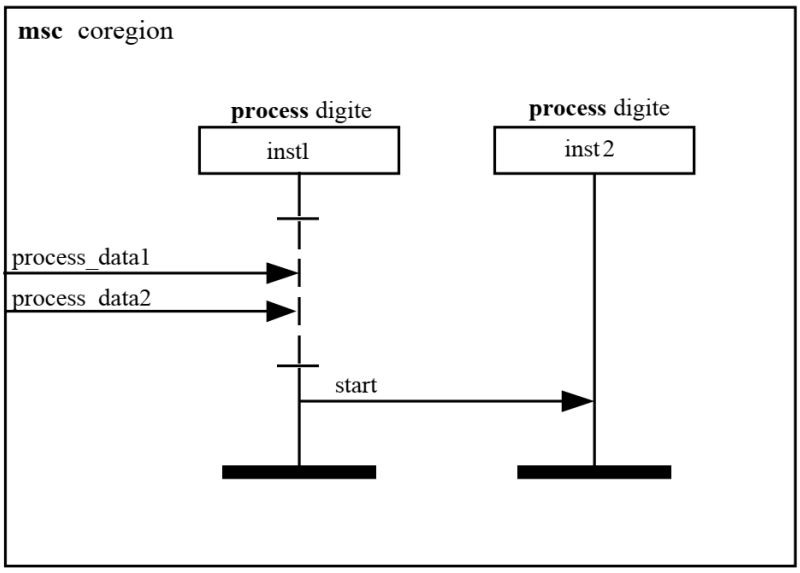


Figure 3.1: MSC Coregion (taken from [15]).

3.1.1 Race Conditions

In a distributed system, a message may arrive at a certain role in a different order than what is expected, and this situation is called a *race condition*. A race condition occurs when the implementation fails to conform to the order of the global requirements model [11]. In general, a race condition may happen when a certain role in the system requirements requires a receiving event to happen after another receiving or sending event, and both are located in the same role. A possible race condition could happen in the MSC of Figure 2.4 (a). We note that the messages m_2 and m_3 may lead to a race condition at the reception by role z. That is, m_3 may arrive before m_2, and this is inconsistent with the MSC which requires m_2 to be received before m_3.

To avoid race conditions, the MSC Coregion (concurrent region) [15] is introduced. This is a situation where the reception of the messages inside the concurrent region is not ordered in time. It is represented using two horizontal bars inside the MSC (see, for instance, Figure 3.1) where the messages '$process\_data_1$' and '$process\_data_2$' are received inside a Coregion. In this example, '$process\_data_1$' could be received before '$process\_data_2$' or the other way round; therefore, the race condition is explicitly allowed.

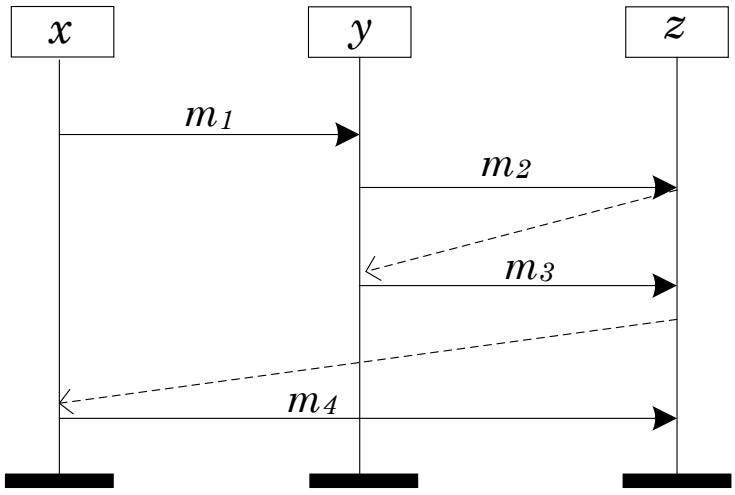


Figure 3.2: Example of Race Condition in MSC.

For analyzing the race condition, one has to distinguish between the sending and receiving events. Any role can control when the sending events should happen, but this is not the case with the receiving events. The underlying communication service affects the occurrence of race condition. Between any pair of roles, in-order delivery prevents race conditions. For example, FIFO communication between y and z in Figure 3.2 (without the dashed arrows) allows the role z to receive m_2 and m_3 in the correct order. But if more than two roles are involved, a race condition will not be prevented using this type of communication, *i.e.*, m_4 may be received before m_3.

Using a pool of received messages which can be requested for consumption in any order (instead of a FIFO queue) solves the problem of race condition (see for instance [27, 28]). One distinguishes between messages reception and consumption, and this allows messages consumption based on the order defined by the receiving role (not based on the order of messages reception). Khendek [28] used the SDL Save construct to achieve such message reordering. This helps to solve the race condition for the messages received from the same source (if no FIFO communication is used), and also solves the race condition when multiple resources are considered.

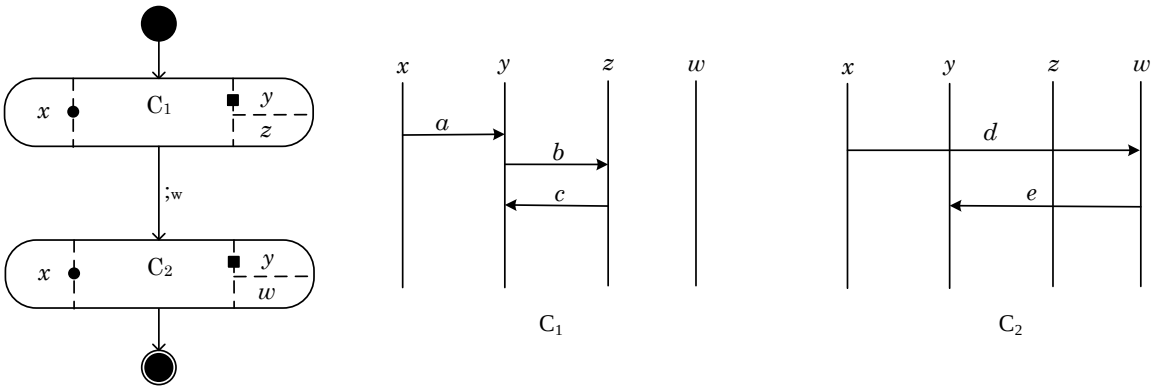


Figure 3.3: Example of race condition in weak sequential composition.

People in the literature discussed the possibility of having separate FIFO queues (one per sender), e.g., suppose the role z in Figure 3.2 (without the dashed arrows) has two separate input queues; one for the inputs from x and another queue for the inputs of y. Then for each given sender (x or y), the messages are consumed by z in FIFO order. When different messages are received at z from the different senders, *i.e.*, m_3 and m_4, where this situation is like having a reception pool at z, in this case, z should decide from which queue should consume first. Consider m_4 is received at z before m_3, then z should wait for m_3 to be received in its queue and consumed, then, z will consume m_4 from the other queue. Therefore, z can decide how to consume the messages received from different sources. The difference between this and the reception pool is: the pool can reorder the messages that are received from the same source, while the separate queues can't do this.

We should note that race conditions may happen not only in simple MSCs or collaborations but also in composed collaborations or HMSC due to weak sequence. Consider the composed collaboration $C_1;_w C_2$ in Figure 3.3, the role x can initiate C_2 after sending a message a in C_1, thereby, the action in C_2 may overlap with actions in C_1. For example, if message a is too slow, message e may be received at y before message c, which leads to a race condition.

Different solutions have been proposed to solve the race condition in MSCs. Synchronization messages are used to eliminate the race condition [29, 30]. The synchronization messages work as acknowledgments to achieve the defined ordering of by the specifica-

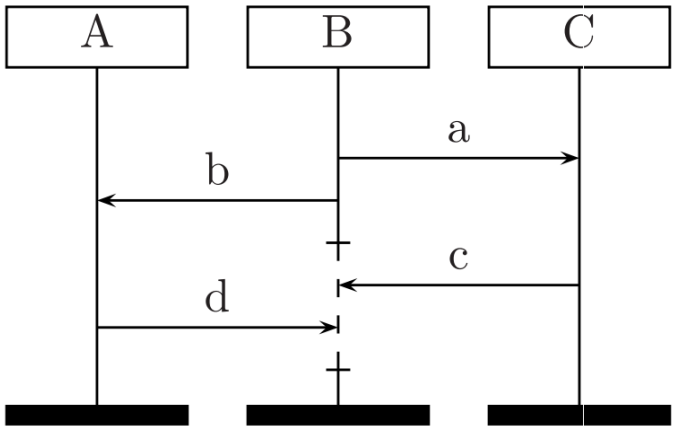


Figure 3.4: Example of a specification written in MSC notation (taken from [29]).

tion (see, for instance, the dashed arrows in Figure 3.2 which represent acknowledgment messages). Mitchell [29] proves that partial order in practice will conform to the order of the specification ($<_c$) by introducing a race-free partial order. He proposed two race-free partial orders: (1) the inherent refinement ordering ($<_R$), which is a minimal strengthening of $<_c$ by introducing addition orderings (using the coordination messages), and (2) the race-free inherent casual ordering ($<_I$), which is a minimal weakening of $<_c$, which is obtained by eliminating certain order relation of the specification to achieve the required order. The overall ordering relationship is: $<_I \subseteq <_c \subseteq <_R$.

Consider the example in Figure 3.4, the specification order asserts that $!b <?c$ (the reception of message c happens after the sending of message b). But in the implementation, C does not know when the event $!b$ occurs. That is, message c could arrive at role B before b is sent.

Based on Mitchell first solution, the inherent refinement ordering strengthens the event's partial order by using additional messages that will eliminate the race condition (see for instance Figure 3.5 (right)). The added coordination message will enforce $?c$ to happen after $!b$. However, in most cases, the reception pool [27] solves the problem without any need for extra messages.

The second solution: The inherent casual ordering changes the specification to achieve the required order (see for instance Figure 3.5 (left), we made a change here compared to

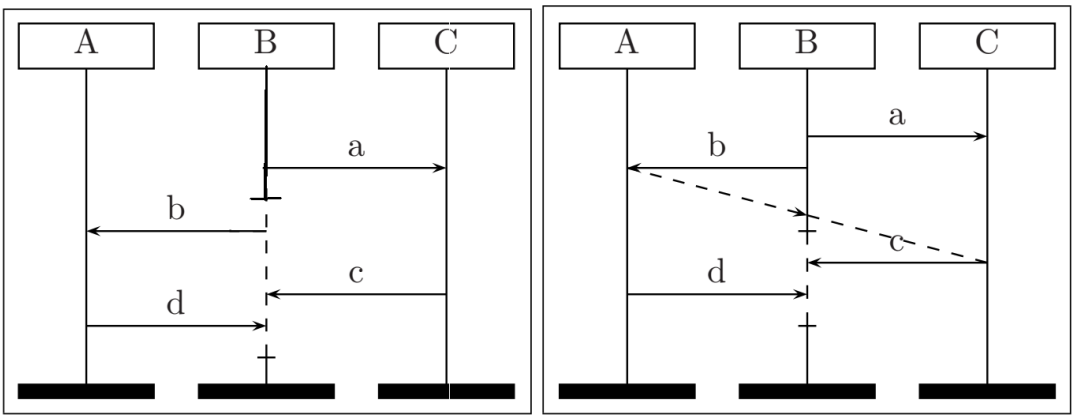


Figure 3.5: Inherent casual ordering (left), and inherent refinement ordering (right).

[29] by making the sending of message a outside the Coregion), because of Coregion, the reception of c and the sending of b could be in any order, and the race condition will be avoided. This solution does not work in all cases, because in some problems, we need the events to be executed in a certain order (for instance, if the following order relation $!b <?c$ must be guaranteed) and changing the order of events will not solve the problem.

A race condition could happen in a weak while loop (see for instance Figure 3.6) [4, 31], which is called *race choice* in [27]. A weak while loop is a collaboration composed of a sub-collaboration C_1 (loop body), which is repeated zero, one or more times, followed by a collaboration C_2 (follow-up), written $C_1 *_w C_2$. The choice between C_1 and C_2 is a local choice. There is a weak sequence between the end of C_1 and the execution of C_1 again (or the follow-up). In Figure 3.6, consider the loop is executed twice (C_1 is executed twice then C_2 starts). If b (in C_1) is too slow, message c (in C_2) may arrive at role z before the last instance of b, and may result in a situation depicted in Figure 3.7, which is called a *race condition*.

The use of a sequence number, which indicates the repetition number of the loop, has been proposed in the literature to solve this problem. Two implementations have used this idea to avoid the race condition in a weak loop [4, 31]. In [4], there is a guard controlling the consumption of the messages. The message pool will only accept messages for consumption that have a sequence number parameter that satisfies the guard. We call this implementation *implementation with consumption guards* [4]. The reception pool

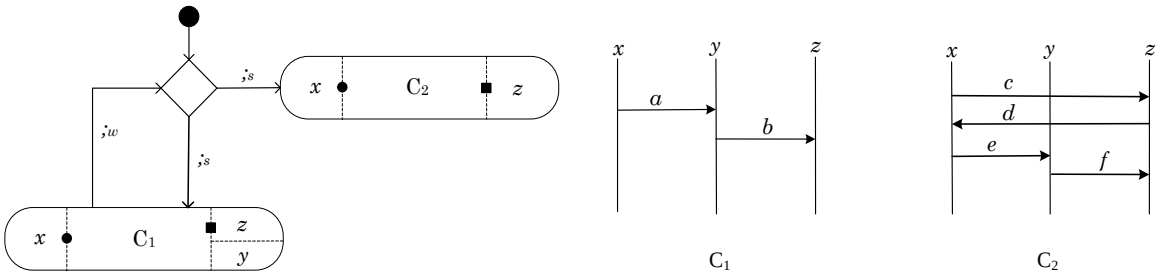


Figure 3.6: Example of race condition in a weak while loop.

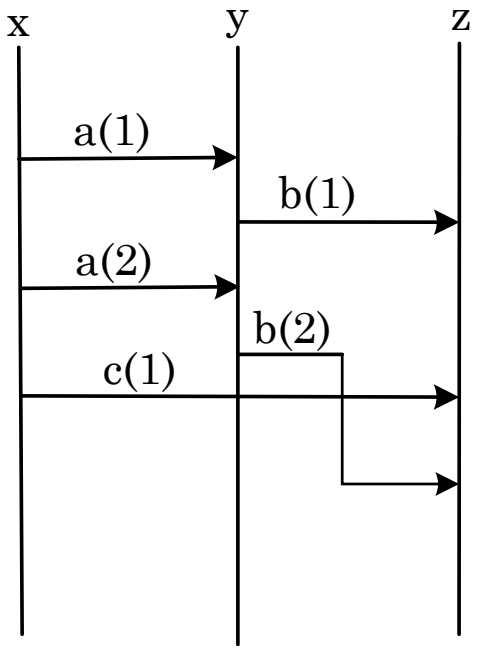


Figure 3.7: Behavior implied by Figure 3.6.

of some distributed environments such as BPEL cannot control message consumption by their parameter; messages are only distinguished by message type. An implementation in such environments is described in [31], and is called *implementation without consumption guards*. These two implementations are defined in our thesis as follows:

Definition 3.1 (Implementation with consumption guards) *This implementation assumes that the interface of the role with the message reception pool allows the role to select messages to be consumed, not only by the type of message in question, but also depending on conditions (guards) that must be satisfied by the message parameters. The role has a local variable num which is initialized to 0 before the while loop starts. Each time the first message of C_1 is received, num is incremented. When the role is at the decision point between the execution of C_1 or C_2, the role is ready to consume the first message of C_1 or the first message of C_2, but only if the following condition is satisfied: the message parameter must be equal to num+1. Then the variable num is incremented, and the actions for the role in collaboration C_1 or C_2, respectively, are performed.*

Definition 3.2 (Implementation without consumption guards) *This implementation assumes that the interface of the role with the message reception pool allows the role to select messages to be consumed only by the type of message in question. We assume for this implementation that the first messages for C_1 come through a FIFO channel and are delivered by the pool in FIFO order. Two variables num_1 and num_2 are required, which are initialized to -1. The role performs the following algorithm:*

- *When the role is at the decision point between the execution of C_1 or C_2, it is ready to consume either the first message for C_1 or C_2.*
- *If the message consumed is the first message for C_1 then its parameter is stored in num_1, and C_1 is performed, else the first message of C_2 is stored in a local buffer, and its parameter is stored in num_2.*
- *If $num_2 = num_1$, then C_2 is performed, otherwise the role goes back to the first step.*

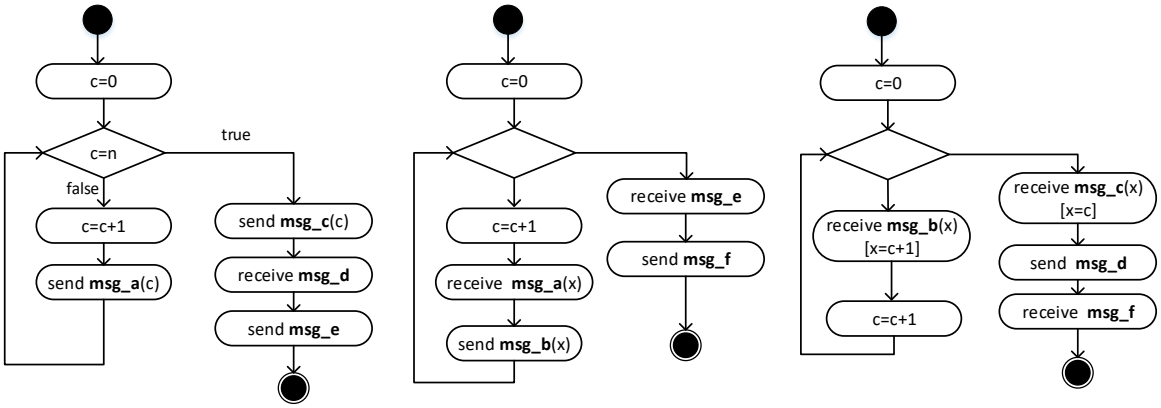


Figure 3.8: Solving race condition using *implementation with consumption guards* for the example in Figure 3.6 (the variable c here corresponds to a variable *num* in Definition 3.1).

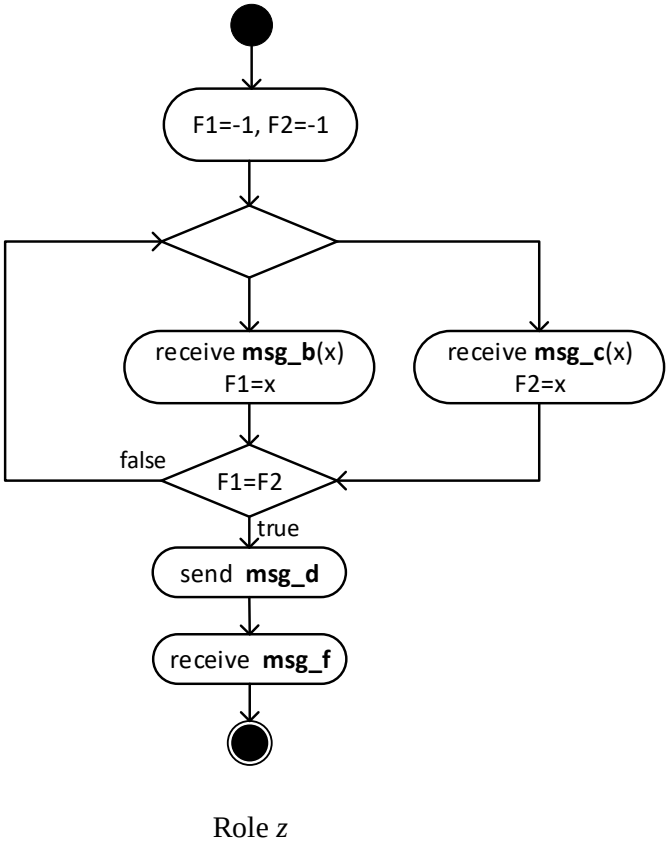


Figure 3.9: The implementation for the role z using the *implementation without consumption guard* (the variables F1 and F2 here corresponds to the variables num_1 and num_2, in Definition 3.2), respectively.

In Figure 3.8, we present the implementation for the roles of Figure 3.6, *i.e., x, y* and *z*. The race condition is avoided using the *implementation with consumption guards*.

The problem is also solved by the second solution [31] (*implementation without consumption guard*). In this case, the implementation for the roles: x and y will not change (see instance Figure 3.8). For the role z, the implementation is presented in Figure 3.9.

3.1.2 Non-local Choice

In the composition of alternatives, which is specified by means of the choice operator, the choosing roles are the ones who decide which alternative will be executed. The choice is called "local" if only one role does the decision. The choice is non-local if several roles are involved [9].

Figure 3.10 (a) shows an example of local choice where role x has the initiative in both alternatives and is called the choosing role. This is not the case in Figure 3.10 (b) where two choosing roles are involved, *i.e.*, x and z (the first actions in C_1 and C_2 executed by the roles x and z, respectively) and they decide which alternative is to be executed, therefore, it is a non-local choice. Local choice is simple since only one role decides which alternative will be chosen, while with the non-local choice, the process gets more complicated because different choosing roles may try to initiate their alternative simultaneously, which may lead to conflicts.

There are several types of non-local choice from the application point of view:

- Several components are involved in the choice, and they agree on the alternative to be selected based on some preceding negotiation or by agreeing to use a mutual exclusion protocol (e.g. using a circulating token) to determine which role is allowed to make the choice.

- Several components are involved in the choice, and there is no coordination between them. Several alternatives could be initiated concurrently. This situation is called *mixed initiatives* [2]. There are two types of mixed initiatives:

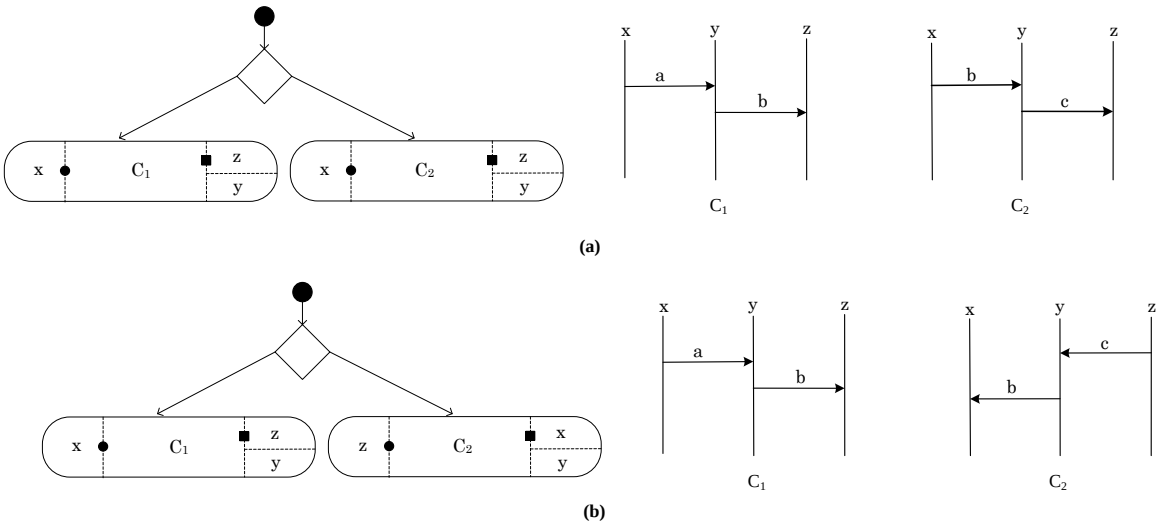


Figure 3.10: Example of alternative composition: (a) local choice, (b) non-local choice.

- The mixed initiatives are called *competing initiatives* if the alternatives have different goals, and only one of them should be selected.
- The alternatives could have the same goal, and there is no conflict between them. An example is presented in [2], where the patient and doctor roles disconnect the call simultaneously after the consultation is done. This situation is called *mixed initiatives with common goal*.

Despite many types of research that had considered the non-local choice problem, there is no consensus on how it should be treated. Some of the authors consider the elimination of non-local choice because it is a result of an "invalid" specification [9]. Some authors had considered the non-local choice as an obstacle for realizability, and they proposed a restricted version of HMSC [32]. Gouda [33] and Mooij [27] consider the non-local choice as a common and inevitable issue in the design of distributed systems. They addressed the problem at the distributed design level and proposed a generic approach for the implementation of non-local choice. Gouda solves the problem by giving priority to one component over the other. A conflict-resolution protocol called *circulating token* was also proposed to solve the non-local-choice: a token circulates between the components involved in the choice, and any component that has the token can make the decision; mutual exclusion

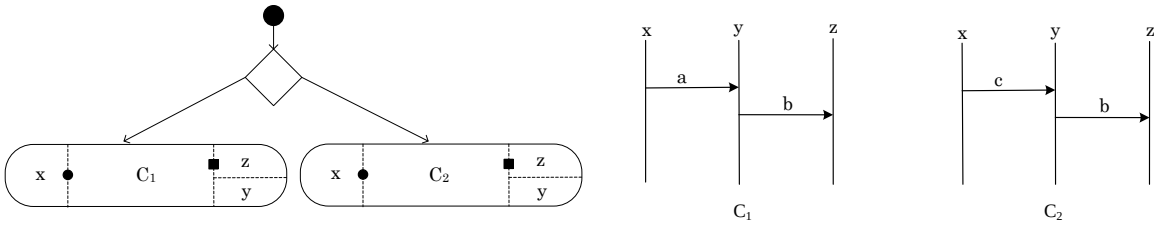


Figure 3.11: Example of choice with non-deterministic propagation.

is given to the component that has the token. The authors of [34] also applied Gouda's approach for solving the mixed-initiative problem. However, they discussed the implementation of one component only (the one with the highest priority). They did not address the implementation of the other component.

3.1.3 Choice Propagation Problem

When the choice is made by the choosing role, it should propagate correctly to the non-choosing roles in order to help them to execute the chosen alternative. In each non-choosing role, the first action is message reception (one or several messages), this is called the *triggering trace* [2]. It is the job of the triggering trace to enable the non-choosing role to determine the selected alternative. In some cases, the non-choosing role can't determine the selected alternative, which is made by the choosing role (see, for example, the role z in Figure 3.11). As we see in this figure, the first message received by role z in both alternatives is the same, *i.e.*, the reception of message b. Therefore, after the reception of message b, z can't determine which alternative is chosen (C_1 or C_2). We say that the decision of role x is ambiguously propagated to z. This is called *non-deterministic choice* [10]. Triggering traces such as $(?b, ?d)$ and $(?b, ?e)$ in Figure 3.12 cause initial conflict (ambiguous propagation), since $?b$ is received in both alternatives, but it is resolved upon the reception of the second messages: $?d$ or $?e$, in C_1 or C_2, respectively. Bochmann [13] solves the ambiguous choice by introducing different message types that indicate to which alternative the message belongs.

Consider Figure 3.13 below, if C_2 is selected in the choice, z will not know about the

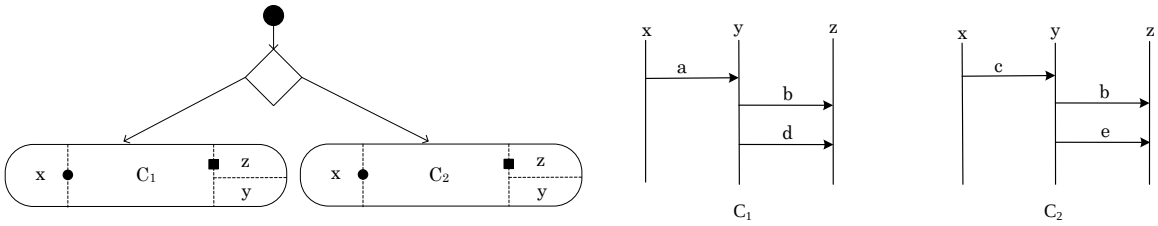


Figure 3.12: Example of choice with non-deterministic propagation solved by triggering trace.

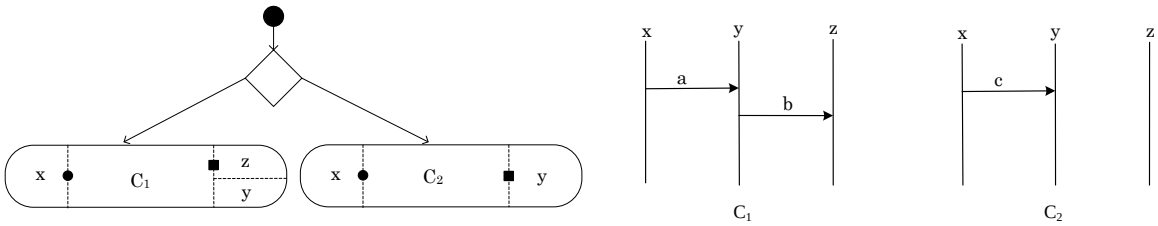


Figure 3.13: Example of choice propagation problem (a role does not participate in all alternatives).

choice since it does not participate in C_2, *i.e.*, the choice decision will not be propagated to the role z. To solve this problem, Bochmann [4] suggests the choice indication message (*cim*) which indicates the choice to those roles that do not participate in the chosen alternative, and this is necessary if z participates as an initiating role in another collaboration after the choice.

3.2 Review of Work on Realizability

We are interested in (a) a global requirements model, which describes the system behavior abstractly in terms of local actions to be performed by different system roles or messages exchange between these roles, and (b) a distributed design model that describes the behavior of each role separately, including its local actions plus the exchange of coordination messages, which are necessary to assure that the actions of the different roles are performed in the required order. The distributed design model should conform to the global requirements model. The problem of finding such a distributed design models is called

realizability. The global model is not realizable if such a design model can't exist, such as the case of implied scenarios [35] (see Section 3.2.1).

In distributed system design, we get the design model for a given role r from the global requirements by projecting its behavior onto role r, that is, deleting all actions that are associated with other roles $r' \neq r$. We call this a **basic implementation** [27, 4]. There is the option that each role has a reception pool in which the received messages are stored until their consumption [27], otherwise they would have a FIFO queue. In some cases, there is no realizability without such a pool in the design model. A global requirements model is said to be **directly realizable** if the basic implementation conforms to the global requirements [26, 2].

3.2.1 Problem of Realizability

We call the problem of deriving distributed design models from the global requirements model the *realizability* of the global requirements model (also called protocol synthesis).

The problem of realizability has been extensively studied by many authors, where different formalisms have been used for defining the global requirements. While for the definition of the local behavior of each role, normally state machine models were used.

Different notations were proposed for describing the global requirements models. Alur [1, 35, 36] used Message Sequence Chart (MSC), and Hierarchical Message Sequence Chart (HMSC), Castejón [26, 2] proposed the concept of Collaborations and Activity Diagrams (also used by [37]), PO-Charts was used in [38], finite state machine (FSM) in [39, 40, 41, 42], and recently UML state machine [43, 44, 45, 34]. Petri-Nets were used in [46, 47, 48, 49, 50, 51] for modeling the service specification. In [52, 53], they study the deviation of protocol specifications (design models) from a service specifications written in the LOTOS language (see [54]).

The conditions for the realizability of HMSC have been studied in [1, 55], for Message sequence graph (MSC-graph) in [36], and for compositional MSC [27]. Some authors have discussed the pathology's in HMSC that prevent their realization, such as non-local choice

[56].

The problem of implied scenarios (ISs) (also called emergent behavior) has been studied by many authors [35, 57, 58, 59, 60, 61]. Quite often, a single MSC can't be realized because any implementation of the MSC will automatically realize other different scenarios, called implied scenarios. Based on [57], ISs can lead to unexpected behaviours; however, they are not always unacceptable behaviors.

To understand this problem, consider the following example [35] (see, for instance, Figure 3.14) where two processes perform remote access on two variables UR and NA, which are used in the control of nuclear power system (UR and NA control the amount of nitric acid in the system). If we consider MSC_1 alone as a global model, then this model can't be implemented because the distributed state machines that realize MSC_1, would also realize MSC_2 and MSC_3, which are implied scenarios.

Alur et al. [35] propose a method to detect the ISs in a set of MSCs based on reachability analysis from the derived component models. Muccini [57] detect them by analyzing the non-local choices in the specification. Song et al. [58] propose a method to detect ISs and find the causes of such problems in the global model. Fard and Far [59] cluster the interaction between different components into different categories in order to find a point in the global model which leads to ISs. However, their approach does not find the similarity among these ISs. Finally, de Melo et al. [60] deal with all ISs at a time by finding a common behavior among them. This allows to investigate the cause of multiple ISs, and the removal of many of them with a single fix.

In [35] they consider the global specification in the form of a set of MSC that should include all implied scenarios. In this example, the amount of UR and NA should be equal to avoid a chemical accident. In the requirement, P_1 should increment both UR and NA and P_2 double them. This requirement is realized by MSC_1 in Figure 3.14. MSC_2 is a good implied scenario from MSC_1, because after both transactions have finished; the equal amounts of UR and NA is maintained. However, MSC_3 could also happen, and this is a bad implied scenario because it leads to different amounts of UR and NA. This is an example where the set of implied MSC is not a valid global model.

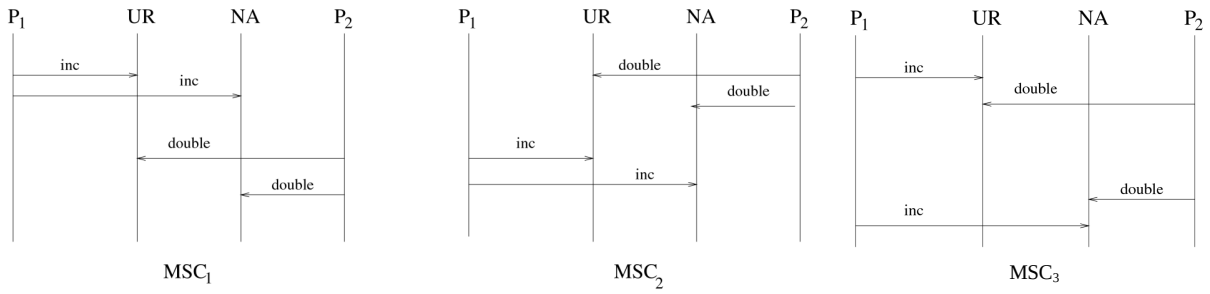


Figure 3.14: The problem of implied scenarios (taken from [35]).

The work of [35] has been extended in [36] by studying the realizability of MSC-graph, under FIFO communication. In both papers, the specification is realizable if there exist concurrent automata which implement the set of MSCs. They define two types of realizability: *weak realizability* (distributed design may deadlock) and *safe realizability* (where no additional deadlock is introduced). The behavior of each role in the distributed design model is modeled by a finite state machine communicating through FIFO queues.

Hélouët [56] studies the realizability problem of HMSC and, in particular, the non-local choices. He considers the non-local choice as a problem that leads to unspecified behaviour. An algorithm has been developed to detect them in the context of HMSC.

3.2.2 Toward a Constructive Approach for Deriving a Distributed Design Model from a Global Requirements Model

The basic implementation for the design model is a design model obtained by projection as described above (with a reception pool), and without any coordination messages. A global requirement model is directly realizable if the basic implementation conforms. In the previous section, we are concerned with a global model written in the notation of MSC and MSC-graph, which describe the behavior of the system by messages exchange. This section is concerned with the global model, which has no messages and only local actions.

If an order should be introduced between two actions $a_1 \rightarrow a_2$ associated with different roles r_1 and r_2, respectively (this case corresponds to strict sequencing), a coordination message (called "flow message" in [8, 62, 4]) should be sent by r_1 (after the execution of a_1),

and to be received by r_2 (before the execution of a_2). Using such coordination messages makes the specification not directly realizable, *i.e.*, realizable using coordination message.

Bochmann in [4] introduces the *cim* message for the realization of alternatives. The *cim* is required for a role that participates in one of the alternatives and not in the other.

The realization of a weak while loop has been studied by many authors [4, 31], they report the race condition which prevents the direct realizability of a weak while loop. Additional parameters (sequence numbers) are introduced to achieve realizability.

Mooij et al. [27] formally study under which conditions the global requirement is directly realizable. They prove that the absence of non-deterministic, race and non-local choice lead to sound choice which is directly realizable. Different composition operators (*i.e.*, weak and strict sequence, alternative and parallel) between sub-collaborations are studied in [26, 2], and how they affect realizability, *e.g.*, strict sequence between two collaboration C_1 $;_s$ C_2 is directly realizable if all terminating roles of C_1 and all initiating roles of C_2 are located at the same role, otherwise, coordination messages are needed to achieve realizable implementation (not directly realizable).

In many cases, the global requirements are not directly realizable, however, they are realizable with additional coordination messages or certain parameters. In [63], additional data is added to the sent message as a parameter to achieve the safe realizability for MSC, and in [32] for the realizability of local HMSC. In [26, 2], they present under which conditions the coordination messages are needed when strict and weak sequence are used.

3.3 Previous Work on Choreography

3.3.1 Global Description of the Requirements

The concept of using the software as a service is important where the execution of the software at a certain component can be delivered as a service through the network. Service-Oriented Computing (SOC) is a computing paradigm that utilizes the service for the building of low-cost distributed applications. It allows a free composition of independent

and autonomous peers within their distributed environments (Web Services is an example [64]). In this context, the specification of service composition can be addressed using two main approaches: *service orchestration* and *choreography*. The orchestration is a communication-centered programming paradigm, and it is useful for building applications that are coordinated by specific components (agents). A WS-BPEL (Business Process Execution Language) [65] is a language for defining such a coordination component. On the other hand, with choreography, there are direct interactions between composed services without any mediator. The meaning of the word choreography can be summarized as:

The dancer's dance following a global scenario without a single point of control

BPEL is a high-level programming language used for creating composite centralized-control web services. A BPEL program interacts with several services distributed over several servers. However, these services consider the orchestration approach, *i.e.* centralized control. In [66] they consider a technique that partitions a BPEL composite service into an equivalent set of decentralized processes in order to increase parallelism and reduce network traffic for the application. In this decentralized approach, messages can be sent from the data producer component to the data consumer component without any centralized coordinator. The resulting proposed implementation is more efficient since the number of required coordination messages is reduced.

The Web Services Choreography Description Language (WS-CDL) [67] developed by the W3C WS-CDL Working Group is a choreography-based language that describes peer-to-peer interactions of participants by defining, from a global point of view, their common and supplementary behavior. Mainly, WS-CDL is used to describe the order of messages exchanged and the local actions at each participant in the global view. After such a scenario is specified, it will be executed by individual processes without any centralized control, *i.e.*, there is no single-point control. A WS-CDL specification is platform-independent; it is targeted for composing interoperable collaborations regardless of the supporting platform. Also, it supports sessions in communication where the session is established between communication components, and this helps to distinguish the messages involved from the different executions of the same or other protocols.

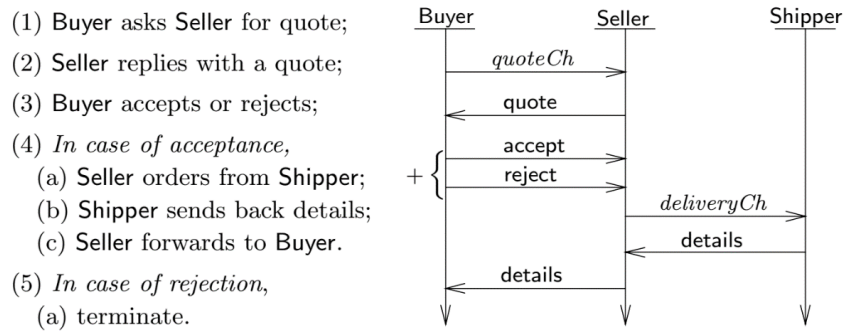


Figure 3.15: Buyer-Seller protocol (taken from [69]).

Global Calculus [68], which is originated from (W3 WS-CDL) describes the distributed participant interactions from a global point of view. Consider the "Buyer-Seller protocol" example in Figure 3.15 (this example comes from a use-case presented in the web service choreography description language (WS-CDL) description [67]), the protocol is presented with both text and a sequence diagram where (+) denotes a choice.

Global Calculus uses two software engineering concepts: *Service Channel* and *Session*. There is a channel between any two participants involved in the communication, *e.g.*, the channel is defined statically before the first communication between Buyer and Seller (or between Seller and Shipper) is established. This channel can be shared and used repeatedly. In order to prevent confusion between the concurrent sequence of transactions between participants, the session concept is used, and in one channel, there could be several sessions. The session binds the series of interactions between any two participants into one, and this helps in distinguishing them from other messages related to other sessions.

Figure 3.16 describes the Buyer-Seller example in Global Calculus. The protocol starts by defining a channel $quoteCh$ between Buyer and Seller (line 1) where the Buyer asks the Seller for a quote, $quoteCh$ is considered as public URL for a service, the session s (bound by v) is used to start the information exchange between Buyer and Seller. In line 2, the Seller replies through session s with a quote amount (300), which is stored in the variable x at the Buyer side. Line 3 in Figure 3.15 (choice between accepting or rejecting) is written in Global Calculus in the lines 3 to 8 (see Figure 3.16). The same session s is used between Buyer and Seller, then another channel $delivCh$ and session t is defined for

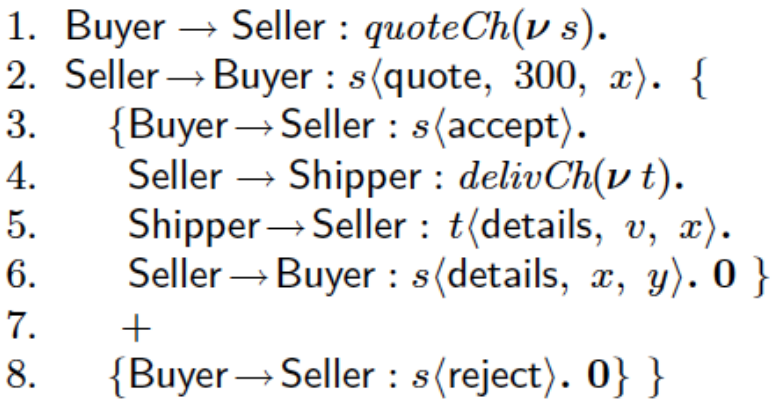


Figure 3.16: Buyer-Seller example described in Global Calculus (taken from [69]) .

the interaction between Seller and Shipper where Shipper sends the delivery details to the Seller through session t, then it is forwarded from Seller to Buyer using session s.

We note that WS-CDL assumes synchronous message exchange (atomic), *i.e.*, the message sending and receiving are assumed to occur at the same time. The problems related to race conditions and weak sequencing, as discussed in our work, do not occur in this type of communication. In the implementation, however, the "Fire and forget" send action as in asynchronous communication is often preferred over synchronous communication where handshaking should be used [70]. To this end, they report [71] how the synchronous interactions between participants that realize a choreography can be desynchronized; in other words, the atomic message exchange is broken up into asynchronous send and receive events. This procedure has a drawback because a deadlock might occur. In [70], they proposed a technique to avoid this problem during the design time, each message can individually define the transfer communication type: synchronous or asynchronous, and this helps in refining the choreography model.

In [72, 73], they study the realizability of a choreography; whether it is realizable or not, and if not, they study whether a specific reconfiguration can be applied to solve the problem. They use the Chor Calculus [74] (abstract model of WS-CDL for describing the peers from a global point of view) as a specification language for choreography. They

propose an encoding from Chor to the FSP algebra, which is supported by the LTSA toolbox. The encoding helps with verifying the choreography specification (written in Chor), generates a peer protocol from the global specification by projection, and tests the realizability of the specification. The proposed scheme is implemented by a prototype tool.

3.3.2 End-Point Projection

As we said, the Global Calculus describes the distributed interactions between participants from a global point of view, while the End-Point Projection (EPP) [75] identifies the local behavior of each participant precisely, *i.e.*, the projection of global specification on to an end-point participant, so we can get the local behaviors to realize the original global scenario [62, 4, 27, 2]. Consider the following simple global interaction between Buyer and Seller through the channel ($B2Sch$), where the Buyer accepts the quote with 100$ as price, then the Seller receives the price and stores it in a local variable x:

$$Buyer \rightarrow Seller : B2Sch(QuoteAccept, 100, x) \tag{3.1}$$

Now if we do the projection on each end-point participant, *i.e.,* Buyer and Seller, we get the following:

$$Buyer[\overline{B2Sch}(100)] \tag{3.2}$$

$$Seller[B2Sch(x)] \tag{3.3}$$

In (3.1), the sending and receiving of information are written together as one thing, but they are decomposed to (3.2) and (3.3) after projection, where the horizontal bar over a channel indicates that it is for output.

The following is the end-point projection of the example in Figure 3.16 at the Buyer and Seller, respectively:

$$Buyer[\ \overline{quoteCh}(vs)\ .\ s \triangleright quote(x).(\ \overline{s} \triangleleft accept.\ s \triangleright details(y).\ 0 \oplus \overline{s} \triangleleft reject.0)] \tag{3.4}$$

$$Seller[!quoteCh(s)\ .\ \overline{s}\ \triangleleft\ quote(300).\ s(\ \triangleright\ accept.\ \overline{deliveryCh}(vt).\ t\ \triangleright\ delivery(x).$$
$$\overline{s}\ \triangleleft\ delivery(x).\ 0 + reject.\ 0)] \quad (3.5)$$

Note that $\overline{s} \triangleleft op()$ refers to a branching output type, and $s \triangleright op()$ is a branching input type.

In this context, the authors of [75] discussed the realizability by comparing the resulting behavior obtained by projection at each participant with the original global choreography. It leads to a directly realizable design, which is similar to what we study in our work. First, they discuss the **Connectedness Principle**. Consider the following example (note that "." means sequencing (strict sequencing), $B2Sch$ stands for Buyer to Seller Channel, $Sp2Dch$ is Shipper to Depot channel):

$$Buyer \rightarrow Seller : B2Sch(s).\ Shipper \rightarrow Depot : Sp2Dch(t) \quad (3.6)$$

The code in (3.6) above is not directly realizable because the Shipper should contact depot only if the Buyer sends a request to the Seller. Consequently, a coordination message is required from Seller to Shipper. However, the following example (3.7) is directly realizable and therefore *Connected*:

$$Buyer \rightarrow Seller : B2Sch(v\ s).\ Seller \rightarrow Shipper :\ S2Spch(v\ s).$$
$$Shipper \rightarrow Depot : Sp2Dch(v\ t). \quad (3.7)$$

Well-threadedness is another principle related to causality, it should be maintained in the EPP for a directly realizable design. Consider the following example (note that this example is connected):

$$Buyer \rightarrow Seller : ch_1(v\ s).\ Seller \rightarrow Shipper : ch_2(v\ t).$$
$$Shipper \rightarrow Buyer : ch_3(v\ u).\ Buyer \rightarrow Seller : s(op, v, x). \quad (3.8)$$

As stated in [69], this example is not readable at end-point participants. In the first action, a session s is created at the Seller side and invoked by the Buyer thread. This

thread becomes inactive when ch_3 at the Buyer side is invoked. Finally, the Buyer tries to communicate with Seller using the session s opened in the initial action. After the session s is created at the Seller's server, if the session is identified by the identities of the two threads that communicate, then the identity of the thread at the Seller's server must be communicated to the Buyer so that the Buyer can talk to the Seller's thread. In this example, there is no message going back to the Buyer through the session s, so the identity of the peer thread in the server (Seller side) is not known to the Buyer. Therefore there is a problem with the last message transmission ($Buyer \rightarrow Seller : \ s(op, v, x)$) and the code above is not well-threaded. The following interaction is well-threaded:

$$Buyer^1 \rightarrow Seller^2 : ch_1(v\ s).\ Seller^2 \rightarrow Buyer^3 : ch_2(v\ t).$$
$$Buyer^3 \rightarrow Seller^2 : \ t(op_1, v_1, x).\ Seller^2 \rightarrow Buyer^1 : \ s(op_2, v_2, y). \quad (3.9)$$

3.4 Chapter Summary

This chapter presents previous work that is related to our research. We describe the main problems which are associated with distributed systems, such as *race condition*, *non-local* choice, *choice propagation problem*, and *implied scenarios*. We discuss (a) the global requirements model, which describes the system behavior abstractly in terms of local actions to be performed by different system roles or messages exchange between these roles, and (b) a distributed design model that describes the behavior of each role separately, including its local actions plus the exchange of coordination messages, which are necessary to assure that the actions of the different roles are performed in the required order.

We study *realizability*, which is the problem of finding a distributed design model that conforms to the global requirements model. Then we study the *basic implementation*, which is a design model obtained by projection (without coordination messages), where a design model is *directly realizable* if the basic implementation conforms to the global requirements model. We study the problems that prevent the direct realizability (such as the strict sequence) and the proposed solutions, and the problem which prevent the

realizability (such as the implied scenarios)

Chapter 4

Realizability of the Global Requirements Model

Note: This chapter is a slightly edited version of our conference paper [76].

In this chapter, we are concerned with the transformation from a global requirements model, which describes the behavior of a distributed system in an abstract manner by defining the local actions to be performed by the different system roles, to a distributed design model, which defines the behavior of each system role separately, including its local actions plus the exchange of coordination messages which are necessary to assure that the actions of the different roles are performed in the required order. This problem is often called "realizability of the global requirements model", and the global model is said to be "directly realizable" if a design model can be constructed without any coordination messages. Many difficulties are associated with the realizability of the global requirements model like non-local choice [9], non-deterministic choice [10], and race conditions [11]. Most of the work in this area uses Message Sequence Chart (MSC) [15] or UML interaction sequences [25] as a modeling paradigm for the global requirements model. The concept of collaborations has also been proposed for defining the requirements model [25, 26, 2, 7]. A collaboration identifies a certain number of system roles that play certain roles in the collaboration and defines a global behavior to be performed by these roles. The behavior is defined in terms of actions to be performed by the roles and a partial order that

defines constraints on the order in which these actions may be performed. Normally, the behavior of a collaboration is defined in terms of sub-collaborations that are performed in a specified order. The ordering relationships are strict or weak sequential order, alternatives, concurrency, and looping behavior. This formalism is similar to HMSC [56] and UML [16].

In [4], an algorithm is proposed to derive a distributed design model from a global requirements model with a behavior defined by sub-collaborations in sequential, alternative, concurrent or looping composition. As in [8, 62], the algorithm may introduce coordination messages for strict sequencing. It also deals with weak sequencing and introduces a choice indication (*cim*) message if one of the roles does not participate in all alternatives of a choice, and introduces sequence numbers in all the messages in the body of a loop with weak sequencing between the repetitions. The algorithm assumes that each role has a message pool where received messages are stored until they are consumed in the order required by the role. In this chapter we investigate this problem in more detail. The main contributions are as follows:

- We show that in many cases the *cim* message is not required.
- We discuss the reception of coordinating messages if an alternative with different sets of terminating roles is followed in strict sequence by another sub-collaboration; a case which was not covered in [4].
- We discuss in detail under which conditions sequence numbers in the messages of a weak while loop are required for coordination. In particular, we have a proposition which states under which condition a weak while loop is *directly realizable* (without sequence numbers nor additional coordination messages); we distinguish between message delivery with and without FIFO order; and we point out that, in general, not all messages need sequence numbers when the loop is not *directly realizable*.
- We show that the distributed design model for a weak while loop may be constructed such that for certain roles, the direct realization approach can be taken, while the approaches of [4] or of [31] may be used independently for the other roles. The approach of [4] assumes that the message pool has an interface that allows waiting

for a message with a specific sequence number. If such a function is not available, as for instance, in typical programming languages interfaces, the approach of [31] (which is more complex) can be used.

- We discuss in detail the functions that the interface of the message pool of a role should provide for the different behavior composition rules.

4.1 Definitions and Notations

4.1.1 The Concept of Collaboration

Note: the reader can skip this section since it was covered in Section 2.2.

As mentioned above, a collaboration is a collection of actions that are performed by a distributed system. In the global requirements model, a certain number of roles are identified, and each action is performed by one of these roles such that their execution satisfies a given partial order. An example is shown in Figure 4.1 (a) using a graphical notation borrowed from [7]. Collaboration A has three roles, x, y, and z, and includes 6 actions, a_i ($i = 1, 2, \ldots 6$). The partial order for the execution of these actions is shown in Figure 4.1 (b), where "$a_i \rightarrow a_j$" means that the execution of action a_i is performed before the execution of action a_j.

For the composition of two collaborations A and B in strict sequence, written "A $;_s$ B", any action of B may only be executed when all actions of A have been completed. Therefore it is important to identify the initial actions of B (those for which there are no earlier actions in the partial order) and the final actions of A (those for which there are no later actions in the partial order). As discussed in [62], for ensuring strict sequencing between A and B, it is sufficient to ensure that all initial actions of B start after all final actions of A have completed. Figure 4.1 (c) shows a more abstract view of collaboration A, showing only the initial and final actions of the collaboration. The order of execution of the actions of collaboration A is also be presented in Figure 4.1 (d) using the notation proposed

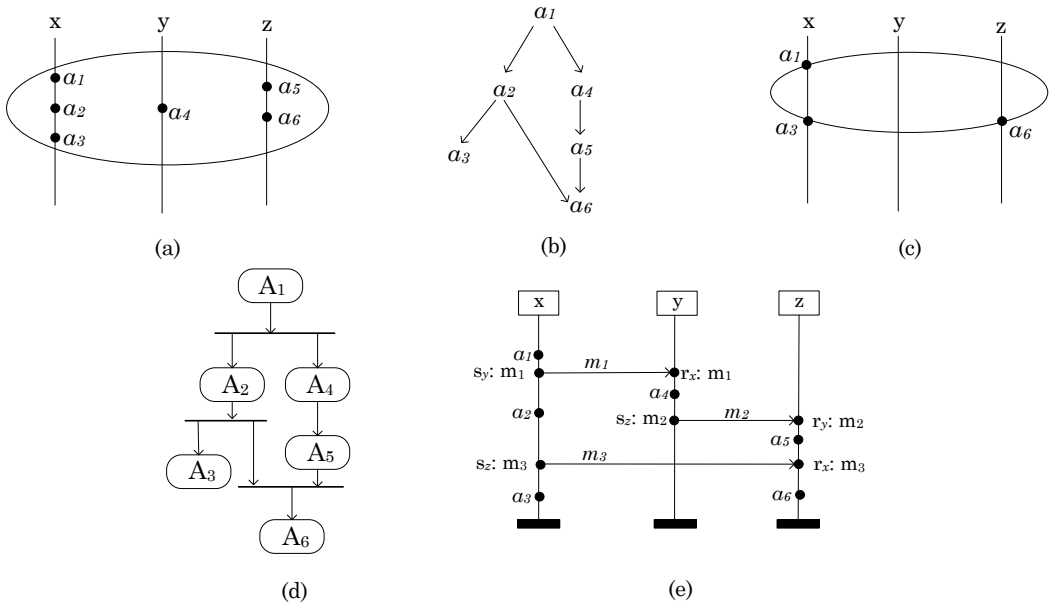


Figure 4.1: (a) Example of a collaboration A, (b) The partial order for the execution of the actions in (a), (c) An abstract view of the collaboration in (a), (d) Another representation for the order of actions executions in (a) (adapted from Activity Diagrams), (e) Distributed implementation for the collaboration in (a) using MSC notation.

in [2] (adapted from Activity Diagrams). Here the A_i ($i = 1, 2, \ldots 6$) are sub-collaborations, and each A_i contains a single action, namely a_i.

Figure 4.2 shows two other compositions of collaborations. Figure 4.2 (a) shows a collaboration including two decision points: the possible execution orders are $C_0 ;_s C_1 ;_s C_3$, $C_0 ;_s C_2 ;_s C_3$, and $C_0 ;_s C_2 ;_s C_4 ;_s C_3$, where the C_i are arbitrary sub-collaborations. Figure 4.2 (b) shows a weak while loop where sub-collaboration C_2 is performed after zero, one or more executions of sub-collaboration C_1. The sequencing between successive executions of C_1 and the final execution of C_2 are weak sequences, which means that sequencing is only enforced locally by each role, but not globally.

4.1.2 Behavior of Collaborations: A formalization

The behavior of the collaboration shown in Figure 4.1 can be modeled by a pomset $(E, \sum, <, l)$ (see Section 2.5), where $E = \{e_i \mid i = 1, 2, \ldots 6\}$, $\sum = \{a_i \mid i = 1, 2, \ldots 6\}$, $<$ is as

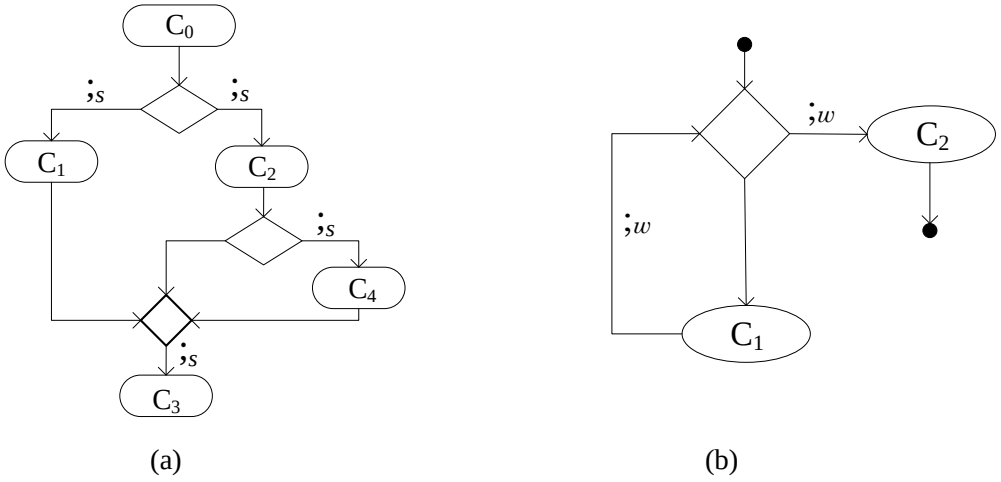


Figure 4.2: (a) An example of compositions of collaborations including decision points, (b) Weak while loop.

shown in 4.1.(b), and $l(e_i) = a_i$ for $i = 1, 2, ...6$. Note that the events in the figure are labelled with the action labels, not with the event names.

The collaboration shown in Figure 4.1 is a special case of a requirements model which has a behavior defined by a single pomset. However, in general, the behavior of a collaboration consists of several pomsets. Gischer [19] uses the term "process" to designate a behavioral model, such as a collaboration, and the term "behavior" for the set of pomsets that are allowed for the execution by that model. The rules are defined for the behavior of process (or collaboration) compositions, see Section 2.5.

As an example, the behavior of the collaboration shown in Figure 4.2(a) can be defined by the expression $C_0 \ ;_s (C_1 + C_2 \ ;_s (C_4 + 1));_s \ C_3$, where "1" represents the pomset with an empty set of events. We note that the literature on pomsets usually only considers strict sequencing, which makes abstraction of system roles. These concepts are necessary for the derivation of a distributed system design model, and for the definition of weak sequencing (which was first introduced for MSC). Therefore we introduce the concept of a collaboration-pomset, which is an extension of a pomset as follows:

Definition 4.1 (Collaboration-Pomset) *A collaboration-pomset is a tuple* $(E, \sum, <, l, \mathbb{R}, \rho)$,

where $(E, \sum, <, l)$ *is a pomset,* $\mathbb{R}$ *is a set of roles, and* ρ *is a mapping* $\rho : E \to \mathbb{R}$. *ρ assigns a role to each event.*

Definition 4.2 (Collaboration-Behavior) *A collaboration-behavior is a set of collaboration-pomsets which have a common set of roles* $\mathbb{R}$.

We consider in the following mainly collaborations that have a behavior (*i.e.* a collaboration-behavior) which can be defined by regular expressions, such as discussed above, or by diagrams, such as shown in Figures. 4.1 and 4.2.

Definition 4.3 (Weak Sequence) *The weak sequence of two collaborations* C_1 *and* C_2, *written* $C_1;_w C_2$, *has the following behavior: the set of all weak concatenations of one collaboration-pomset in the behavior of* C_1 *with one collaboration-pomset in the behavior of* C_2, *where the weak concatenation of two collaboration-pomsets* P_1 *and* P_2 *means that, for any role r, all events of* P_2 *that are assigned to the role r are after all events of* P_1 *that are assigned to r.*

Note: It was shown in [38] that associativity does not always hold for multiple strict and weak sequencing.

Like the Kleene operator for strict sequencing C^{*s} (mentioned above), we also define arbitrary, multiple weak sequencing using the notation C^{*w} . We consider in Section 4.3 the distributed design model for a weak while loop, as shown in Figure 4.2 (b), which is defined by the expression "$C_1 *_w C_2$".

4.1.3 Comparing Two Behavior Models

We use the same modeling concepts for requirement models and distributed system design models, namely collaborations (as defined in Section 4.1.2). In this subsection, we ask the question: Does a given design model C_2 conform to a given requirement model C_1? The conformance relation should be defined such that if C_2 conforms to C_1, then any implementation that conforms to C_2 will also conform to C_1.

We assume that a design model C_2 conforming to a more abstract model C_1 should include all events of C_1, each associated with the same actions and roles as in C_1, but it may contain additional events that are introduced during the refinement process (derivation of a design model). We also assume that the partial order defined by C_1 should be realized by C_2, but the order of C_2 may be stricter. Therefore we provide the following definitions.

Definition 4.4 (Conformance of Pomsets) *Given two collaboration-pomsets P_1 and P_2, we say that P_2 conforms to P_1 if (a) the events of P_2 include the events of P_1, (b) the order of P_2 is a refinement of the order of P_1 , and (c) the restriction of the labelling and role mapping functions of P_2, restricted to the events of P_1, is the same as in P_1.*

Definition 4.5 (Conformance of Collaborations) *Given two collaborations C_1 and C_2, we say that C_2 conforms to C_1 if for each collaboration-pomset P in the behavior of C_2, there is a collaboration-pomset in the behavior of C_1 to which P conforms.*

4.2 Deriving Conforming Distributed Design Models

4.2.1 Basic Ideas

Since the early work in this area [62, 27], the following basic ideas were proposed for the derivation of a distributed design model from a global requirements model:

1. The distributed design consists of processes for each role. The processes performed by different roles communicate through the exchange of messages.

2. The process of a given role r is obtained from the global requirements collaboration by projecting its behavior onto role r, that is, by deleting all events that are associated with other roles $r' \neq r$.

3. If an order should be introduced between two actions $a_1 \rightarrow a_2$ associated with different roles r_1 and r_2, respectively, one should introduce a coordination message (called "flow message" in [4]) to be sent by r_1 after the execution of a_1, and to be received and consumed by r_2 before the execution of a_2.

4. Each role has a reception pool where received messages are stored until their consumption is requested by the local behavior. We distinguish the following cases:

   (a) A single input queue which receives the messages from all other roles.

   (b) For each other role, messages are transmitted in FIFO order and stored in the pool in separate FIFO queues.

   (c) A simple pool of messages which can be requested for consumption in any order (see for instance [27]).

If we apply these principles to the global requirements model of Figure 4.1(a) and (b), we obtain the distributed design model of Figure 4.1(e), which is shown in the form of an MSC. The messages in this design are introduced according to point (3) above. For the message sending and receiving actions, we use the notation "$s_y : m_1$" and "$r_x : m_1$", where y and x are the roles to which the message is sent, and the role from where the message was received, respectively, and m_1 represents the type of the message involved.

It is important to note, that the messages m_2 and m_3 Figure 4.1(e) may lead to a race condition at reception by role z, that is, m_3 may arrive before m_2, although it is expected to arrive afterwards. A reception pool of type (b) or (c) (see above) is introduced in order to deal with such race conditions. If such a pool is used by role z, then it may consume these two messages in the order it expects, that is, first m_2 then m_3 (if m_3 arrives before m_2, it will be stored in the pool; z will wait for m_2; and then it will request the consumption of m_3). We note that a reception pool type (a) will lead to a deadlock if m_3 arrives before m_2. Therefore, this type of pool should be avoided.

One says that a global requirements model is **realizable** if a conforming distributed design model can be found. We call **basic implementation** the design model obtained by the projection approach above without any additional coordination message (using point (3)). We say that the requirements model is **directly realizable** if the basic implementation with reception pool conforms to the global requirements. We note that in the case that the global requirements are given in the form of the simple MSC without alternatives, the specification is directly realizable since it contains already all messages required for

enforcing the order of the distributed actions (for instance, if we take Figure 4.1(e) as the global requirements model).

4.2.2 Alternatives with Different Terminating Roles

For enforcing a weak sequence between two collaborations C_1 $;_w$ C_2, no coordination messages are required in the distributed design model since the ordering defined by weak sequence is a local order only. This is different for the strict sequence C_1 $;_s$ C_2, which defines globally that all actions of C_2 must be after all actions of C_1. This can be ensured by introducing coordination messages from all roles performing a final action of C_1 (called "terminating roles" of C_1) to all roles performing an initial action of C_2 (called "initiating roles" of C_2) [4, 8].

In the case of a collaboration with alternatives C_1 and C_2, followed in strict sequence by another collaboration C_3, the situation is, in general, more complex, as shown by the example of Figure 4.3. This case is not mentioned in [62], and it is excluded from the discussion in [4, 8]. If the alternatives have different sets of terminating roles, the initiating roles of C_3 have to wait for two alternative sets of coordinating messages. In the example of Figure 4.3, the choice between the two alternatives is made by role y (local choice). The role w has to consume, before the action e_6 in collaboration C_3, either two coordination messages from roles x and z (if alternative C_1 is selected), or another message from role z (if alternative C_2 is selected). We assume here that all coordination messages can be distinguished by their type.

4.2.3 Interface Provided by the Reception Pool

The reception pool should provide an interface to the local behavior at the given role, which allows specifying which messages are a candidate for consumption. For avoiding the race condition in the behavior of Figure 4.1 (e), the local behavior at role z would request the consumption of message m_2, and then of message m_3. In the case of the local behavior of role z in the behavior of Figure 4.3, the first action in both alternatives would be the

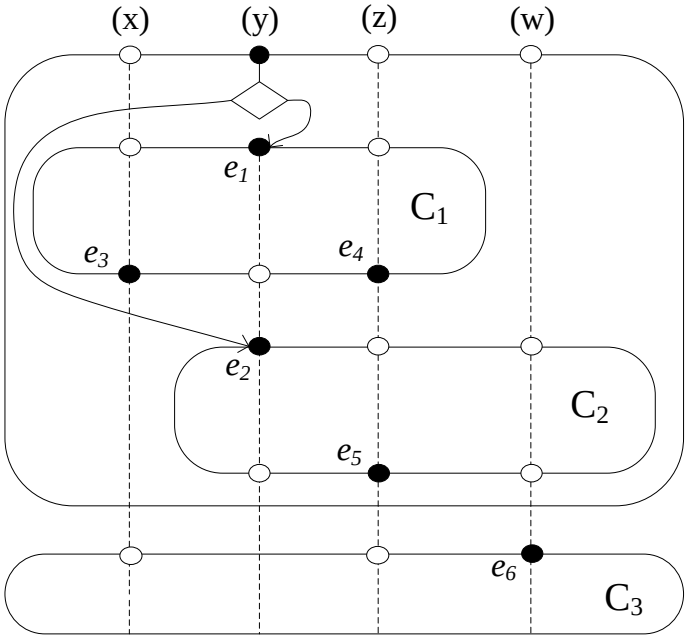


Figure 4.3: Alternatives with different terminating roles.

reception of a message (sent within collaboration C_1 or C_2). The local behavior would request the consumption of one of these messages and would be informed which message was received. We note that we assume that the messages can all be distinguished by their type (and/or by the sending role). A reception pool providing such an interface could be easily constructed using the basic Internet socket interface for communication with a single partner. Such an interface is also provided by the BPEL programming environment, which is used for writing Web Service Applications. We call this interface a "basic pool interface".

We note, however, that this basic interface is not natural for handling strict order between two collaborations. In this case, an initiating role of the second collaboration would start with requesting the consumption of a set of messages, namely all messages to be received from the terminating roles of the first collaboration. The situation becomes even more complex for strict sequence after alternatives with different sets of terminating roles, as discussed above. In this case, the initiating role after the alternative would naturally

request two or more alternative sets of messages to be consumed. In the example of Figure 4.3, the behavior of role w for sub-collaboration C_3 would start with requesting either the set of messages $\{r_x : m_1,\ r_z : m_2\}$ or $\{r_z : m_3\}$. Such an interface is unfortunately not provided for BPEL programming. Also, the interface function, which allows for requesting the consumption of a certain message type with a parameter that has a given integer value (which is useful for handling weak while loops, as discussed in the next section), is not available with BPEL.

4.2.4 A Role Does not Participate in All Alternatives

We consider here the case of choice between several alternatives, as shown in Figure 4.4, where one of the alternatives does not involve all roles, *i.e.*, alternative **A** does not involve role z. Bochmann [4] suggested the introduction of a choice indication message *cim* to indicate the choice to those roles that do not participate in the alternative. Without such a coordination message, the role z in Figure 4.4 would not know when to start the initial action of the collaboration C_3 when this alternative is chosen. We note that this problem was not mentioned in [27]. Here we would also like to point out that this *cim* message is not required if the subsequent collaboration follows in strict sequence, and in the case of weak sequence only if the role in question has initiating action in the subsequent collaboration. If in Figure 4.4 the message m_6 message would go in the opposite direction (and role z would not be initiating), role z could simply request the pool for the consumption of m_5 or m_6.

We conclude that the *cim* message is not always necessary. The *cim* is required by a role if it does not participate in one alternative, and it is an initiating role in a collaboration following the alternatives in weak sequence. In this case, it is sufficient and necessary.

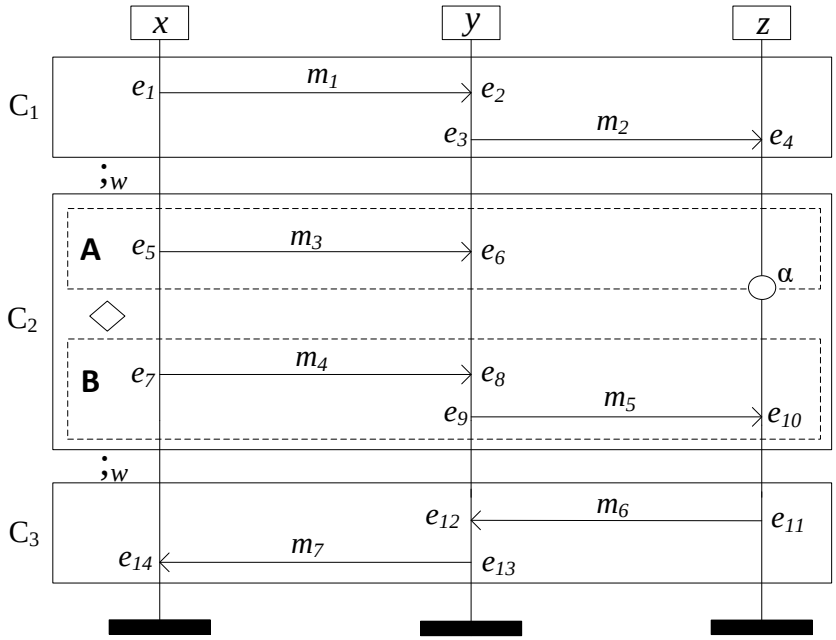


Figure 4.4: An example where role z does not participate in all alternatives.

4.3 Weak While Loop

We consider in this section a requirements model including a weak while loop as shown in Figure 4.2(b). We assume that the decision (control of the loop) of repeating collaboration C_1 (loop body) or finishing with C_2 (loop follow-up) is a local choice. In the example of Figure 4.5, this is done by role x. We call this role the "initiator" of the loop. The other roles are called "dependent roles". It is important to note that the first event of a dependent role in C_1 or in C_2 is always a reception of a message (since otherwise the role would be initiating, and the choice would not be local).

It has been pointed out in the literature that a race may occur for a dependent role between the reception of the first messages of C_1 and C_2. For instance in Figure 4.5, if the transmission of m_2 during the last repetition of C_1 is delayed for some reason, the role z may receive m_5 before the last message m_2. We call such a race a **termination race** of the loop.

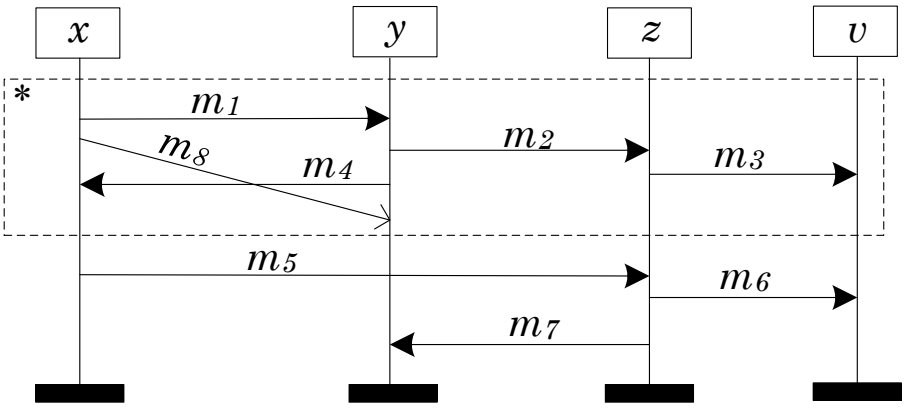


Figure 4.5: An example of weak while loop where the role x is the loop initiator.

Another problem that may occur is the following: If a given type of message of C_1 is not transmitted over a FIFO channel from the sending role to the receiving role, then it may happen that the message instance of the n^{th} repetition of C_1 is overtaken by the instance of the next repetition. If the message has no parameters, then there is no problem, but otherwise the parameter values would arrive out of order. We call this problem **message overtaking**. For instance, we note that message type m_8 in Figure 4.5 may have message overtaking in the case that C_1 is repeated twice and the transmission of the first message m_8 is very slow.

Proposition 4.3.1 *If there is a realization for collaborations C_1 (loop body) and C_2 (loop follow-up) of the weak while loop, then the control (decision) of this loop is directly realizable if there is no termination race.*

Proof. The absence of termination race means: For any dependent role r that receives the first messages m_1 and m_2 in C_1 and C_2, respectively, it can never happen that the message m_2 is received by r while some message m_1 is still in transit. Also, the absence of message overtaking means: For any dependent role r and any message type m received within C_1, it can never happen that a message of type m is overtaken by another instance

of that type belonging to the next repetition of C_1. There is no need for any coordination messages since the messages are consumed in the right order. Therefore the weak while loop is directly realizable. □

In the following, we discuss under what conditions there are no such problems and how a given role can be implemented in the distributed design model if there are problems.

4.3.1 Absence of Termination Race and Message Overtaking

We analyze in this subsection the requirements model and give some propositions that ensure the absence of termination race and message overtaking.

Proposition 4.3.2 (Absence of termination race) *A dependent role r of a weak while loop has no termination race if one of the following conditions is satisfied:*

1. *The reception of the first message m by r in C_1 is before the last event in C_1 of the initiator.*
2. *The first messages received by r in C_1 and C_2 are sent by the same role r', and the communication is over a FIFO channel.*

Proof. **For (1)**: If the first message m in C_1 to be received by r is in transit, then it must have been sent and not yet received. Since the reception is before the last event in C_1 of the initiator, the initiator must be involved in the execution of C_1 when a message m is in transit. Therefore, the first messages received by r in C_1 and in C_2 can never be in transit at the same time. Therefore there cannot be a race. **For (2)**: Since we assume that the sending role r' has no termination race, the first messages in C_1 will be sent before the first message in C_2. Because they are sent over a FIFO channel, they will also be received in this order. □.

Proposition 4.3.3 (Absence of message overtaking) *For the reception of a message type m by a role r during the repetitions of C_1, there is no danger of message overtaking, if one of the following conditions is satisfied:*

1. *The reception of the message is before the last event in C_1 of the initiator.*

2. *The message is received over a FIFO channel from the sending role.*

3. *The receiving role is the initiator of the loop.*

Proof. The proof for **(1)** is similar to the previous proposition. Point **(2)** is evident. Point **(3)** follows from the fact that the initiator waits for receiving all messages related to one repetition of C_1 before it starts another repetition. □

4.3.2 Deriving a Distributed Design Model for a Weak While Loop

To get a conforming distributed design model in the case of a termination race, it was suggested in the literature to include in the first messages received by a role in C_1 and in C_2 a sequence parameter which indicates the number of repetitions of C_1, and accept the first message of C_2 only if it contains the right sequence number. This approach was also used in [4], however, for all roles and with sequence parameters in all messages. This, by the way, also solves the message overtaking problem. We should mention here that adding a sequence number to message leads to a different message type.

In this section, we discuss how a conforming design model can be constructed by introducing a minimum number of sequence parameters in messages.

As pointed out by Proposition 4.3.1, the basic implementation for the control of the loop (loop decision) provides a conforming design model if there are no problems of termination race nor message overtaking. We propose to construct a conforming design model by starting out with the basic implementation, and then performing modifications to the behavior of those roles that have any of these problems.

For a role r that has termination race (TR), one of the following modifications can be used:

- **Modification-TR-1:** In this modification, we apply *Implementation with consumption guards* which was explained in Definition 3.1.

  This modification corresponds to what is proposed in [4]. It presents the difficulty that a message pool providing a suitable interface must be used. The message pool will only accept messages for consumption that have a sequence number parameter that satisfies the guard. To avoid this difficulty, Modification-TR-2 was proposed.

- **Modification-TR-2:** In this modifications, we apply *Implementation without consumption guards* which was explained in Definition 3.2 and in [31].

We should mention here that if there is termination race at certain role r. Using sequence number as a parameter is sufficient and necessary only for the first messages of r in C_1 (loop body) and C_2 (loop follow-up), respectively.

For a role r that has message overtaking (MO) for a message type m, the following modifications can be used:

- **Modification-MO-1:**
  - A sequence parameter is introduced into the message that has the problem of overtaking.
  - The behavior has a local variable N which is initialized to 0 before the while loop starts. Each time the message m is received, N is incremented.
  - In the request for consumption to the message pool of message m, a condition is added to the consumption, namely that the parameter value is equal to N+1.
  - This assumes that the behavior of the sending role also has a modification introducing a local variable N (which is incremented) and sending the value of N as message parameter.

- **Modification-MO-2:** Ensure that the message m is received through a FIFO channel.

Modification-MO-1 has the disadvantage that an additional message parameter must be introduced, and the message pool needs to support consumption requests with parameter conditions. It is often much easier to ensure FIFO delivery between the sending and receiving roles.

Proposition 4.3.4 (Conforming design model) *Given a requirements model R and a distributed design model D. D conforms to R if the following condition is satisfied: D is obtained from the basic implementation of R by applying the following modifications:*

1. *For each reception role that has a termination race according to R, apply Modification-TR-1 or Modification-TR-2.*
2. *For each reception by some depending role of some message with the problem of message overtaking according to R, apply Modification-MO-1 or Modification-MO-2.*

Proof. We have to show that the following conditions are satisfied:

- (a) Collaboration C_1 is executed by all roles the same number of times.
- (b) During the N^{th} execution of C_1 by a given role r, each message m consumed by r was sent by the sending role r' during its N^{th} execution of C_1.

Condition (b) follows from point (2) of the Proposition. It is straightforward to prove that the Modification-MO-1 or Modification-MO-2 assure that there is no message overtaking in the design model. For proving Condition (a), we have to prove that if Modification-TR-1 or Modification-TR-2 are introduced for a role r, this ensures that C_2 is executed by r only after C_1 has been executed N times, where N is the number of time that the loop initiator executed C_1.

For this purpose, we group the roles into role-sets RS(i) ($i = 0, 1, 2, ...$). The initiator is in RS(0), and any dependent role that receives the first message of C_2 from a role in RS(i) is in RS(i+1). Now we do the proof by induction over i. Suppose that the roles in RS(i) execute C_1 the same number of times N as the initiator, then any role in RS(i+1) receives

the first message of C_2 with the parameter N. It is easy to see that Modification-TR-1 or Modification-TR-2 ensure that the role will also be executed in C_1 N times before it executes C_2. When it executes C_2 and sends an initial message to another role, then this message will also include the parameter N. We conclude that all first messages of C_2 will include the same parameter value, and therefore all roles will execute C_1 the same number of times. □

4.3.3 Nested Weak While Loops

Note: this section was not included in [76].

Figure 4.6 shows the hierarchically nested weak while loops. In this example, *loop a* represents the outer loop, and *loop b* is the inner loop. As we said in Section 4.3.2, a sequence number is needed as a parameter in the messages to solve the *termination race* and *message overtaking* problems.

The question is whether we can use a single sequence number in the case of nested weak while loops? Figure 4.7 shows a possible execution of the weak while loops in Figure 4.6 and the messages received at the different roles. The figure shows that two sequence numbers are used to solve the termination race problem, one for the outer loop and one for the inner loop. As we see from the figure, some instances of the message m_1 are slow and they are received at r_3 after the messages m_2 and m_3. Without using the sequence numbers as parameters in the received messages, the role r_3 does not know how many times (and in which order) the message m_1 must be received before the messages m_2 and m_3 can be processed (consumed). Therefore, two sequence numbers are required in this example. In general, for hierarchical weak while loops, each loop requires its own sequence number.

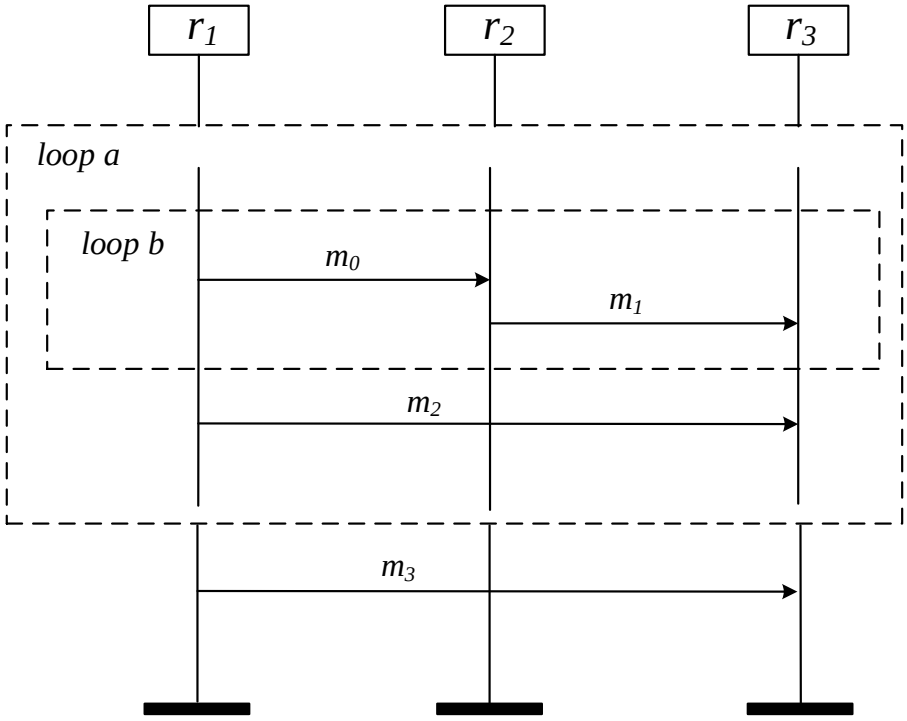


Figure 4.6: Hierarchical weak while loop example.

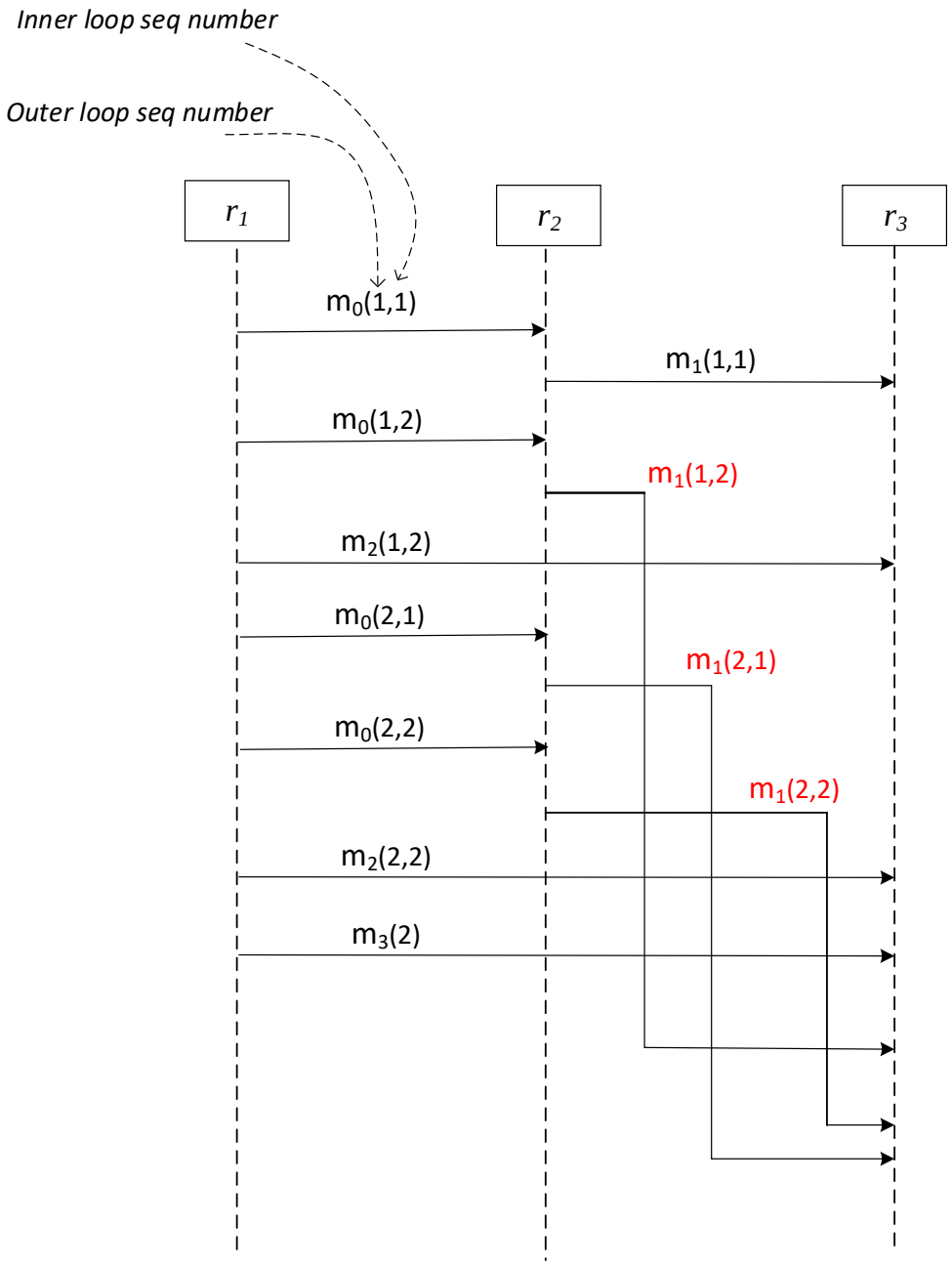


Figure 4.7: Weak while loop execution and messages reception at the different roles.

4.4 Chapter Summary

This chapter is a slightly edited version of our conference paper in [76]. In this chapter, we consider the derivation of a distributed design model from a global requirements model, which identifies the different actions to be performed by the different system roles and a partial order that determines the order in which these actions may be performed. The distributed design model defines for each role the local actions to be performed and their order. We call basic implementation a design model obtained by projection of the requirements model onto each role. If this design model conforms to the requirements, we say that the requirements are directly realizable. However, in most cases, additional coordination messages or parameters must be introduced to coordinate the order of actions at different roles. We study special cases of alternatives followed by strict or weak sequence. We show that the choice indication message *cim*, introduced in [4] is not required in many cases.

We also study the implementation of the weak while loop, which may have the problems of termination race and message overtaking. We show under which conditions these problems are absent. For the other cases, we show how a conforming design model can be obtained by introducing minimal changes to the basic implementation. Overall, this is an important improvement over what is proposed in [4].

Finally, we discuss the hierarchically nested weak while loops. We show what kind of sequence numbers are required in such a problem.

Chapter 5

A Role Based Modeling Paradigm Based on HSM

Note: This chapter is a slightly edited version of our conference paper [77].

Various kinds of modeling notations can be used during the development process of distributed systems. In this thesis, we are concerned with the transformation from a global requirements model, which describes the system behavior abstractly (defined in terms of local actions to be performed by different system roles), to a distributed design model which describes the behavior of each role separately.

In this chapter, we propose to use the concept of UML Hierarchical State Machine (HSM) [16] for modeling the requirements model since it is a well-known notation. In many aspects, the HSM formalism is similar to HMSC and PO-Charts. A composed state defines a partial order of actions, very similar to the notation of partial orders of Pratt [14] and Gischer [19]. We introduce certain extensions to HSM in order to describe the different roles that may be involved in the actions performed within one (composed) state, and for distinguishing between strict and weak sequencing [78].

For deriving a distributed design model from a global requirements model, an algorithm was described in [4] which assumes that the requirements are given in the form of a collaboration which consists of several sub-collaborations that are ordered by strict or weak

sequence, alternatives, concurrency, or strict or weak while loop. The algorithm introduces flow messages (if required) for the coordination of actions related by a strict sequence as proposed in [8, 62]. It also introduces a choice indication message (*cim*) to inform a role that does not participate in some alternative when this alternative is chosen. In order to solve a problem of race conditions during the termination of a weak loop, it introduces an additional parameter in all messages of the loop body containing a sequence number which indicates the number of times the loop has been executed. We have shown in [76] that in many cases the loop does not exhibit any termination race, and, in addition, the sequence number is not needed in all messages within the loop when a termination race exists.

In this chapter, we follow these ideas and show how a distributed design model can be derived from a global requirements model written in our notation of extended HSM. The distributed design model obtained by our algorithm contains, for each identified system role, a HSM which contains the local actions of that role identified in the global requirements and the sending and reception of the coordination messages generated by our derivation algorithm. These local HSM can be easily implemented by any suitable tool that generates implementation code from UML HSM.

We also show the general condition for the absence of a termination race in weak loops that can be checked when the global requirements are given in the form of a HSM.

5.1 Describing Distributed Systems in a Global View

Different concepts have been proposed for describing the behavior of a distributed system in a global view, which is sometimes referred to as *choreography*, in contrast to *orchestration* which represents a centralized view of the behavior [79].

5.1.1 Review of Notations for Describing Global Requirements

Different notations were proposed for describing the global requirements models, such as Message Sequence Chart (MSC) and Hierarchical (also called High-Level) Message Se-

quence Chart (HMSC), Partial Order Charts (PO-charts), Activity Diagram, and other (See Chapter 2 and Section 3.2.1).

Figure 5.1 shows a weak while loop example written in the following notations: (a) MSC, (b) MSC-Graph, and (c) the equivalent PO-charts. In this thesis, we extend the notation of PO-Charts by indicating which role does a choice (see for instance the role r_1 in the choice state in Figure 5.1 (c)). This is useful when comparing PO-Charts with our proposed notation of HSM (see Section 5.1.2).

5.1.2 Using Hierarchical State Machine (HSM) for Describing the Global Requirements

The state machine concept is a powerful modeling formalism for describing the behavior of a part of a system [16]. Hierarchical state machine enable the modeling of complex system behavior concisely; each state is either a simple state or a composite state, which represents another state machine. Hierarchical state machine formally define the syntax and semantics of concurrency and nesting concepts, which are not allowed by simple state machine [80]. This feature reduces the complexity of the state machine diagram since it allows for hierarchical nested states and the modeling of more expressive diagrams [81].

In our proposed model, we assume that a local action (as introduced above) is executed as a "do-action" of a simple state, and we extend the UML hierarchical state machine notation in order to define the role that performs the do-action of a simple state. Therefore a simple state is associated with a single role. This allows us to calculate, for a composite state, the set of initiating, terminating and participating roles (which is important for the derivation of a distributed design, as explained in Section 5.2). This extension of adding roles to the states make it possible to derive a distributed design model from the given requirements.

We note that there is a preliminary step in the development of the global requirements model which consists of selecting the roles and their major collaborations, possibly using UML Collaboration Diagrams. These collaborations may evolve into composed states in

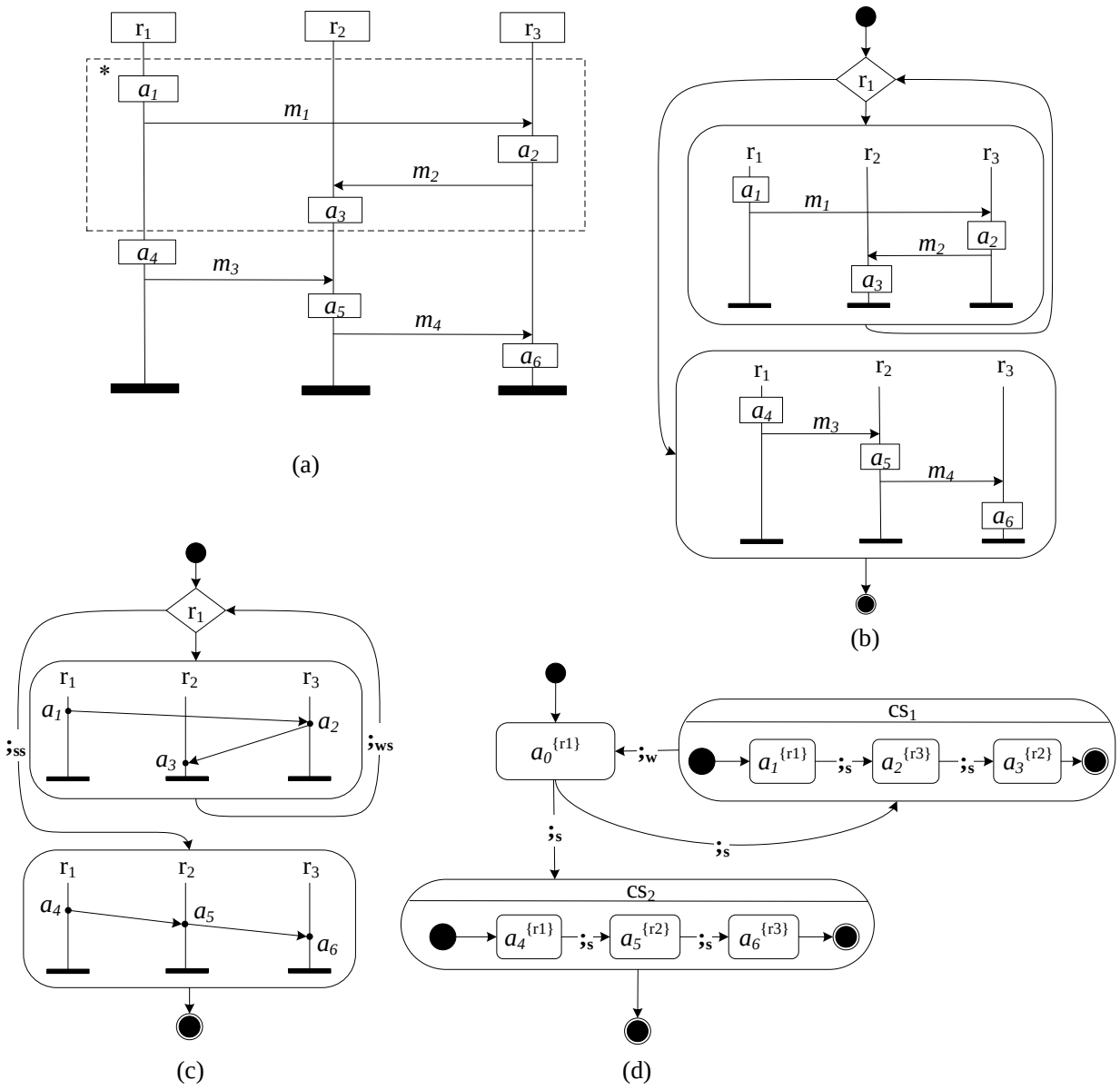


Figure 5.1: Example of a weak while loop specification written in the following notations: (a) MSC (b) MSC-Graphs (c) PO-Charts (d) Hierarchical State Machine

the final global model.

Figure 5.1(d) is an example of a weak while loop written in HSM, which is equivalent to the PO-Chart in Figure 5.1(c). In our notation, we assume that the *do-action* in a simple state (if it exists) is the same as the state name. For example, the event a_1 in the PO-chart is realized as a do-action a_1 in the HSM (Figure 5.1(d)). We use a notation where the role of a simple state is between curly brackets. A composite state could have multiple initiating, participating, and terminating roles (written as IR, PR and TR, respectively). The transitions between states are "completion transitions" (in the sense of UML) which means they are executed as soon as the actions of the state have been completed. They represent either strict or weak sequencing, written as "$;_s$" or "$;_w$", respectively.

Figure 5.2 shows global specifications for different behaviors written in the HSM notation. Figure 5.2(a) shows an example of a weak sequence between two simple states: a_1 and a_3, which have r_1 and r_3 as participating roles, respectively. The composite state "cs_1" in Figure 5.2(b) has a weak sequence type, and it has IR=$\{r_1, r_2, r_3\}$ and the TR=$\{r_2, r_3\}$. Note that the initiating (IR), terminating (TR), and participating (PR) roles of composite states are calculated based on the rules presented in Figure 5.3 (taken from Table 2 in [4]). The symbols "$;_s$" and "$;_w$" refer to a strict and weak sequences, respectively, as we mentioned above. The symbol "[ ]" refers to the choice operator, "$*_s$" to the strict while loop, "$*_w$" to the weak while loop, and "||" to concurrency.

We prefer the notation of extended HSM semantics for modeling the global system requirement because the standard HSM are well-known and the derived distributed design model consist, for each system role, of a standard HSM model which is relatively easy to implement.

In fact, our notation is similar to Gischer's work [19]; he proposed a state diagram to model the behavior of collaborations that consist of several partial orders. He used the term "process" which corresponds to a composed state in our notation. He used the same sequencing operators as in our notation, except the weak sequence. Gischer's notation does not show the roles involved in a process. Also, its alternatives are well-structured (see [82]) which is not the case for the branching structure of state machine.

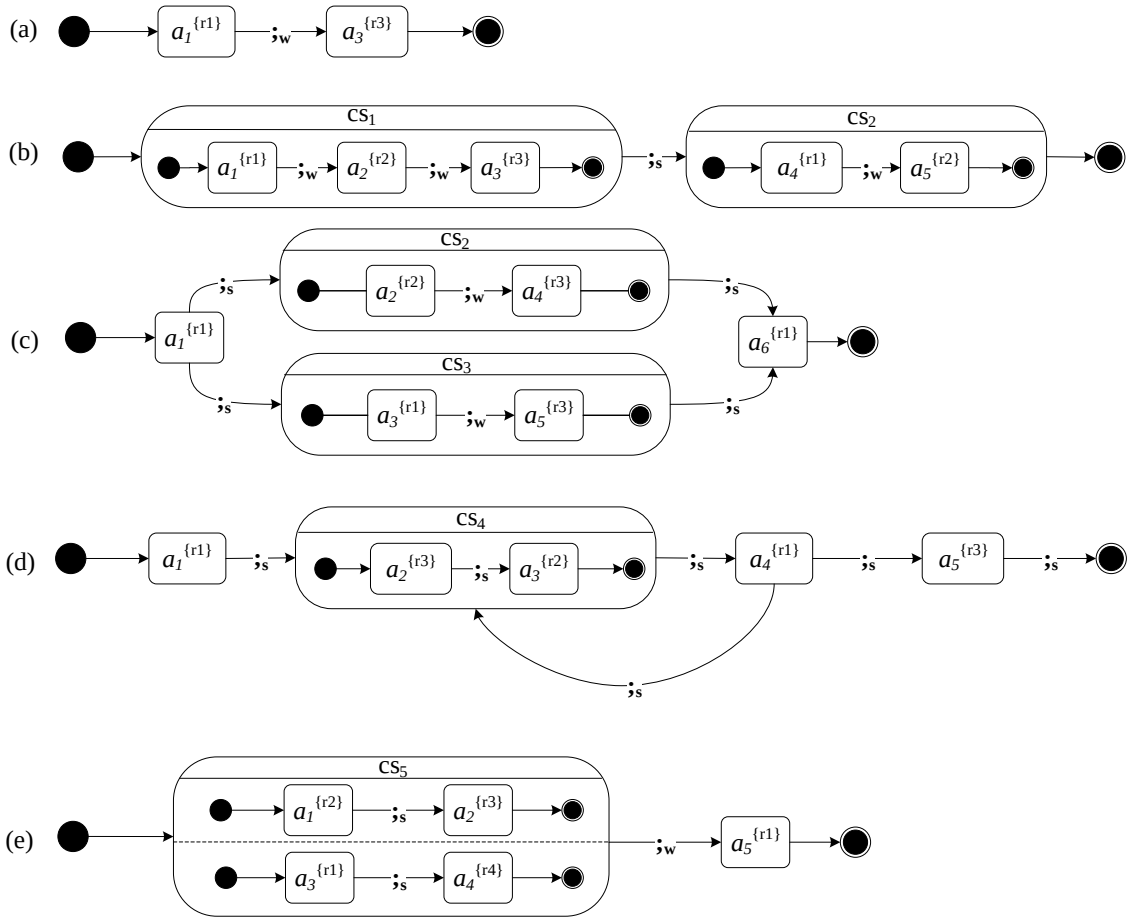


Figure 5.2: Global specification for different behavior expressions written in HSM notation: (a) Weak Sequence composition (b) Strict Sequence (c) Alternatives with Strict Sequence (d) Strict sequence with loop (e) Concurrency

Operator	Starting roles (SR)	Terminating roles (TR)	Participating roles (PR)
<action>$^{(r)}$	{r}	{r}	{r}
<subcol>$^{(R)}$	SR(<name>)	TR(<name>)	PR(<name>) = R
C_1 ;$_s$ C_2	SR(C_1)	TR(C_2)	PR(C_1) U PR(C_2)
C_1 ;$_w$ C_2	SR(C_1) U (SR(C_2) - PR(C_1))	TR(C_2) U (TR(C_1) - PR(C_2))	PR(C_1) U PR(C_2)
C_1 [] C_2	SR(C_1) U SR(C_2)	TR(C_1) U TR(C_2)	PR(C_1) U PR(C_2)
C_1 *$_s$ C_2	SR(C_1) = SR(C_2)= {r}	TR(C_2); SR(C_1) if C_2= ε	PR(C_1) U PR(C_2)
C_1 *$_w$ C_2	*as above*	TR(C_2) U (TR(C_1) - PR(C_2))	PR(C_1) U PR(C_2)
C_1 || C_2	SR(C_1) U SR(C_2)	TR(C_1) U TR(C_2)	PR(C_1) U PR(C_2)

Figure 5.3: Rules for calculating the initiating, terminating, and participating roles (taken from [4]]).

5.1.3 Example of Using the Notation of Extended HSM

The example in Figure 5.4 shows a requirements model of a taxi application. It defines the global behavior of a taxi session involving the Manager {M}, a Client {C}, and Taxi {T}. For all simple states, we show the role involved in curly bracket (as we discussed above). The taxi application involves a single manager who deals with many clients and many taxis. He establishes a session with a client and a suitable taxi during the *Assign* collaboration. The figure shows one of these sessions. The client can request a taxi by activating the *Request Taxi* collaboration or pick up a taxi on the street using the *Pick-Up* collaboration. The client may cancel a request using the *Withdraw* collaboration. When the client reaches his destination, the taxi session is terminated through the *Goodbye* collaboration, and the taxi becomes free for another session.

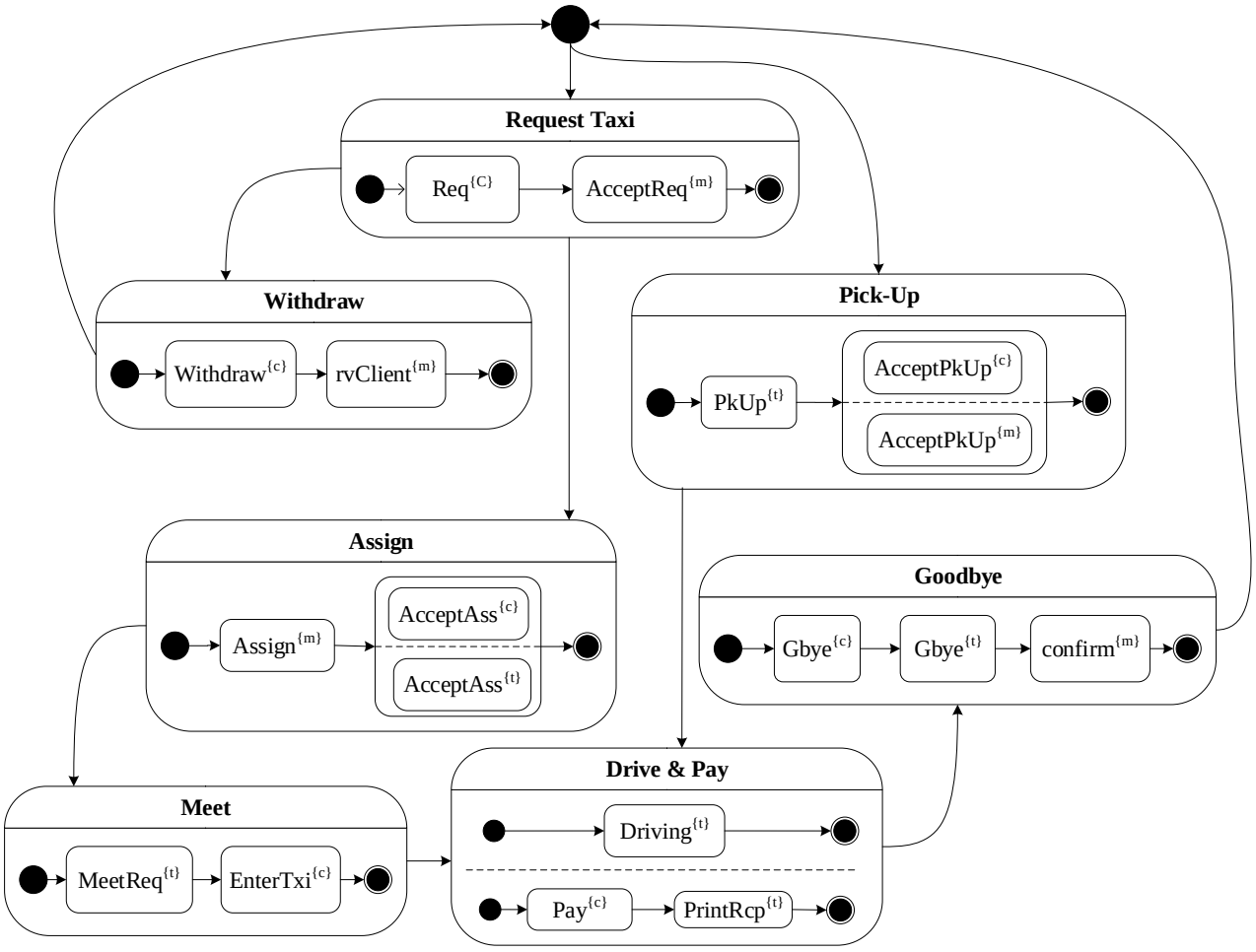


Figure 5.4: Taxi example written in extended HSM notation and involving three roles: Manager {M}, Client {C}, and Taxi {T}.

5.2 Deriving Distributed Design Models

In this section, we discuss the implementation algorithm for deriving a distributed design model from a global requirements model written in HSM notation.

5.2.1 Structure of the Global Requirements Model

We make the following assumptions about the structure of the global requirements model:

1. A simple state is associated with a single role and contains a do-action that is performed by that role.
2. A composite state has one of the following forms:
   (a) A linear sequence of (possibly composed) states **weakly** sequenced. See the example in Figure 5.2(a).
   (b) A collection of (possibly composed) states connected by **strict** sequencing. The connection structure may contain alternative branching and loop, and there are single initial and final states. This strict composite state could be one of the following:
      - A linear sequence of states without a choice, see Figure 5.2(b).
      - A linear sequence of states with a choice. A state that has several outgoing transitions is a choice state and must be a simple state. In order to get one final state, we have the following scenarios:
        – All alternatives in a given a choice must end together and there is no loop as in Figure 5.2(c).
        – The alternatives do not end together where one of the them goes back to the choice state. This case called *strict while loop*. Figure 5.1(d) is an example (but with a strict sequence between the states cs_1 and a_0).
        – It is also possible that one of the alternatives goes higher up (above the choice state). This case called *do-while loop*, see Figure 5.2(d). Note that the possibility of loops was not included in our paper [77].

In the partial order notation for local choice (see Chapter 4), we considered that the initial actions of the alternatives are executed by the role that does the choice. In the state machine notation for local choice, we consider that the initial actions of the alternatives are integrated into the choice state. Therefore, in the HSM notation, the first state of an alternative could be a simple state associated with a different role or a composite state with several initiating roles.

(c) A **weak** while loop where the body and follow-up are single (possibly composed) states, and the transition from the body leading back to the initial choice state is weakly sequenced. See the example in Figure 5.1(d).

(d) Several concurrent (possibly composed) states. See the example in Figure 5.2(e).

Comments:

- We do not allow for a linear sequence of (composed) states with mixed strict and weak sequencing, since strict and weak sequencing are not associative [38] and such a sequence could have an ambiguous meaning.

- We assume that all choices are local, that is, are performed by a single role (in Chapter 6, we discuss the non-local choice). Choices are associated with strict sequencing to ensure consistent choice propagation [83].

- The important difference of the weak while loop, as compared with loops with strict sequencing, is the fact that it may have a termination race [76] (see also Section 5.2.7.1) and that it may lead to an unbounded number of messages in transit [1].

- The *cim* message is not required in the alternative which loops back to the choice state (the case of *strict while loop*) or a higher state (the case of *do-while loop*). This is an improvement compared to our work in [77].

5.2.2 Algorithm for Deriving a Design Model from HSM Requirements

In the following, we define an algorithm that derives a design model for a given role r_n from a requirement model which is represented by the function $genDM(StateMachine\ sm, role\ r_n)$. The inputs for the function are an HSM "sm" describing the global requirement model, and the target role r_n. The algorithm returns a design model for that role.

The algorithm distinguishes the cases 1 through $2d$ above (see Section 5.2.1) and invokes for each case a specific operation which is explained in the following subsections. The function "genDM" is first applied to the whole global requirements model and is then recursively applied to all its composed states and sub-states.

For the generation of a design model for a given role from the global requirements, a technique of projection has often be used [62, 27, 4, 2]. Different notations have been used for the requirements and the local design model. Using projection, the design model for a role r is obtained by projecting the global requirement model onto role r, that means, deleting all events (states) associated with other roles $r' \neq r$. A design model obtained by projection without any coordination messages is called **basic implementation** [76].

In the presence of weak sequencing, this projection approach is usually combined with the use of a message reception pool which allows the role to wait for the reception of specific types of messages [27, 4], thus avoiding *race conditions* [11] because the other types of messages will be stored in the reception pool until they are (later) requested by the local implementation.

In the following sections, we discuss in details, the operations of "genDM" in each of the cases above.

5.2.3 Simple State

In the case of simple states, the algorithm keeps the state as is if the role r_n participates in that state, and otherwise replaces it by a dummy state. There is no recursive call for genDM in the case of the simple state.

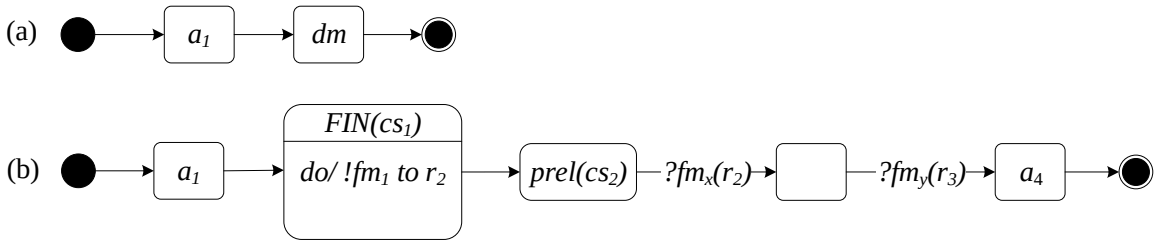


Figure 5.5: Local design models for r_1 derived from the requirement models: (a) weak sequence in Figure 5.2(a), (b) strict sequence in Figure 5.2(b)

5.2.4 Weak Sequence

The distributed design model for weakly sequenced behavior does not need any coordination messages and is only based on the local sequencing by each role (see, for instance, Figure 5.2(a)). Therefore the basic implementation provides a correct design model.

The function "genDM" in the case of weak sequence will, therefore, operate as follows. It will generate a state sequence corresponding to the states in *sm* by recursively applying "genDM" to the states in which the role r participates and replacing the other states by dummy states (denoted *dm*, without any local actions). The resulting state machine will be the design model for r_n, and it will have the same state structure as in the global design. Figure 5.5(a) is the generated design model for r_1 in Figure 5.2(a). Before generating an implementation from the design model, we may optimize the resulting model by deleting the dummy states.

5.2.5 Strict Sequence and Alternatives

The problem of deriving a local design model from a global behavior expression that contains a strict sequence was discussed in [13, 8, 4]. Coordination messages (called flow messages (fm) in [4]) were introduced for a strict sequence (C_1 ;$_s$ C_2) to ensure that all actions in C_1 are completed before any actions in C_2 can start. Besides, since these flow messages must be distinguished to avoid ambiguities, a parameter x is added to each flow messages (fm_x) to differentiate between them. The derivation of a local design model

from a global requirements model containing alternatives has been studied by many authors [8, 62, 4, 27]. In all these references, the local choice is assumed, *i.e.*, the choice decision is always made by a single role. In a non-local choice, several roles are involved in a choice [9]. Different algorithms were proposed for solving non-local choice in a distributed environment, e.g., a circulating token. Different forms of competing initiatives were discussed in [2]. Gouda proposed to give different priorities to the different initiatives to solve this problem [33]. In the following, we assume that all choices are local. Note that for non-local choice, certain proposals could be easily integrated with our derivation algorithm to solve this problem. We note that the semantic definition of choice, as given in [83], explicitly shows that the choice must be followed by a strict sequence. Therefore we discuss here the design model for strict sequence and alternatives. In the following, we discuss a basic algorithm for deriving a design model for a given role from the global requirements in the form of a single sequence of strictly sequenced states, and then discuss how choices can be integrated into this context.

Basic derivation algorithm for role r_n in the case of a linear strict sequence of states:

1. Create a local state s_l in the local design model for each state s_g in the global requirements. Call the generation function "genDM" to establish the design model for each local state s_l according to the corresponding global state s_g. Calculate for s_g the sets of initiating roles (IR), terminating roles (TR), and participating roles (PR).

2. For each global state s_g, except the last state in the sequence, consider the outgoing transition t (directed from s_g to the next state s_{next}). If $r_n \in TR_s$, a flow message should be sent to each role in IR of the next state (written IR_{snext}) except r_n, written as $(IR_{snext} - \{r_n\})$, see [4]. For this purpose, a new state called **"final state of s"** (written FIN(s)), is be created (in the local design model) containing as do-actions the sending of these flow messages. A completion transition is created from s to **FIN(s)**. We use the term **"outflow state of s"** to designate the FIN state of s, if it exists, or s itself.

3. For each global state s_g, except the first state in the sequence, consider the incoming transition t (from s_{prev} to s_g). If $r_n \in IR_s$, a flow message should be received from each role in TR of the previous state except r_n, written as $(TR_{sprev} - \{r_n\})$), see [4]. For this purpose, a new state called **"preliminary state of s"** (written prel(s)), is created (in the local design model) containing no do-action but having an outgoing transition which is triggered by the reception of one of the flow messages to be received. If more than one flow messages are to be received, an additional intermediate state is created for each additional flow message with an outgoing transition to receive the corresponding flow message. These additional states are linked sequentially (in an arbitrary order – since the reception pool allows the receiving role to determine in which order it wants to receive these messages) to finally lead to the state s_l. We use the term **"inflow state of s"** to designate the preliminary state of s, if it exists, or s itself.

4. For each local state s_l in the design model, except the last state in the sequence, create a completion transition from the outflow state of s_l to the inflow state of s_{next}.

We note that the flow messages introduced for the strict sequence are sufficient and necessary for the correct operation of the generated design model.

Figure 5.5(b) shows the generated design model for r_1 in the linear strict sequence in Figure 5.2(b). The simple states a_1 and a_4 are the remaining states (after deleting the dummy states) from the composite states cs_1 and cs_2, respectively.

What we explained above for the derivation of strict sequence can also be applied in the case of alternatives where a choice state may have two or more outgoing transitions. Here we consider the alternative in a composite state with strict sequence only. If the alternative is followed by a weak sequence, any role (other than the choice role) which participates in one of the alternatives can start execution before the choice, and this is not allowed. Consider the requirement model in Figure 5.2(c), which contains alternatives and strict sequence. Based on the derivation rules above and considering the alternatives, the local design model for r_1 is shown in Figure 5.6.

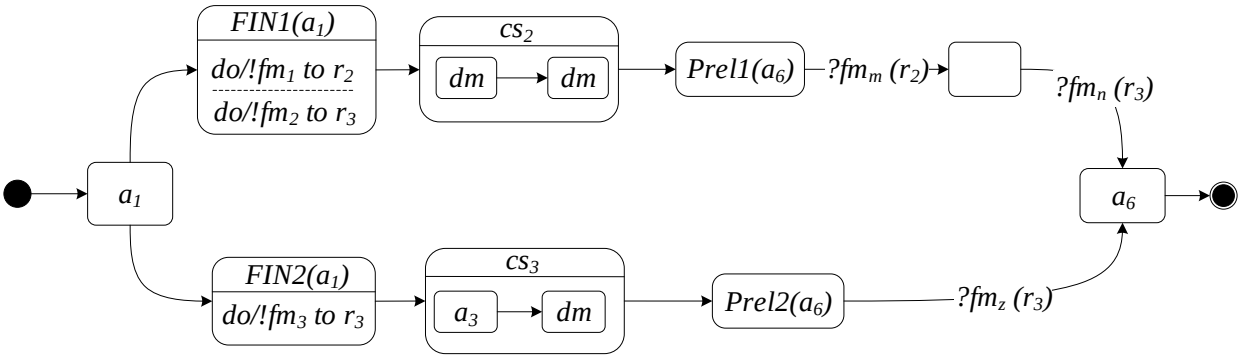


Figure 5.6: Local design model for r_1 derived from the requirement model in Figure 5.2(c)

As we see in the figure, for any composite state, the global function "genDM" is called recursively to evaluate the design model for that state, see for instance the composite states: cs_2 and cs_3. For the outgoing transitions from the choice state, each one will be treated separately based on the rules described before. In the first alternative, the coordination messages fm_1 and fm_2 are sent to $IR_{cs2} = \{r_2, r_3\}$ in the first FIN state of a_1 (denoted $FIN1(a_1)$) (see concurrent do-actions), and fm_3 is sent to the $IR_{cs3} = \{r_3\}$ in the second alternative (see state $FIN2(a_1)$). At state a_6 (executed by r_1) where the two alternatives are combined, r_1 receives coordination messages from the terminating roles (TR) of the previous state (in each alternative) in any order. $Prel1(a_6)$ and $Prel2(a_6)$ are the preliminary states of a_6; the receiving of flow messages starts after these states.

It is important to note a problem with choice propagation related to "dummy choice state". These are (dummy) choice states generated by our algorithm for the roles that have to follow the choice made by the role executing the choice state. If the dummy choice state is followed by dummy states (where the role does not perform any actions), the role may not know which choice should be taken. An example is shown by the example design model of in Figure 5.7(a), where the role does not know which choice to follow. One solution is to eliminate in each branch all dummy states until a (simple or composed) state is encountered where the given role participates. For example, eliminating the first dummy state "s_5_dm" in the right branch of the choice state $dmChoice_1$ allows the role to follow the right choice. This leads to the right branch shown in Figure 5.7(b). In the left branch,

we eliminate the dummy choice state $dmChoice_2$ as shown in Figure 5.7(b). We note that this design model has a different hierarchical model structure than the global model. We call this solution the *minimal approach*.

We propose a different solution. We propose a construction of design models that have the same structure as the given global model. This also means that the design decisions for a given composed state of the global model can be made without the knowledge of the environment of that state and without the knowledge of the structure of the composed states that are part of the given state. Therefore we propose that the choosing role of a choice state should send an additional *cim* (called *cimC* message in Chapter 7) to all roles for which the first non-dummy state in the branch that is chosen is a composed state, and the role is not initiating in that state, we call this approach *structure preserving*. This *cim* message here (see cim_1 on the left branch in Figure 5.7(c)) is sufficient, but not necessary, since there are other ways to obtain a conforming design, such as a design that is not structure-preserving, as we will see when we discuss the cimC message in Chapter 7.

We should mention here that using the *structure preserving* approach in the derivation of the design model. We are working on the different forms of composite states independently. This means, the solution which is used in a given state does not lead to a conflict with other solutions at different hierarchical levels, and this is an advantage.

5.2.6 Concurrency and Strict While Loop

The generation of a design model for a composed state that contains several concurrent sub-states is recursive without additional coordination messages. Since there is no interaction between the activities of the different concurrent sub-states, the design model for a particular role r_n is constructed by creating a composed state for a role r_n that contains concurrent sub-states that contain the design model (for r_n) for each concurrent sub-state of the global requirement model in which role r_n participates.

The generation of a design model for a composed state that contains a strict while loop is also recursive without additional coordination messages since it consists of alternatives

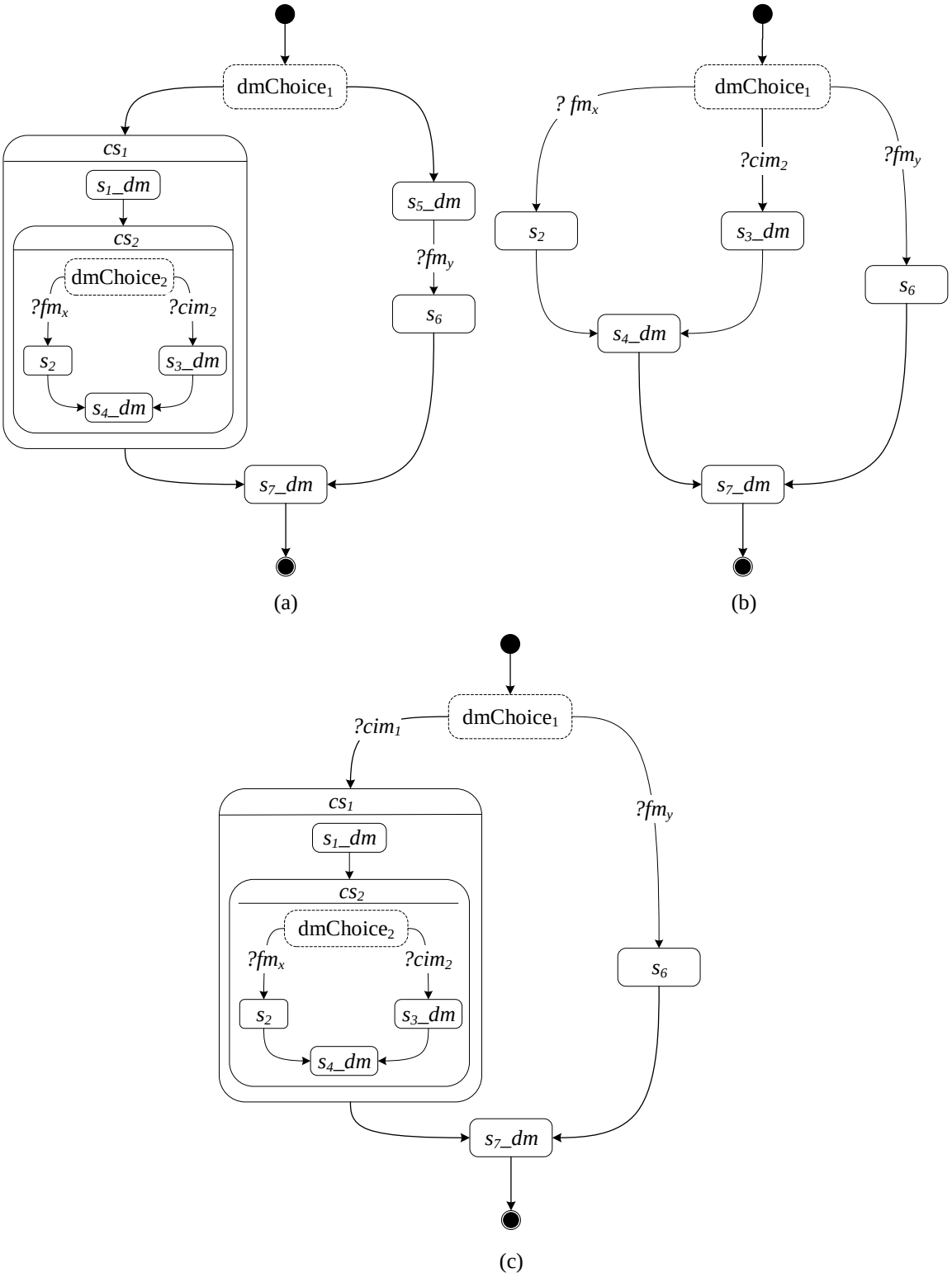


Figure 5.7: (a) incorrect local design model for the role r_n, (b) design model for r_n using *minimal approach*, (c) design model for r_n using *structure preserving approach*.

and strict sequence. It is sufficient to apply the rules explained in the previous section to the occurrences of strict sequences and the alternative which make up the strict while loop.

5.2.7 Weak While Loop

The derivation of the design model from a requirement model, including a weak while loop, was studied by many authors [4, 31, 2, 76]. Most of this research was in the context of requirement models defined in terms of MSC. Because of the weak sequencing between subsequent MSC, the messages received by a given role could arrive in an unexpected order which is called a *race condition* [11]. This problem is usually resolved by making a distinction between message reception and consumption: a received message is put into a message pool where it is stored until it is consumed in the order requested by the role.

However, in the case of a weak while loop, an additional problem may occur. A weak while loop may be **unbounded** [1]. In this case, a role may receive a message terminating the loop when there are still some messages waiting in the pool to be consumed within a repetition of the body of the loop. The role cannot decide whether to accept the terminating message for consumption. This situation was called **Termination Race** [76] (see also Section 4.3). To solve this problem, it was proposed to include an additional parameter in the messages of the loop which indicates how often the body was repeated to date (see for instance, [4, 31]). This solution was improved in [76] by indicating that the sequence number is not required in many cases, and by minimizing the number of messages that need such a parameter.

In the following, we discuss how to generate a correct design model for a given role r_n for a composed state in the global requirements model that contains a weak while loop, written in the HSM notation.

As in the case of the strict while loop, the rules described in Sections 5.2.4 and 5.2.5 can be applied to obtain a basic implementation of the weak while loop, which must be modified if sequence numbers are required. Algorithm 5.1 (see below) can be used to determine whether there is a termination race for this role. If there is a termination race, the first flow messages received by r_n in the loop body and in the follow-up part

should contain a sequence number (written as $?fm_x(seq)$). Also, the roles that send these messages to r_n (in the loop body and in the follow-up part) should include these sequence numbers in the send actions ($!fm_x(seq)$). These roles, as well as r_n, need a local counter n to record the number of times that the loop was executed. As explained in [76], there are two ways to implement, in role r_n, the consumption of the next message with the correct sequence number. If the interface to the pool allows the role to request the consumption of a message with a particular sequence number, then the role may simply ask for a message of the body with number $(n+1)$ or a message of the follow-up with sequence n. In case the pool interface does not allow this choice (but only a choice based on the message types), an alternate implementation is described in [31, 76].

5.2.7.1 Check for Termination Race

In this section, we determine whether the termination race could happen for a role r_n in a given weak while loop. Based on the results in [76], there is no termination race for role r_n if the first event of r_n in the loop body is before the last event of the loop initiator (r_i) inside the loop body. To check the termination race, we evaluate the function $precede(r_n, r_i, loop\ body)$ (see Algorithm 5.1). We know that no termination race exists if the result is true, since the meaning of $precede(r_1, r_2, s)$ is the following: An action of r_1 precedes an action of r_2 during the execution of the behavior of the composed state s.

The function $precede(r_1, r_2, s)$ is evaluated according to the different forms that the composed state s may take according to the cases mentioned in Section 5.2.1.

Algorithm 5.1 *precede* Function

```
Boolean Function precede(r_1 , r_2 , s)
    switch s do
        case (1) linear strict sequence of states s_1,s s_2,s ... ,s s_k do
            return ((∃ i ∈ [1 : k] : precede(r_1, r_2, s_i)) OR (∃ i, j ∈ [1 : k] : i < j,
            r_1 ∈ PR_si and r_2 ∈ PR_sj))
        case (2) strict state machine do
            return (∀ linear sequences ls allowed by s, we have precede(r_1, r_2, ls))
        case (3) several concurrent sub-states of the form ccs_1 || ccs_2 || ... || ccs_k) do
            return (∃ i in [1 : k] : precede(r_1, r_2, ccs_i))
        case (4) strict or weak while loop with body and follow-up do
            return (precede(r_1, r_2, s.follow-up) OR
            (r_1 is the loop initiator (r_i) and r_2 ∈ PR_s.follow−up))
        case (5) linear weak sequence of states s_1,w s_2,w ... ,w s_k do
            return ((∃ i ∈ [1 : k] : precede(r_1, r_2, s_i)) OR
            (∃ r_x1, r_x2, ... r_xm ∈ PR_s with r_x1 = r_1 and r_xm = r_2 and
            a number of states s_y1, s_y2, ... s_y(m−1) with yi < y(i+1) for i = 1, ... m−2 such
            that precede(r_xi, r_x(i+1), s_yi) for i = 1, ... m − 1))
```

We give in the following a reasoning for the evaluation of the function $precede(r_1, r_2, s)$ in the different cases:

- Case (1) – lineat strict sequence of states: The function is true if there is a state s_i in the sequence for which *precede* is true or if role r_1 participates in one of the states and role r_2 in a subsequent state.
- Case (2) – strict state machine (the composite state structure may contain alternative branches and loops and all states are connected by strict sequence): Different linear strict sequences of states are possible. Since we do not know which one will be executed, *precede* must be true for all of them.
- Case (3) – concurrent substates: Since all concurrent substates will be executed, it is sufficient that *precede* be true for one of them.
- Case (4) – while loop: The body may not be executed, while the follow-up will always be executed.

- Case (5) – weak sequence: Like for strict sequencing, *precede* may be true because it is true for one of the substates s_i . The second possibility relates to intermediate roles. In the case that $m = 3$, we have the roles r_1 , r_2 and an intermediate role r_{x2} . Suppose that $y1 = 2$ and $y2 = 5$ (and $k = 6$). Then the formula states that $precede(r_1, r_2, s)$ is true if $precede(r_1, r_{x2}, s_2)$ and $precede(r_{x2}, r_2, s_5)$ are both true. The actions of r_{x2} in state s_2 are before those in state s_5 because of the weak sequencing relationship.

5.3 Chapter Summary

This chapter is a slightly edited version of our conference paper in [77]. It deals with deriving a distributed design model from a global requirements model written in the notation of extended Hierarchical State Machine (HSM). In this chapter, we extend the UML notation of HSM to indicate the roles that participate in the actions of each state of the global behavior. A simple state represents some local actions, while a hierarchical state usually represents a collaboration between several roles. Our global HSM requirements model describes the sequencing of collaborations and local actions. We compare this notation with other notations such as UML Collaborations, Hierarchical Message Sequence Chart (HMSC), PO-Charts and others. Then we explain how a distributed design model, including all required coordination messages between the different system roles, can be automatically derived from a global requirements model. We consider the following sequencing constraints between different collaborations: weak or strict sequence, alternatives, weak or strict while loop, and concurrency.

The local design model generated by our algorithm, for each component, is an HSM which represents the local actions identified in the requirement model and executed by this component, and the exchange of the coordination messages which are generated by our proposed algorithm. These local state machines can be easily implemented by any suitable tools that generate code from a design model written in UML HSM.

Chapter 6

Resolving Competing Initiatives in Distributed Applications

As we said before, different problems are associated with the realizability of the global requirements model. The realizability problem which will be addressed in this section is related to non-local choice.

If the global requirements model contains several alternatives, specified by means of the choice operator, the choosing roles are the ones who decide which alternative will be executed [4]. The choice is called "local" if only one role makes the decision. The choice is non-local if several roles are involved.

This decision is local (and simple) if the initiating roles of all alternatives are local to one role. The decision process gets complicated if more than one role are involved in selecting the alternative, and this is called non-local choice [9].

There are several types of non-local choices from the applications point of view (see Section 3.1.2). *Competing initiatives* are one of these types and occurs when several roles propose different alternatives, and there is no coordination between them, and only one of them should be selected.

As we said in Chapter 3 different researches had considered the non-local choice problem, and there is no consensus on how it should be treated. Gouda [33] solves the problem

of competing initiatives by giving priority to one role over the other. In this chapter, we discuss the following:

- We follow Gouda's solution for competing initiatives and show that this solution requires the undoing of early actions for the low priority behavior.
- We propose a solution for avoiding the undoing of actions using special coordination messages where the role with lower priority asks for permission before it starts its behavior.
- We also consider an example of competing initiatives in a system where three roles are involved in a distributed choice. We discuss some distributed design solutions for this example.

6.1 Competing Initiatives Problem

As stated in the introduction, competing initiatives occur when more than one role take uncoordinated choice initiatives concurrently. In this case, the decision to select the next alternative (behavior) is not localized and may lead to conflicts in the global behavior execution.

Figure 6.1 shows a global model for freelance task distribution (written in extended HSM notation) that contains competing initiates. As we said in the HSM representation of the local choice (see Section 5.2.1), the first actions of the alternatives are integrated into the choice state. In the case of competing initiatives, these actions are executed by different roles, and the choice state is fictitious and called "non-local choice" as indicated in the example in Figure 6.1. There is no role to execute it. The suffixes "_S" of the names of the composite states means that the states within this composed state are sequenced by strict sequence. Each simple state is associated with one role, which is indicated as a suffix (after "_") of the state name. For example, the simple state "taskCompleted_dv" is executed by the Developer (dv) role and the simple state "TaskReceived_dis" by the Dispatcher (dis).

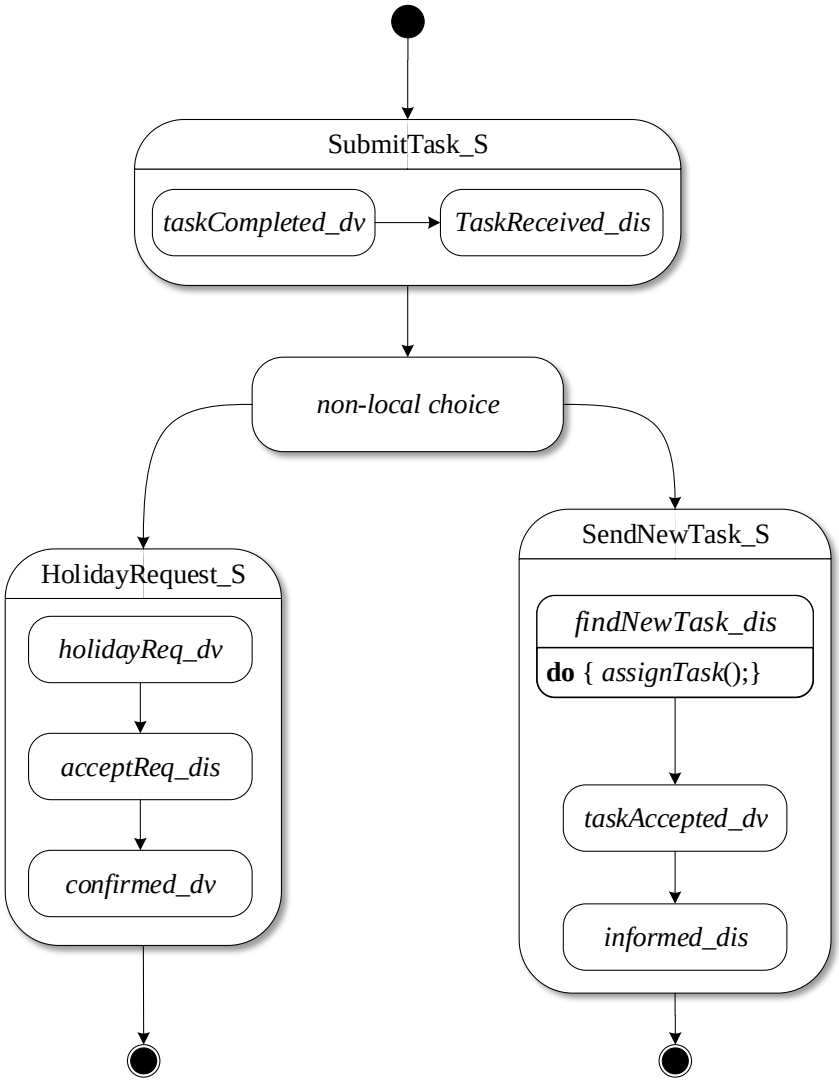


Figure 6.1: Freelance task distribution example (with competing initiatives) written in extended HSM notation.

After the Developer completes the previous task (see the state "taskCompleted") and the Dispatcher is confirmed, they arrive at the "non-local choice" state. Then, either the Dispatcher assigns a new task (if it exists) to the Developer (see the state "findNew-Task") or, possibly, at the same time, the Developer requests a holiday (see the state "holidayReq"), which are two competing initiates. Only one of these alternatives should be executed.

Our derivation algorithm (see Section 5.2) always assumes that there is a local decision that is done by one role. In order to process competing initiatives using the derivation algorithm, we should include the two roles which are involved in competing initiatives in the choice state and generate a design model based on that. In this section, we propose a solution to this problem.

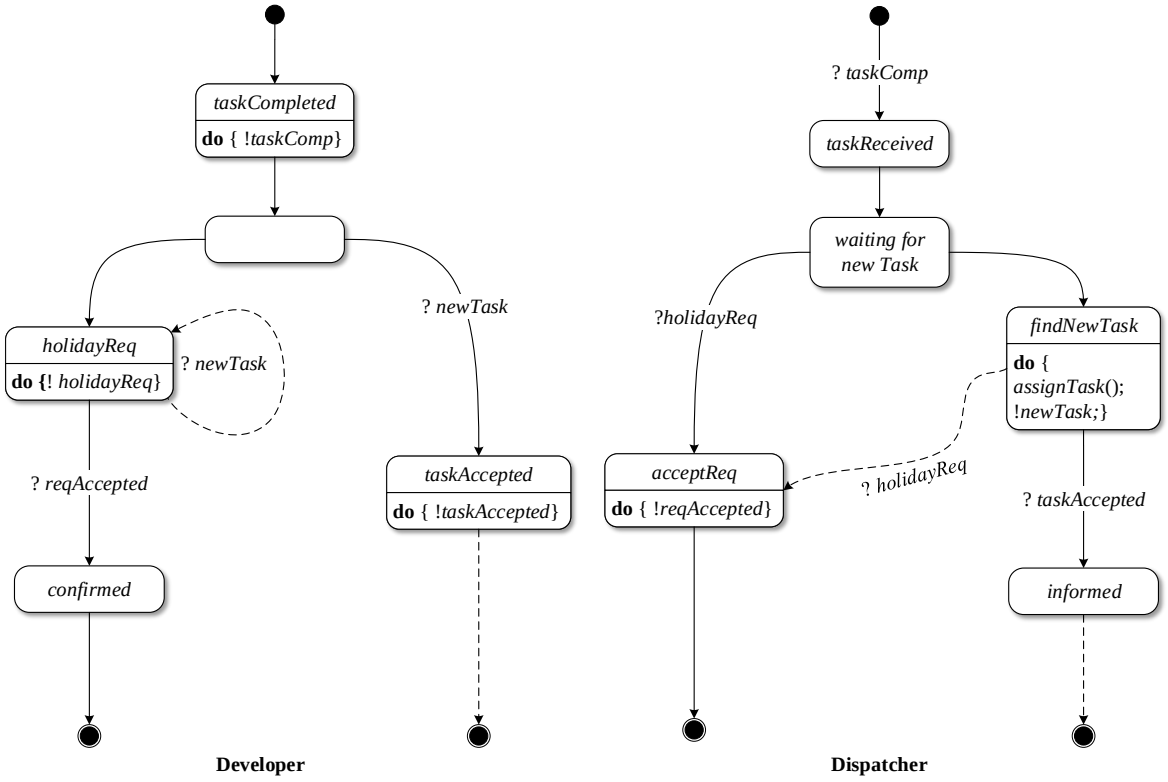


Figure 6.2: Design models for the Developer and Dispatcher (corresponds to the global model in Figure 6.1) based on Gouda's solution.

Figure 6.2 shows the design models for the Developer and Dispatcher (without the dashed arrows). We can see that when the Developer is in the state "holidayReq", he

could receive a *newTask* request from the Dispatcher. Also, the Dispatcher might receive a *holidayReq* from the Developer while executing the state "findNewTask". Both cases lead to conflict.

To resolve this conflict, Gouda proposed to give different priorities to the two alternatives. The dashed arrows in Figure 6.2 are introduced by Gouda's solution, given that higher priority is given to the holiday request by the Developer if a conflict is detected. We can see that if the Developer initiates his behavior by sending a holiday request (*holidayReq*) and at the same time, the Dispatcher initiates his behavior by sending a *newTask* message, this message will be ignored by the Developer since he has priority.

However, if the Developer does not initiate a holiday request and follows the Dispatcher's decision, he will receive the message *newTask*, and then send *taskAccepted*. In the design model for the Dispatcher, he follows the decision of the Developer by accepting the holiday request when it is received. We note that receiving the message *taskAccepted* by the Dispatcher removes the ambiguity about which alternative is selected, and the Dispatcher knows that the subsequent actions will not be interrupted (see, for instance, the dashed region after the state *informed* in Figure 6.2). At any time, if the message *holidayReq* is received by the Dispatcher, the actions executed by the Dispatcher before the reception of the *taskAccepted* message could be interrupted.

We note that the authors of [34] also applied Gouda's approach for solving the mixed-initiative problem. However, they represented the competing alternatives as concurrent regions in a composite state. These regions are assigned primary and secondary priorities. Their implementation is not intuitive because they assumed concurrent initiatives inside the composite state instead of alternatives. They discussed the implementation of one role only (the one with the highest priority), and built a tree by unfolding the composite state and then pruning all the outgoing transitions of the second region (the region with a secondary priority). However, they did not address the implementation of the other role.

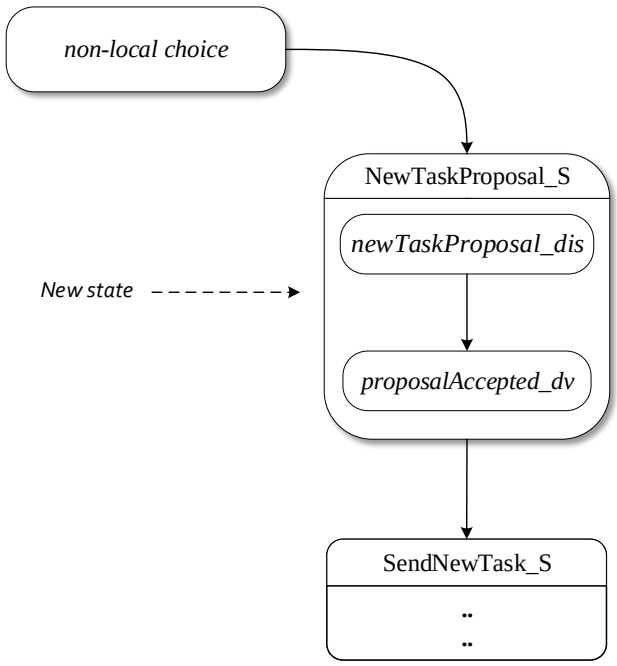


Figure 6.3: Modified global model using a permission design.

6.2 Avoiding Undoing of Early Actions

In competing initiatives, the side effects of the actions in the initial states which are executed by the low-priority role must be undone when the higher priority behavior is executed. In Figure 6.2, both actions: *assignTask()* and "!*newTask*" at the Dispatcher side (low-priority role) could be interrupted if the action *holidyReq* is executed by the Developer (high-priority role); in other words, the side effect from the action *assignTask()* must be undone when the Dispatcher receives a *holidayReq* message and goes into the state "acceptReq".

The undoing of actions is common in long-term transactions, which include a sequence of normal transactions. If a cancel request is initiated in a long-term transaction, the side effect of the preceding sub-transactions must be undone. Consider, for example, a travel reservation application where the user first reserves a hotel then searches for a flight, and if he can't find a suitable flight, he will cancel the hotel. The undoing of such actions can be quite complex, and therefore it is good practice to design an application such that the

undoing of actions can be avoided.

In cases where the interruption of early actions should be avoided in competing initiatives, we propose a design where the low priority role first asks for permission before initiating its behavior. Figure 6.3 shows a modified version of the global model of Figure 6.1 where the permission design solution is used before the state "SendNewTask" in the right branch; in the form of a new composite state "NewTaskProposal". In this state, the Dispatcher sends a new task proposal, and the Developer confirmed before sending the actual task.

The design models for the roles Developer and Dispatcher (corresponding to the global model of Figure 6.3) are shown in Figure 6.4. We can see that the Dispatcher sends a new task proposal message (see *newTaskPr*) and waits for the Developer approval (receiving the *ok* message) before executing the task assignment. Receiving a message other than *ok* (*holidayReq* message) by the Dispatcher means the higher priority behavior (Developer) has started, and the Dispatcher must follow.

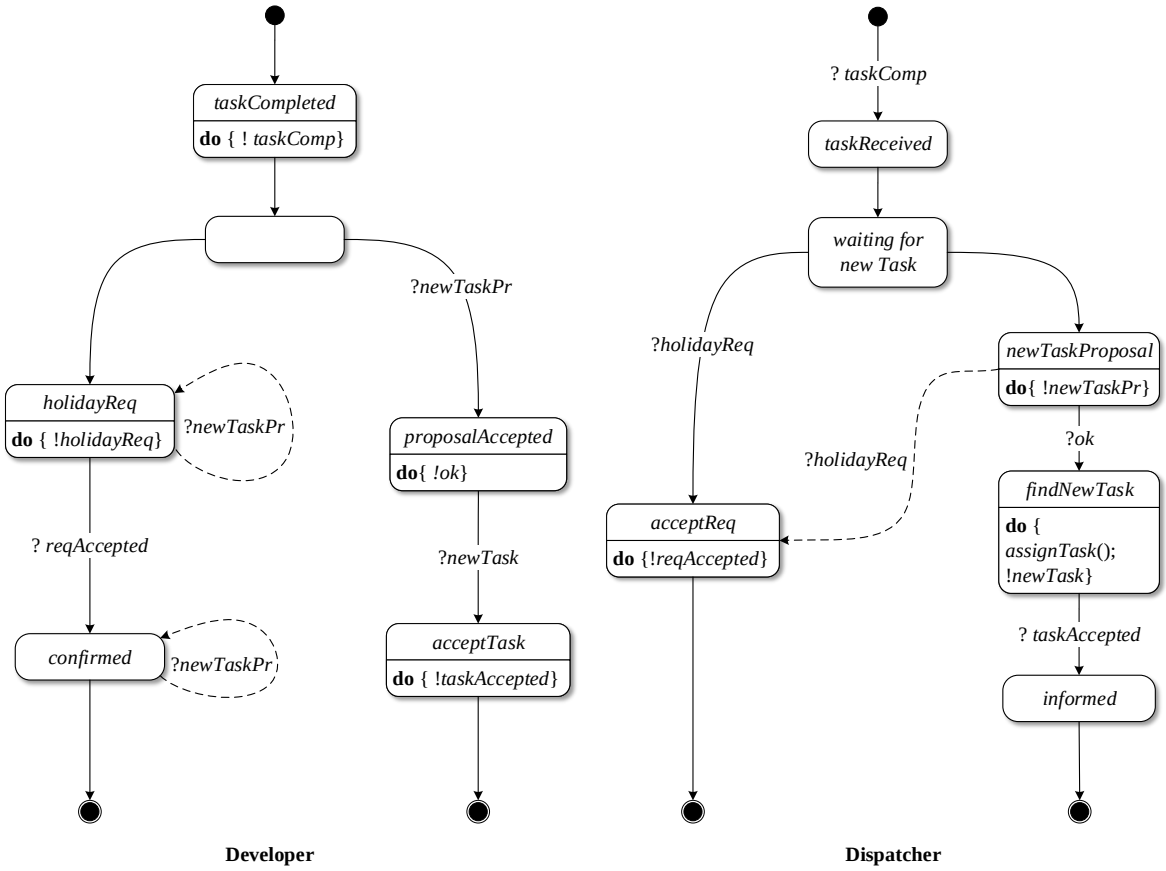


Figure 6.4: The design models for the roles Developer and Dispatcher (corresponds to the modified global model in Figure 6.3) using the permission design

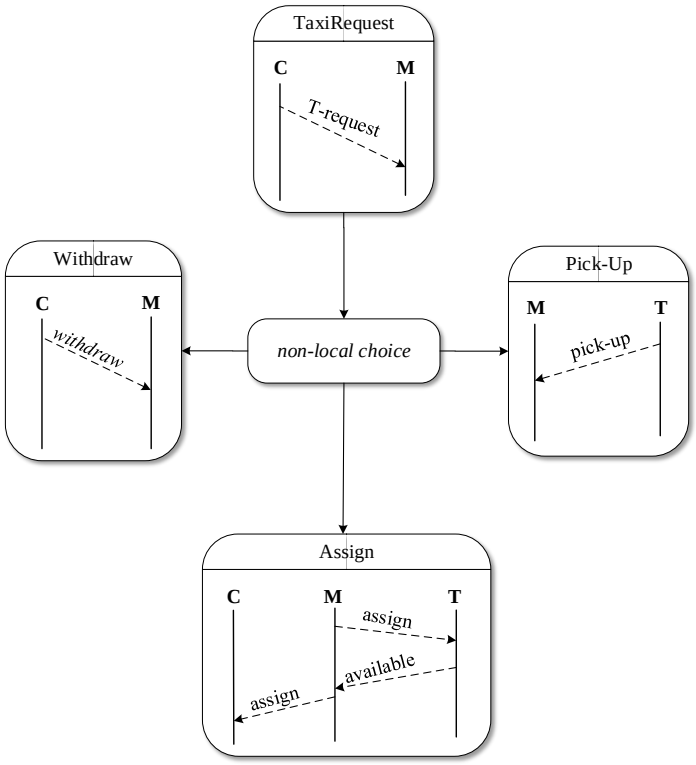


Figure 6.5: Reservation of Taxi example (a global model).

6.3 An Application Involving Three Competing Initiatives

We considered in Sections 6.1 and 6.2 the problem of competing initiatives involving two roles and discussed different ways to resolve the problem: Gouda's approach, and the use of a permission design, which introduces additional states in the global model to avoid the need for undoing the side effects for the early actions in the low-priority alternative. In this section, we consider an application involving three competing initiatives (the reservation of a taxi) and discuss some distributed design choices for this problem.

An example of reservation of a taxi is shown in Figure 6.5 (a complete example written in hierarchical state machine notation is presented in Section 5.1.3, and it is inspired by [84]). There are three roles: Manager {M}, Client {C}, and Taxi {T}. After the Client sends a request for a reservation of a Taxi to the Manager, the following scenarios could

happen simultaneously:

1. The Manager assigns the Client to an available Taxi.
2. The Client withdraws the request.
3. The available Taxi picks-up another Client from the street.

These scenarios involve two competing initiatives: *withdraw/assign* between the Client and the Manager, and *pick-up/assign* between the Taxi and the Manager. There are many scenarios for the interactions between the Client, Taxi and the Manager roles following the global model of Figure 6.5. We discuss in the following some of them. The sequence diagram in Figure 6.6 (a) shows a scenario for the *withdraw/assign* conflict. We can see that the Manager sends first an *assign* message to an available Taxi and waits for the *available* message, and then sends *assign* to the Client. At any time, the Client might send a *withdraw* message to the Manager (the *withdraw* can be sent by the Client even after the *assign* message is received), which cancels the request for a Taxi. If the *withdraw* message is received by the Manager after the *assign* is sent to the Taxi, a *cancel* message must be sent to cancel the assignment. Another case shows the *pick-up/assign* conflict as presented in Figure 6.6 (b). We see that the Taxi could send a *pick-up* indication to the Manager, and the Manager sends an *assign* to the Taxi at the same time. In this case, the assignment operation will be canceled, and the Manager should find another available taxi (T') and send it another *assign* request.

As we see, the assign operation is involved in both *withdraw/assign* and *pick-up/assign* conflicts, and this includes three different roles: Manager, Client, and the Taxi. Figure 6.6 (c) shows a scenario where both conflicts occur in the same transaction.

In order to resolve the competing initiatives problem in the Taxi example, it is important to show the state machines which describe the behaviors of the roles (also called design models): the Manager, Client, and Taxi; and the behavior should conform to the global model in Figure 6.5. Figure 6.7 shows the design models for the roles: Client and Taxi, given that priority is given to *withdraw* by Client, and to *pick-up* by the Taxi. In Figure

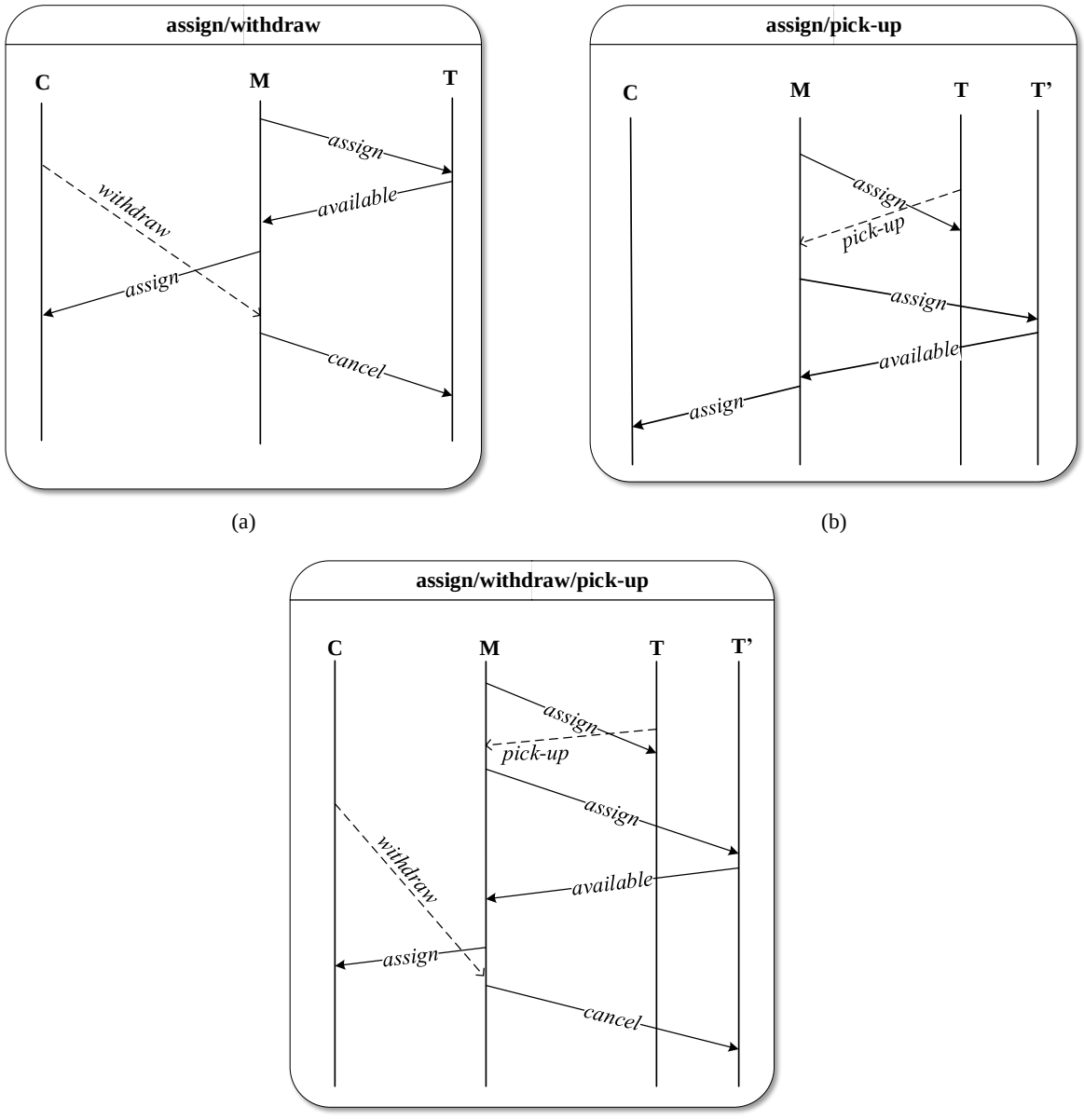


Figure 6.6: Competing initiatives in the Taxi example: (a) *withdraw/assign*, (b) *pick-up/assign*, (c) assign/withdraw/pick-up conflicts.

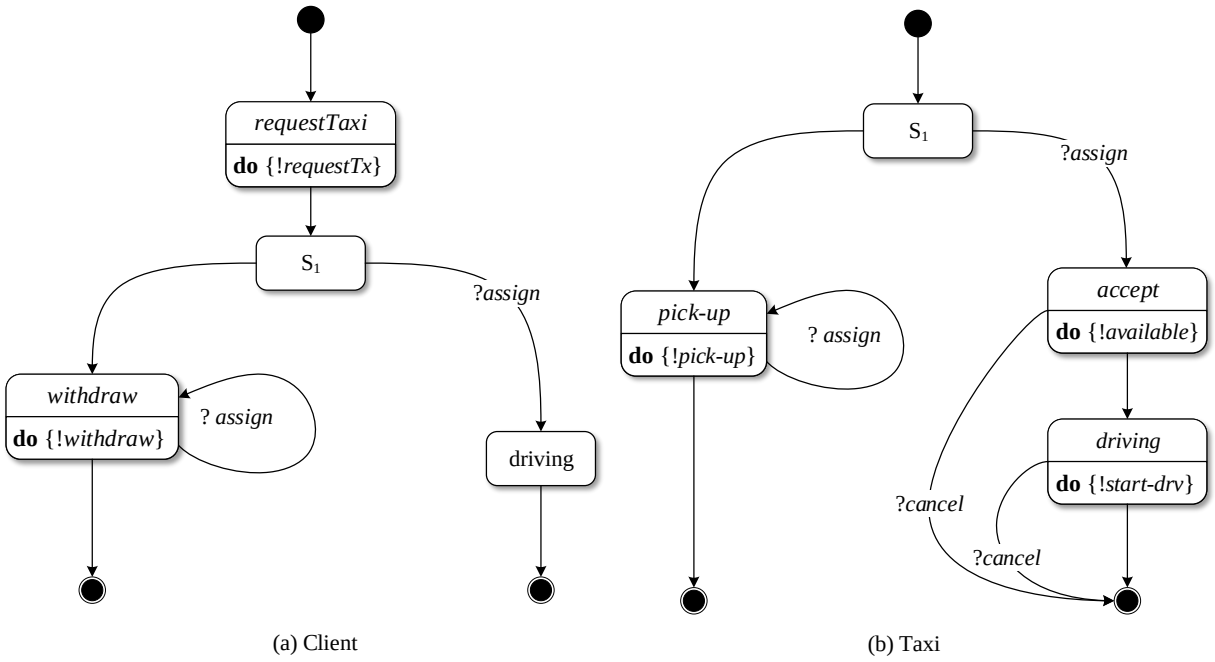


Figure 6.7: The design models for the (a) Client, and (b) Taxi, roles in the Taxi example.

6.7 (a), the Client with the highest priority can either *withdraw* the taxi request (see "!*withdraw*") or wait for the *assign* from the Manager (see "?*assign*"). If the *assign* is received by the Client after he sends the *withdraw*, the *assign* message will be ignored. Similarly, the *pick-up* operation in the design model of the Taxi role (see Figure 6.7 (b)) has a higher priority over the *assign* by the Manager. Also the *cancel* message could be received by the Taxi after receiving the *assign* message from the Manager, which means in the states "accept" and "driving".

The behavior of the Manager role is shown in Figure 6.8. The Manager first receives a request for a Taxi from the Client ("?*requestTx*"), and then sends a *assign* message to an available Taxi. Receiving a *pick-up* request means the Taxi is not available, and the Manager should send another *assign* to another available Taxi. If the *available* message is received from the Taxi, this means, the Taxi accepts the assignment, and the Manager can then send an *assign* message to the Client. In the state s_3, the Manager receives *start-drv* from the Taxi, which means the Taxi is driving to pick-up the Client.

A *withdraw* request could be received by the Manager in the following situations:

- In the states s_3 or s_4 (after the Manager receives *available* from the Taxi): since the *withdraw* by the Client has a higher priority, the Manager must abort the assignment and inform the Taxi (sends *cancel* to the Taxi) after the *withdraw* is received.

- In the state s_2 (after sending the *assign* to the Taxi): If the Taxi replies by *available* (see state s_5); the Manager must cancel the assignment and inform the Taxi (send *cancel* to the Taxi). If the Taxi is not available (after Manager receives a *pick-up* request), the assignment will be ignored implicitly.

- In the state s_1 (before sending the *assign* to the Taxi): The assignment will be ignored by the Manager and no need to inform the Taxi.

In this application, the Manager has a database which has a list of the current Clients and the assigned Taxi for each Client. The Manager should update this information when needed, and this is not very complicated. For the *assign/pick-up* conflict, the Manager sends an *assign* message to the Taxi and waits for the *available* reply before contacting the Client.

In some sense, the Manager here does a permission request in the form of the *assign* message; in addition, the *available* reply is like the *ok* message (discussed in Section 6.2), which means that the Taxi accepts the assignment. However, using the permission design for the *withdraw/assign* does not make sense because the Client must be allowed to withdraw from the taxi request before and after the *assign* is received from the Manager.

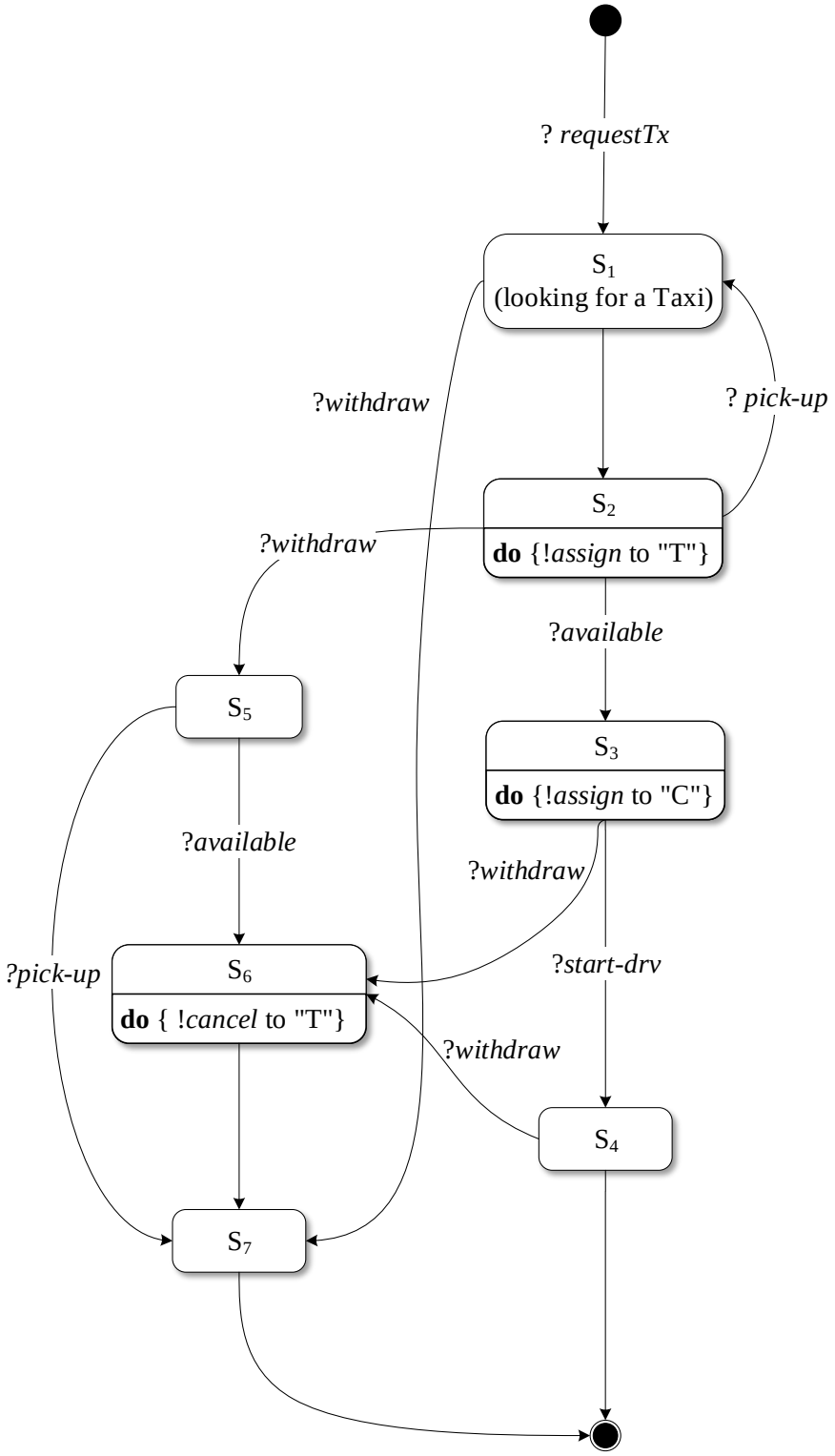


Figure 6.8: The design model for the Manager role in Taxi example.

6.4 Chapter Summary

This chapter deals with the problem of non-local choice, and more specifically, with competing initiatives.

Competing initiatives occur when there are several physical roles involved in a choice without coordination, and several alternatives could be initiated concurrently. In competing initiatives, the alternatives have different goals, and only one of them should be selected.

Competing initiatives cause conflicts in the distributed design model for several roles. In this chapter, we use Gouda's solution for solving these conflicts by giving different priorities to the different alternatives. However, his solution may require the undoing of the side effects of early actions in the low-priority behavior and considers two roles only. This chapter proposes a solution to avoid the undoing of actions using a permission design, by which the lower priority role asks for permission before initiating its behavior. By introducing this feature in the global behavior model, the early actions of the low-priority behavior are only executed after being accepted by the other party, thus avoiding the need for undoing. We also consider an application involving three competing initiatives (the reservation of a taxi) and discuss some distributed design choices for this problem. We show that in certain cases, the use of the permission design does not make sense.

Chapter 7

Derivation Tool

This chapter presents the overall architecture of the derivation software tool, which does the transformation of a global Hierarchical State Machine Diagram into local State Machine Diagrams (one for each role). The derivation process involves five phases described in detail in this chapter. Here is an overview of these phases:

1. Parsing of the input Umple file and generation of the Java instances (corresponding to the input Umple file): The Umple file represents the global model written in a Hierarchical State Machine Diagram. The parsing process of the input file also involves the syntactic validation, making sure that the input Umple file is well-formatted. This phase also generates Java objects (internal Umple representation corresponding to the extended Umple meta-model), which are needed by the following phases.

2. Calculation of Initiating and Terminating roles: For a composite state, the set of initiating, terminating and participating roles (which is important for the derivation of a roles design models) are calculated during this phase.

3. Calculation of the coordination messages that are required by the derivation algorithm.

4. Generation of the design model for each role: The transformation rules (presented in the previous chapters) are applied and generate a design model (in Umple internal representation) that represents the behavior of each role.

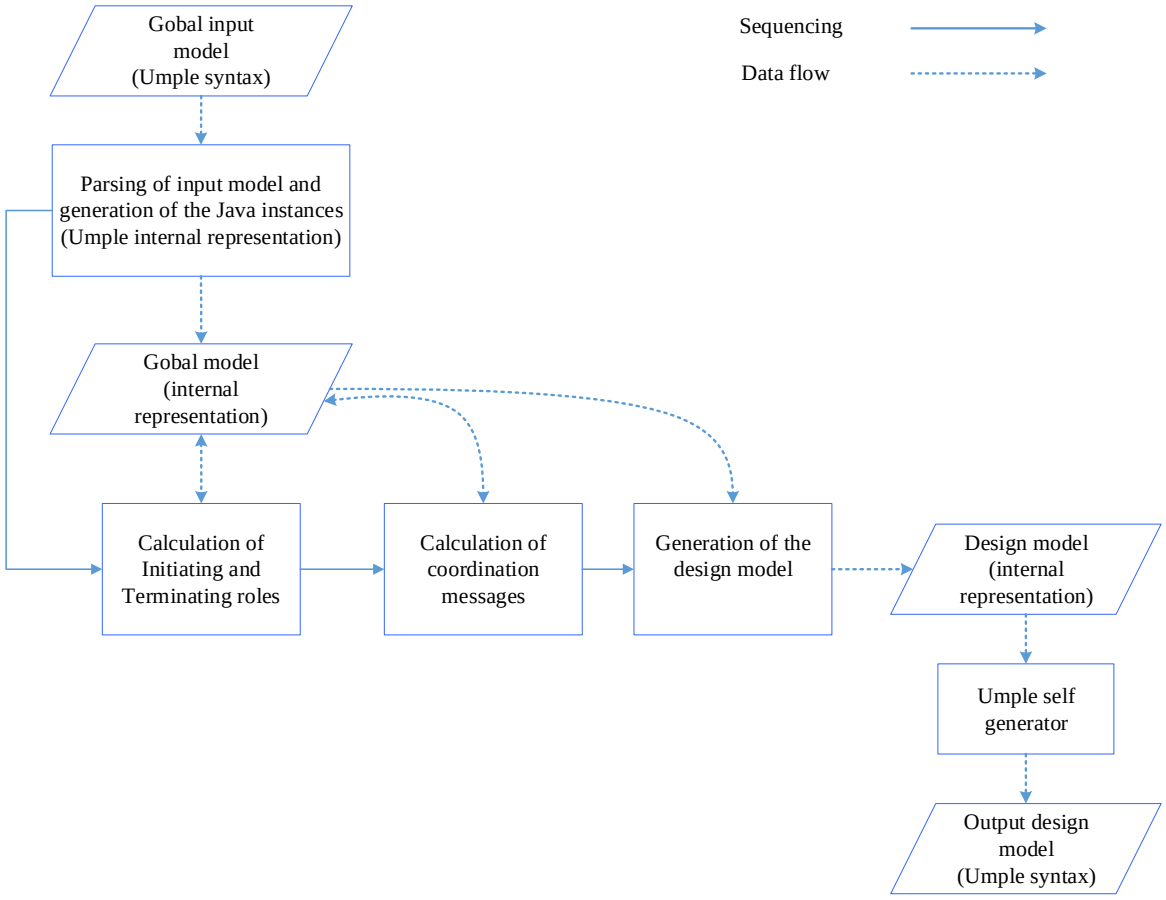


Figure 7.1: Activity Diagram showing the operation of the derivation tool.

5. Umple Code Generator: the Umple internal representation (Java objects) for each role is converted to Umple text using the Umple self generator. The generated Umple file is a state machine that represents a design model for the target role written in Umple syntax.

The existing Umple tools support Phases 1 and 5. The Umple self generator, which should be used in Phase 5, was not working correctly for Hierarchical State Machine. We fixed it, and now it is working correctly. We implemented the programs that realize Phases: 2, 3, and 4. The overall architecture for the derivation tool is shown in Figure 7.1.

7.1 Representations of the Global Input Model

The Derivation Tool takes as input a Hierarchical State Machine (HSM) describing the global view of a distributed system and derives the designs of the different roles participating in the system. The output of the tool is a set of UML HSM representing design models of each system role individually. For each system role, the corresponding HSM generated by the derivation tool is executed locally by this role. However, for the global HSM diagram of the distributed system, which is the input for the tool, each state in the diagram may involve several system roles that are participating in its execution.

A HSM describes the global requirements model as we discussed before (see, for instance, Chapter 5 and [77]). Each state is either a simple state or a composite state, which represents an embedded state machine. Using HSM, we assume that a simple state is associated with a single role and that the local action of such a role is executed as the "do-action" of that state (by the role responsible for that action). For this purpose, we extend the UML hierarchical state machine semantics of the input language in order to define the role that performs the do-action of a simple state. This extension of adding roles to the states is important to derive a distributed design model from the given requirements. The second extension is to define the type of sequencing for the state machine or for composite states. The sequencing will be either strict ($;_S$) or weak sequencing ($;_W$), between all states within a given state machine.

The following are the type of states and the proposed naming convention for the global model written in our HSM:

- **Simple State**: For each simple state in the input model, we append to the name of the state "_ $< name\ of\ role >$". Consider a state S_n associated with a role r_n, the state name is written as $S_n\_r_n$. We use "_" instead of the curly bracket (proposed in Chapter 5, and in [77]) since it is not allowed by Umple in the state name.

- **Composite State**: A composite state represents another state machine. The composite state name is associated with the suffixes "S" (which refers to strict) or "W" (refers to weak) to differentiate the type of sequencing, or with the suffix "WL" to

differentiate the weak while loop. These prefixes are written after "_" in the state name. Consider a composite state CS_n , it has the following forms:

1. A linear sequence of states (possibly composed) **weakly composed**. The composite state name is written as CS_n_W. See the example in Figure 7.2(a).
2. A linear sequence of states (possibly composed) and connected by **strict sequence**. The composite state name is written as CS_n_S. See the example in Figure 7.2(b).
3. A **Choice** with alternatives, it has an initial and final state connected by strict sequences. The state with different outgoing transitions is the choice state and it must be a simple state. The composite state name is written as CS_n_S. See the example in Figure 7.2(c).
4. Several **concurrent** states (possibly composed), and they are either weakly or strictly composed. The composite state name is written as CS_n_S or CS_n_W. See the example in Figure 7.2(d).
5. A **strict while loop** starts with a choice state (simple state), then the loop body and follow-up states (possibly composed), all transitions have a strict sequence type. The name of the composite state is written as CS_n_S. See the example in Figure 7.2(e).
6. A **weak while loop** is written as CS_n_WL, and starts with a choice state (simple state), then the loop body and follow-up composite states. The transitions between the initial choice state and the body and follow-up states are strict sequences. However, the transition from the Body state back to the initial state is a weak sequence. The Body and Follow-up states have a "Body" and "Fup" suffixes, respectively, in the names of the states. See the example in Figure 7.2(f).

The transitions between the states are "completion transitions", which means they are triggered without an event and executed as soon as all actions (entry, exit, and do actions) of the state have been completed. The completion transition between simple states s_1 and

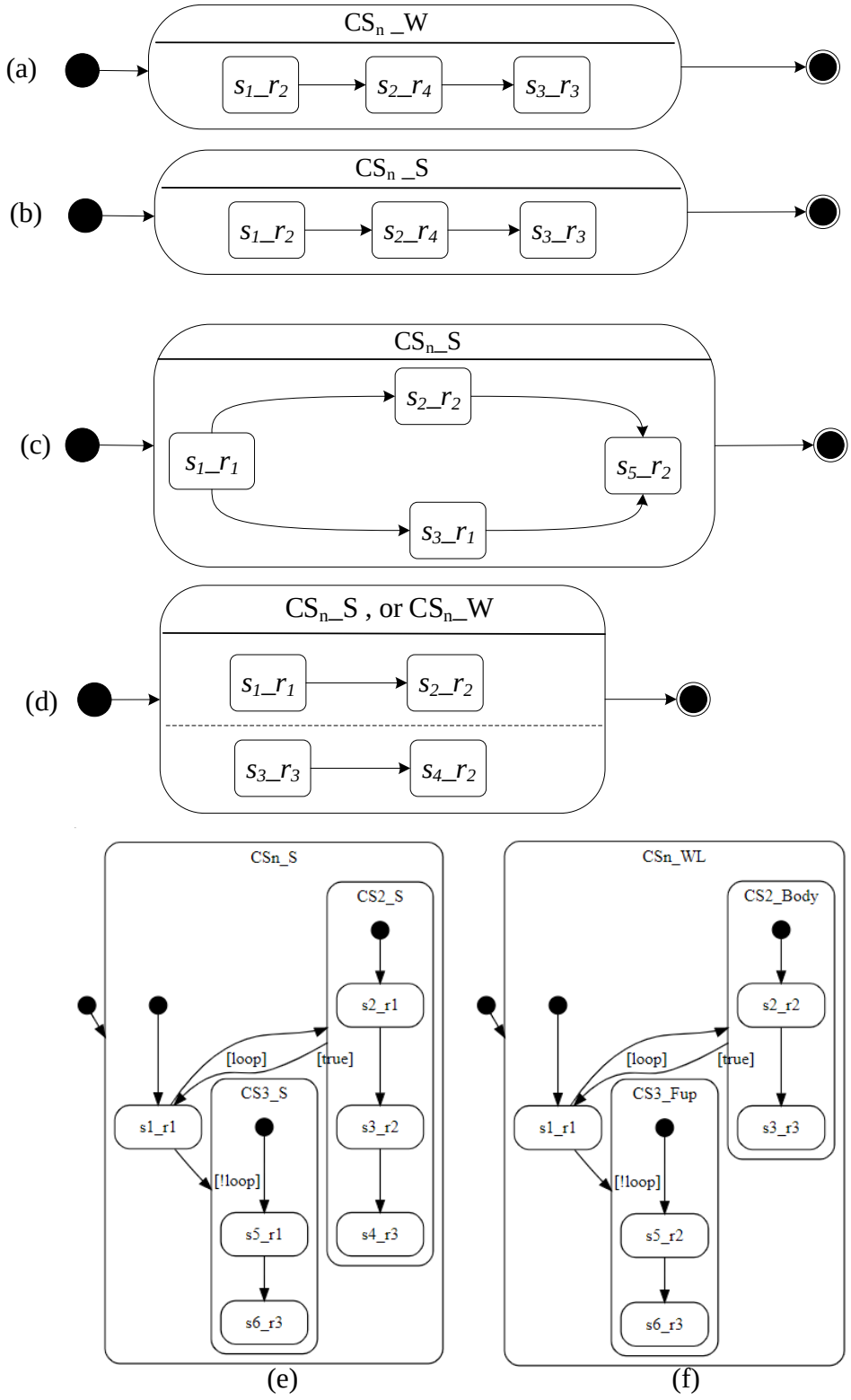


Figure 7.2: Graphical representation for different input models (written in HSM notation) of the tool.

s_2 is written in the linear Umple syntax inside s_1 as " $->$ s_2". Since Umple does not support the completion transition between composite states, we use a completion transition with a "true" guard to solve this problem (as a workaround). The completion transition from composite state CS_1 to state s_2 is written inside CS_1 as " [true] $->$ s_2". In order to implement these naming conventions and the associated semantics, an extension of the Umple meta-model is required and it is described in the following section.

```
1  sm1_S {
2      CSn_S {
3          s1_r1 {
4              -> s2_r4;
5          }
6          s2_r4 {
7              -> s3_r3;
8          }
9          final  s3_r3 {
10         }
11     }
12 }
```

Listing 7.1: Global input model written in Umple syntax for the composite state in Figure 7.2(b)

The models in Figure 7.2 represent the graphical representation of the tool input and correspond to global input models written in Umple linear syntax. For example, the linear Umple syntax for the composite state CS_n_S in Figure 7.2(b) is shown in Listing 7.1. The state machine sm_1_S has strict sequence as indicated in its name at line 1.

The following are the assumptions and constraints for the global input model of the tool:

- All alternatives in a given choice must end together in the same simple state, and this makes the derivation of the design model easier (see Section 5.2).
- If a composite state contains concurrency (concurrent state machines), the sequencing type of the composite state will be applied to all state machines inside. The reasoning

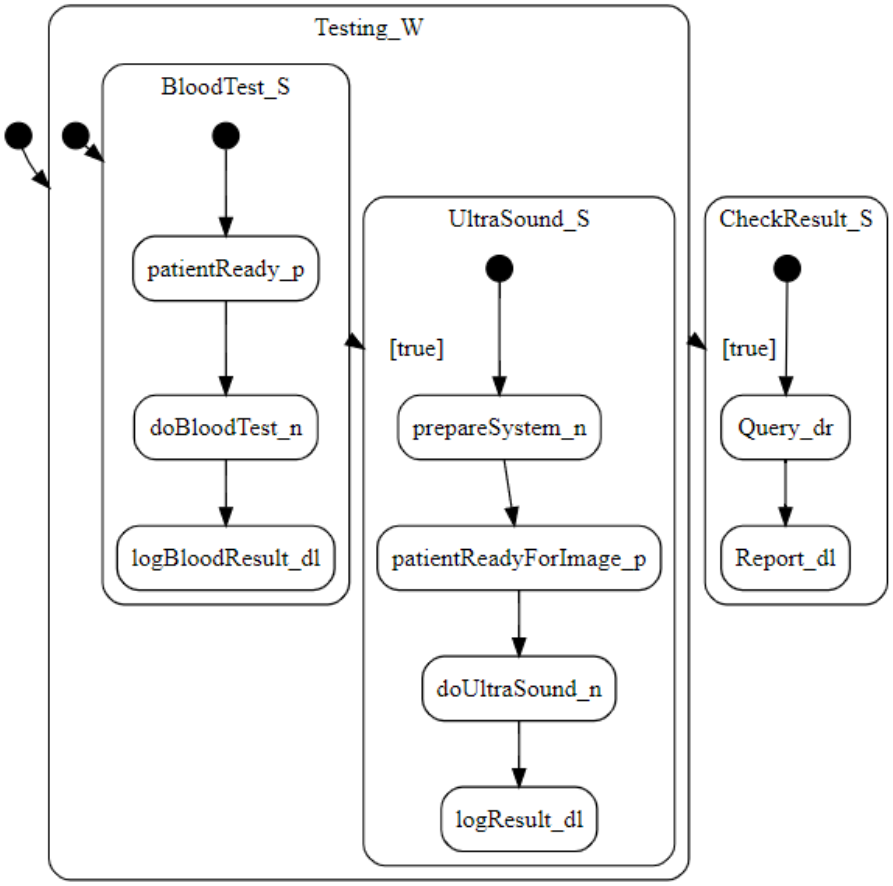


Figure 7.3: Do test activity written in the notation of extended HSM (figure generated by Umple Online).

behind those assumptions will be discussed later in this chapter.

The example in Figure 7.3 shows a global model of a *doTest* collaboration written in the notation of the extended Hierarchical State Machine (this example was inspired by the Test collaboration in [2]). It defines the global behavior of a *doTest* involving the patient p, a nurse n, a doctor dr, and the data logger dl. The corresponding tool input model written in Umple syntax is shown in Listing 7.2.

The $doTest\_S$ (see line 2 in Listing 7.2) state machine has a strict sequence type and involves four composite states: $Testing\_W$, $BloodTest\_S$, $UltraSound\_S$, and $CheckResult\_S$, and several simple states like $patientReady\_p$, $prepareSystem\_n$, and $Query\_dr$. For all composite states, we showed the type of sequencing after "_" as mentioned before.

```
1  class A {
2    doTest_S {
3      Testing_W{
4         [true] -> CheckResult_S;
5          BloodTest_S {
6              [true]-> UltraSound_S ;
7              patientReady_p {
8              -> doBloodTest_n;
9              }
10             doBloodTest_n {
11             -> logBloodResult_dl;
12             }
13             final  logBloodResult_dl{
14             }
15         }
16         UltraSound_S {
17            prepareSystem_n
18            {
19             -> patientReadyForImage_p;
20            }
21            patientReadyForImage_p{
22                 -> doUltraSound_n;
23             }
```

```
24              doUltraSound_n {
25                  -> logResult_dl;
26              }
27              final  logResult_dl{
28              }
29          }
30      }
31      CheckResult_S {
32          Query_dr {
33               -> Report_dl;
34          }
35          final  Report_dl{
36          }
37      }
38  }}
```

Listing 7.2: The do test example of Figure 7.3 written in Umple syntax

7.2 Extension of the Umple Meta-Model

We extended the Umple state machine meta-model to include the features which are required for modeling the global requirements model using HSM. Listing 7.3 describes the extended **State** class, it includes the following new attributes: $initRoles$, $terRoles$, and $partRoles$, which are used to store the initiating roles (or IR), terminating roles (or TR), and participating roles (or PR), respectively. For a simple state, the role that executes the do action is initiating, terminating, and the participating role at the same time. However, a composite state has in general different sets of initiating, terminating, and participating roles. We will discuss this in detail in the next sections.

The "$flowMessages$", and "$cimMessages$" attributes in Listing 7.3 are used to store the coordination messages which are needed during the implementation of the derivation algorithm. We will describe why these messages are required (see Section 7.4), and will discuss how they are calculated by the tool, and how they are used. The new extended

attributes in the **StateMachine** class meta-model is shown in Listing 7.4, we added the "*smSequencingType*" attribute to store the state machine sequencing type.

```
class State
{
    depend cruise.umple.parser.Position;
    isA Node;
    name;
    Boolean isConcurrent = { numberOfNestedStateMachines() > 1 }
    Set<String> initRoles = new HashSet<String>(); // new attribute
    Set<String> terRoles = new HashSet<String>();  // new attribute
    Set<String> partRoles = new HashSet<String>(); // new attribute
    List<String[]> flowMessages = new ArrayList<String[]>(); // new
    attribute
    List<String[]> cimMessages = new ArrayList<String[]>(); // new
    attribute

    1 -- * Activity;
    0..1 -> * Action;
    * -- 1 StateMachine;
```

Listing 7.3: Snippet from the State class in the Umple meta-model

```
class StateMachine
{
    depend cruise.umple.util.*;
    Integer recentSearchDepth = -1;
    name;
    Boolean containsHistoryState = false;
    Boolean containsDeepHistoryState = false;
    String smSequencingType = ""; // new attribute

    * -- 0..1 UmpleClass;

    * -- 0..1 UmpleTrait;
```

Listing 7.4: Snippet from the StateMachine class in Umple meta-model

7.3 Calculation of Initiating and Terminating Roles

The initiating, terminating, and participating roles are important for the derivation algorithm [4] and in the tool as well. These roles are needed for the calculation of coordination messages which are required in the distributed design models. In this section, we discuss how these roles are calculated in our tool. The tool calculates the initiating (IR), participating (PR) and terminating roles (TR) for each state, and stores them in the attributes *initRoles*, *partRoles* and *terRoles* mentioned above. These roles are calculated using the function *calculateStartingEndingRoles(StateMachine sm)*; this function is called recursively for each composite state. The simple state is associated with one role, and it is simultaneously operating as initiating, terminating, and participating role. For the composite states, these aforementioned roles types are calculated based on the rules presented in Figure 5.3 (taken from Table 2 in [4]).

The rules in Figure 5.3 describe how the sets of *initiating* (also called starting roles), *participating*, and *terminating* roles, are calculated in any global model (whatever the global model used, such as: collaboration, activity diagram, PO-charts, or state machine), and based on the sequencing operators used. It can be used to calculate these roles for a state machine or a composite state. The function *calculateStartingEndingRoles*() is called recursively from the upper parent state going down to the bottom simple states. The calculation of the roles is starting from the bottom up to the top.

Consider a composite state CS_n containing a strict sequence between CS_1 and CS_2, based on the rules as defined in Figure 5.3, the initiating role for CS_n or $IR_{CSn} = IR_{CS1}$, the Terminating role $TR_{CSn} = TR_{CS2}$, $PR_{CSn} = PR_{CS1} \cup PR_{CS21}$.

Figure 7.4 presents two examples of state machines with different sequencing types. The [true] guard inside the state CS_3_S in Figure 7.4 (a) should be on the transition between the composite states CS_2_S and CS_3_S and it was not placed correctly by Umple Online (the same problem occurs in Figure 7.4 (b) and other examples in this chapter). The global state machine sm_7_S in (a) is strictly sequenced and contains three states, s_1, composite state CS_1, and s_6. The composite state CS_1_S has another state machine inside with a strict sequence between CS_2 and CS_3. For the state CS_1, $IR_{CS_1} = IR_{CS_2} = \{r_1\}$,

$PR_{CS_1} = (PR_{CS_2} \cup PR_{CS_3}) = \{r_1, r_2, r_3\}$, and $TR_{CS_1} = TR_{CS_3} = \{r_3\}$.

For the state machine sm_8_S in Figure 7.4(b), the state CS_1 is weakly composed (CS_2 ;$_w$ CS_3). Because of weak sequence and based on the rules above, $IR_{CS_1} = IR_{CS_2} \cup (IR_{CS_3} - PR_{CS_2})$, $IR_{CS_2} = \{r4\}$ since CS_2 has a strict sequence, $IR_{CS_3} = \{r_2, r_3\}$ since CS_3 has a weak sequence, and $PR_{CS_2} = \{r_4, r_1, r_2\}$, then $IR_{CS_1} = \{r_4, r_3\}$. $TR_{CS_1} = TR_{CS_3} \cup (TR_{CS_2} - PR_{CS_3}) = \{r_2, r_3\} \cup \{\{r2\} - \{r_3, r_2\}\} = \{r_2, r_3\}$, $PR_{CS_1} = \{r_1, r_2, r_3, and\ r_4\}$.

Another example of a composite state which contains concurrent state machines (see sm_9_S) is shown in Figure 7.5. All state machines in CS_1 have weak sequence type, while they have a strict type in CS_2. The initiating roles of a composite state that contains concurrency is the union of the initiating roles of each state machine inside the composite state as described in the table above (see Figure 5.3), and this rule is applied for the terminating roles as well. The $IR_{CS_1} = \{r_2, r_3\} \cup \{r_1, r_2\} = \{r_1, r_2, r_3\}$. $IR_{CS_2} = \{r_2\} \cup \{r_1\} = \{r_1, r_2\}$, $TR_{CS_1} = \{r_2, r_3\} \cup \{r_1, r_2\} = \{r_1, r_2, r_3\}$, $TR_{CS_2} = \{r_8\} \cup \{r_5\} = \{r_8, r_5\}$.

In Java, since IR, PR, and TR are sets, we implemented the union between sets using the $addAll(set)$ function, for example $IR_{CS_1} \cup IR_{CS_2}$ is written in Java as $IR_{CS_1}.addAll(IR_{CS_2})$. The set difference is implemented using the $removeAll(set)$ function, for example $IR(CS1) - IR(CS2)$ is written in Java as $IR_{CS_1}.removeAll(IR_{CS_2})$.

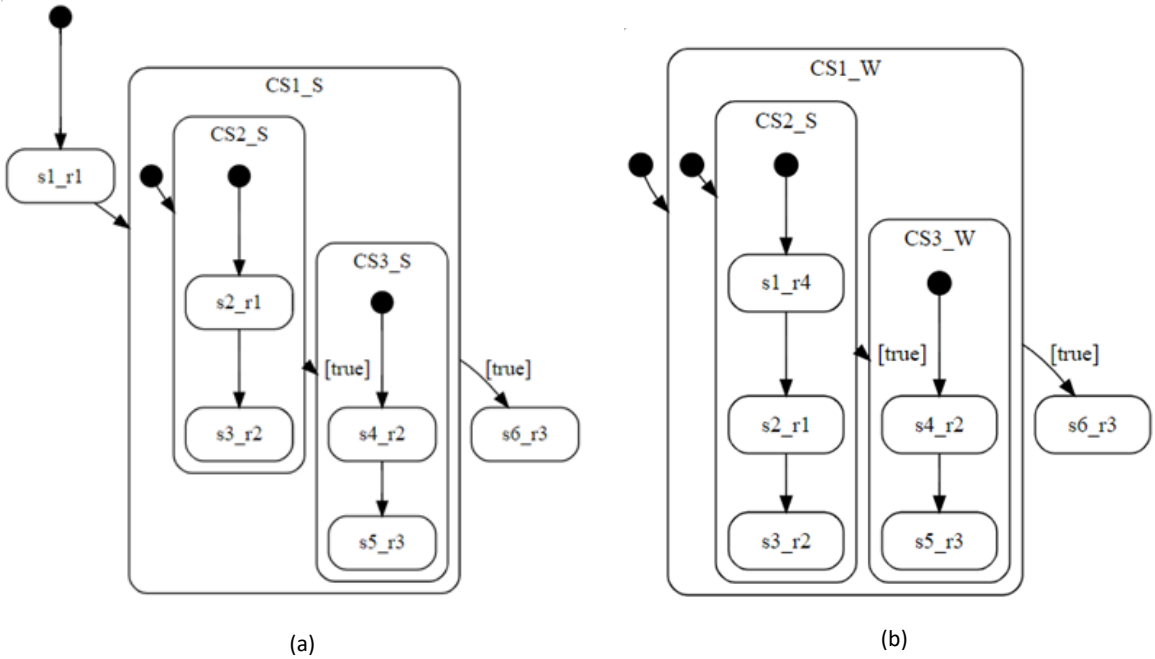


Figure 7.4: Examples of a state machines with different types of sequencing (a) sm_7_S, (b) sm_8_S.

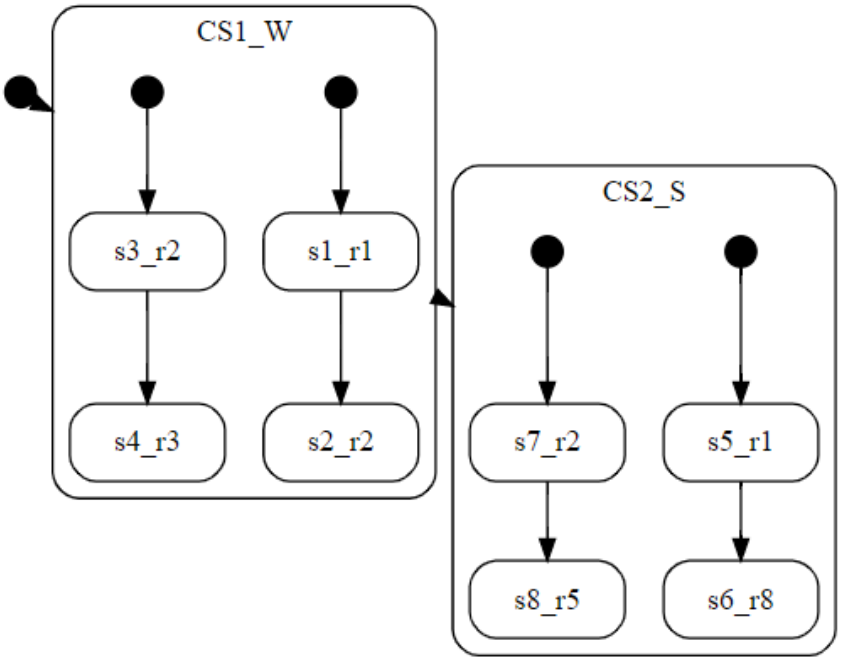


Figure 7.5: State machine sm_9_S with concurrent composite states.

7.4 Calculation of Coordination Messages

The derivation algorithm, which the tool implements (explained in Section 5.2), introduces flow messages (if required) for the coordination of actions related by a strict sequence as proposed in [8, 62]. It also introduces a choice indication message (*cim*) to inform a role that does not participate in a given alternative when this alternative is chosen. For example, flow messages were introduced for a strict sequence $CS_1 \;;_s CS_2$ to ensure that all actions in composite state CS_1 are completed before any action in CS_2 can start. Also, since these flow messages must be distinguished to avoid ambiguities, we solve this by using different message names distinguished by numbers.

In this tool, we calculate the required coordination messages before starting the behavior transformation (derivation of the design models). This is required to ensure consistent message names between the roles which send and receive the messages. Consider a strict sequence between the states $s_1\_r_1$ and $s_2\_r_2$ (written as $s_1\_r_1 \;;s\; s_2\_r_2$), in the design model of the role r_1, a message fm_x must be sent to r_2, and at the role r_2, the same flow message (with the same number) must be received.

The calculated flow and *cim* messages are stored in the lists $flowMessages$ and $cimMessages$, respectively, at the sending state object of the global model (see Listing 7.3)

7.4.1 Flow Messages for a Strict Sequence

In each state machine or each composite state, and in the case of a strict sequence, we calculate the required flow messages between states. Namely, the coordination messages that must be sent from the terminating roles (TR) of a given state to the initiating roles (IR) of the next state. Each entry in the $flowMessages$ list (See Listing 7.3) contains three elements: the role that sends the message, the role that receives it, and the flow message number. The following entry {"r_x", "r_y", "fm_1"} (stored at the sending state which is executed by the role r_x) means: the message fm_1 should be sent from r_x to r_y. Calculating the flow messages before deriving the design model is important for the consistency and correctness of the coordination messages.

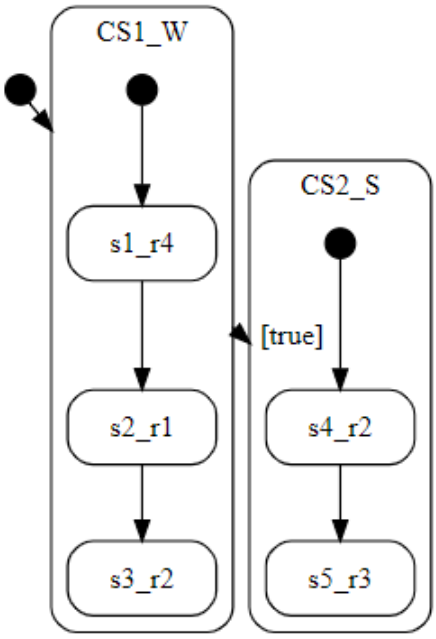


Figure 7.6: State machine sm_{10}_S.

For example, in the state machine sm_{10} (see Figure 7.6), a strict sequence exists between CS_1 and CS_2 and a coordination message must be sent from TR_{CS_1} to the IR_{CS_2}. The $TR_{CS_1} = \{r_1,\ r_2,\ r_4\}$, and $IR_{CS_2} = \{r_2\}$. Therefore, the following flow messages are stored at CS_1 : {"r_1", "r_2", "fm_1"} and {"r_4", "r_2", "fm_2"}.

7.4.2 Choice Indication Message (cim)

The choice indication message (*cim*) is needed if a given role (other than the role which made the choice) does not participate in one of the alternative branches and this branch is chosen (for instance, the role r_3 does not participate in the left branch in sm_{11} in Figure 7.7(a)), a *cim* message must be received at that role, and this message is sent by the role which made the choice (r_1 in this example). Consider the design model for r_3 in Figure 7.7(b), since r_3 does not participate in the left branch, cim_1 is received to inform r_3 if this branch is selected. We calculate the required *cim* messages at each branch and store them in the $cimMessages$ list in the *state* object.

Table 7.1 shows the *cim* messages stored at s_1 in sm_{11}. The "Next State" column in

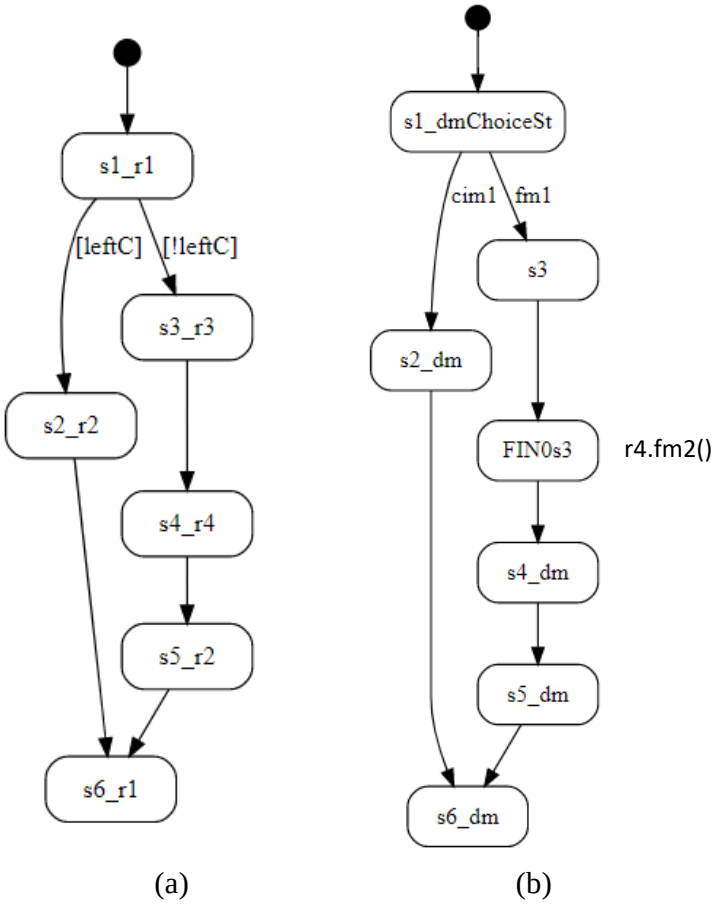


Figure 7.7: (a) Sm_{11}_S (state machine with a choice) (b) Design model for r_3 in Sm_{11}, where "dm" in the state names means dummy state

the table is used to indicate for which branch this message is needed (*e.g.* the next state s_2 means that the left branch is executed, and s_3 means the right one).

Another example is shown in Figure 7.8 (see state machine sm_{12}). The following choice indication messages (see, for instance, Table 7.2) are stored at the choice state s_1 in the global model (sm_{12}). As we see from the table, the message cim_1 is stored for r_5 in the right branch since it does not participate in that branch. The same for the role r_4, which does not participate in the left branch and should receive cim_3. The purpose of the messages $cimC_0$ and $cimC_1$ will be explained later in Section 7.5.5.2.

The following flow messages (see Table 7.3) are stored at s_1, the flow message are needed because of the strict sequence after the choice. Flow messages must be sent from

To	Message number	Next State
r_3	cim_1	s_2
r_4	cim_2	s_2

Table 7.1: The *cim* messages stored at the state s_1 in sm_{11}

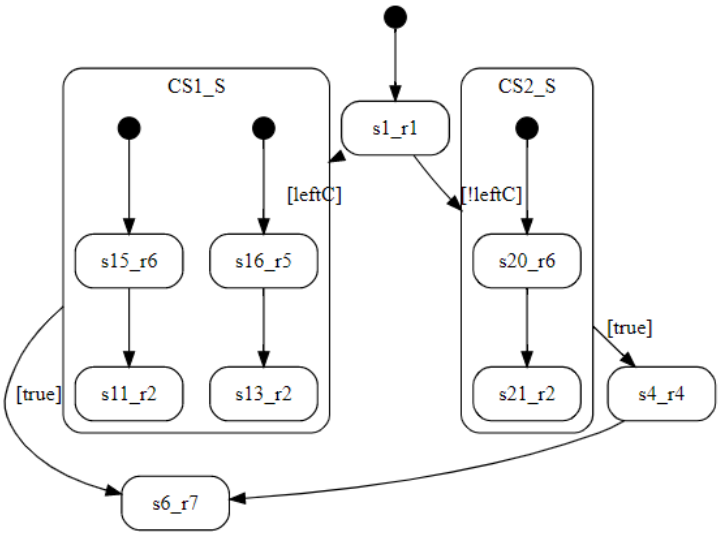


Figure 7.8: Sm_{12}_S (state machine with choice and composite states in the alternatives).

To	Message number	Next State
r_2	$cimC_0$	CS_1
r_2	$cimC_1$	CS_2
r_5	cim_1	CS_2
r_4	cim_3	CS_1

Table 7.2: The *cim* and *cimC* messages stored at the state s_1 in sm_{12}

To	Message number	Next State
r_5	fm_0	CS_1
r_6	fm_1	CS_1
r_6	fm_2	CS_2

Table 7.3: The flow messages stored at the state s_1 in sm_{12}

the choice role "r_1" to the initiating roles of the composite states after the choice.

7.5 Generation of the Design Model

The tool implements the derivation algorithm for the design model from a global model written in the extended HSM notation explained Chapter 5. The function $calculateDM(sm, r_n)$ takes the global state machine sm as input and generates a design model for the role r_n. In the next section, we discuss the tool implementation for different types of states and sequencing operators.

7.5.1 Simple and Composite States

For each simple state, the tool keeps the state as is if the role r_n participates in that state. Otherwise, the simple state is replaced by a *dummy* state (the state $s_n\_r_n$ is written as s_n_dm when it becomes *dummy*). For each composite state, the function $calculateDM$ is called recursively.

7.5.2 Weak Sequence

As mentioned before, the distributed design model for weakly sequenced behavior does not need any coordination messages and is only based on the local sequencing by each role. In the case of weak sequence, the function $calculateDM$ operates as follows. It will generate a design model that has the same structure as the global state machine model, and the rules explained in the previous section (see Section 7.5.1) for the simple and composite states will be applied. This corresponds to the approach *design model generation by projection* (see [4, 2, 27, 62]).

7.5.3 Strict Sequence

For the derivation of a global model with a strict sequence, coordination messages are used to ensure the coordination of actions as described above. Consider a strict sequence between the composite states CS_1 and CS_2, a coordination message must be sent from TR_{CS_1} to IR_{CS_2}. More precisely, in the design model for r_n, a coordination message must be sent

from r_n to all roles in IR_{CS_2}, except to r_n itself if $r_n \in TR_{CS_1}$. For this purpose, a new state called "FINCS_1" is created, and the sending of coordination messages is implemented as a do-action in that state.

For the reception of coordination messages at CS_2, a flow message must be received by r_n from all roles in TR_{CS_1} except for r_n itself if $r_n \in IR_{CS_2}$. For this purpose, a new state called "PRELCS_2" is created, which has an outgoing transition that is triggered by the reception of one of the flow messages to be received. If more than one flow message is to be received, an additional intermediate state is created for each additional flow message with an outgoing transition to receive the corresponding flow message. These additional states are linked sequentially.

As an example, consider the state machine sm_{13} (with strict sequence type "sm_{13}_S") in Figure 7.9(a). It has the following embedded composite states: CS_1 is a composite state with strict sequence (CS_2 ;$_s$ CS_3), CS_2 (has weak sequence), and CS_3 (strict sequence). For weak sequence like the state CS_2, the tool keeps the state if the role participates in that state, and otherwise replaces it by a dummy state (as we mentioned before). For the strict sequence, flow messages are introduced (if required).

Figure 7.9(b) shows the design model for r_1 in sm_{13}. The states where r_1 participates are kept, (like state s_2), and the other states become dummy (or _dm). The FIN states like FIN0CS_2 are used to send the flow messages using the do action. The do-action in this state is "*do* $\{r_3.fm_3();\}$" which represents the sending of the flow message "fm_3" from r_1 ($r_1 \in TR_{CS_2}$) to r_3 ($r_3 \in IR_{CS_3}$).

The same rules are applied in the design model for r_2 in Figure 7.9(c). The do-action for "FIN0CS_2" is "*do* $\{r_3.fm_1();\}$", the message "fm_1" is sent from r_2 ($r_2 \in TR_{CS_2}$) to r_3 ($r_3 \in IR_{CS_3}$). The do-action for the state FIN0CS_1 (this state is located after the composite state CS_1 and before the simple state s_6) is "*do*$\{r_3.fm_0();\}$", the message "fm_0" is sent from r_2 ($r_2 \in TR_{CS_1}$) to r_3.

In the design model for the role r_2 in Figure 7.9(c), because of a strict sequence in CS_3 between s_4 and s_5, r_2 must receive a flow message from r_3. For this purpose, the "PREL0s_5" is created containing no do-action but having an outgoing transition, which is

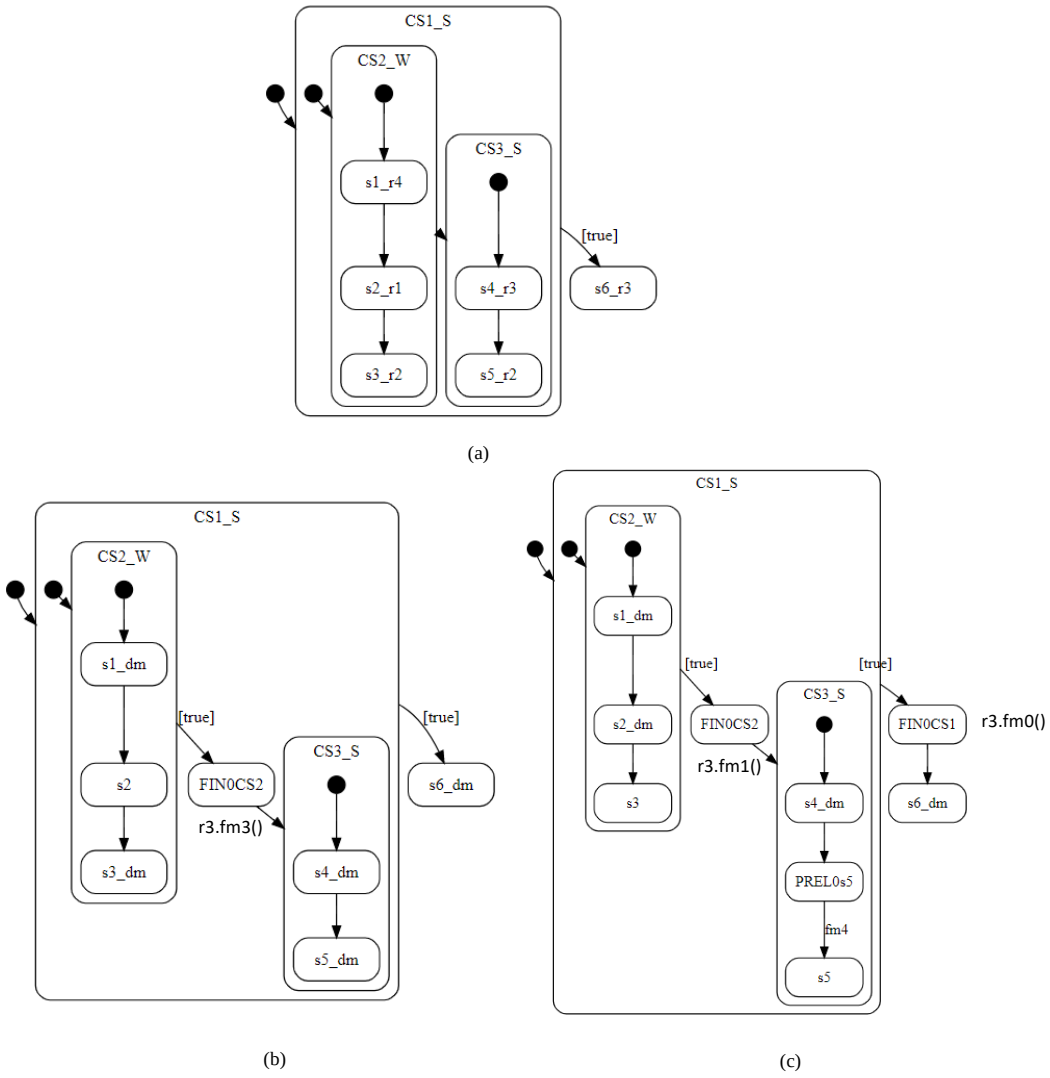
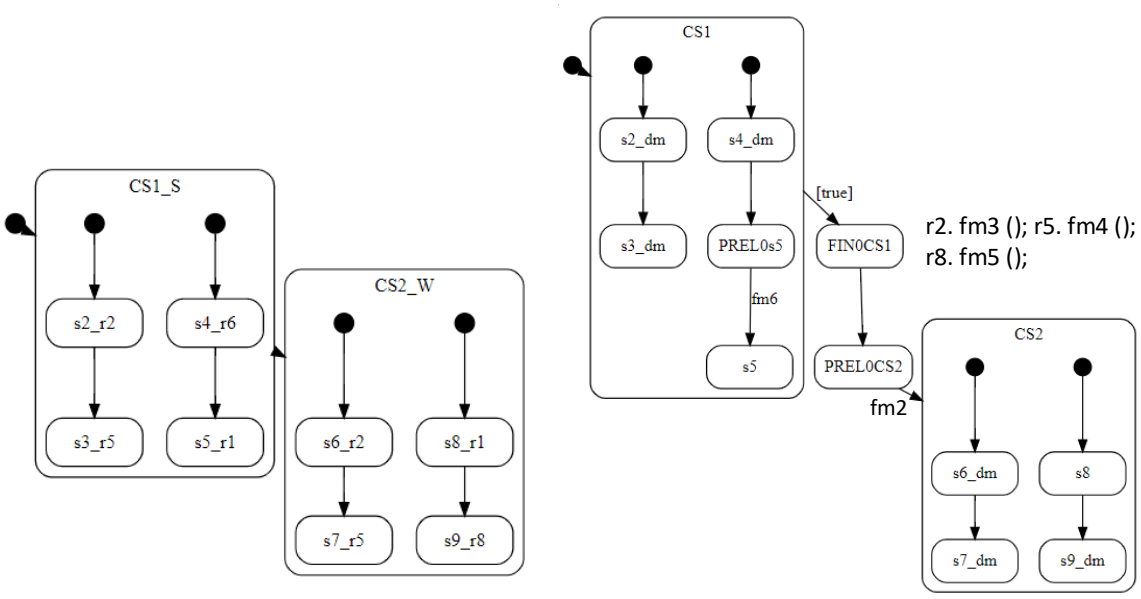


Figure 7.9: (a) Global model sm_{13}_S, (b) Design models for r_1, (c) Design models for r_2 , in sm_{13}.



(a) state machine sm_{14}_S (b) design model r_1

Figure 7.10: (a) Example of a global model with concurrency, (b) Design model for r_1 in (a)

triggered by the reception of the flow message "fm_4".

7.5.4 Concurrency

As mentioned before, the generation of a design model for a composite state that contains several concurrent sub-states is obtained by projection without any additional coordination messages since there is no interaction between the different concurrent sub-states. The design model for such a composite state is constructed by creating a composite state that contains the design model for the given role r_n for each concurrent sub-state in the global model.

As an example, consider the following state machine sm_{14} in Figure 7.10(a) (the guard [true] should be on the transition between CS_1_S and CS_2_W), which contains two composite states with concurrency. As we said in the restriction of Umple state machine (see Section 7.1), the sequence type of a composite state which contains concurrency will be applied to all concurrent state machines inside. Therefore, the two state machines in CS_1_S

```
1    FIN0CS1 {
2       do { r2.fm3();r5.fm4();r8.fm5(); } -> PREL0CS2;
3     }
4     PREL0CS2 {
5       fm2 -> CS2;
6     }
```

Listing 7.5: The Umple code for the generated new states in the design model of r_1 in Figure 7.10(b)

(since CS_1 has a strict sequence) have a strict sequence, and the other two in CS_3_W have a weak sequence.

The design models for r_1 is shown in Figure 7.10(b). The same derivation rules described in the previous section for the weak and strict sequences are applied here. The state "FIN0CS_1" implements the sending of the flow messages from the r_1 in CS_1 to IR_{CS_2} (except r_1), while the state "PREL0CS_2" is followed by the reception of the flow message "fm_2" which is sent by the TR_{CS_1} $\{r_5\}$ (see for instance Listing 7.5). The flow message "fm_2" was not placed on the correct transition by Umple Online and should be on the transition between "PREL0CS_2" and CS_2.

7.5.5 Alternatives

In this section, we will discuss the generation for the global model that contains a choice state with alternatives. As we said before in Section 5.2, the derivation of strict sequence can also be applied in the case of alternatives where a choice state may have two or more outgoing transitions.

As we said in the assumptions in Section 7.1, all alternatives in a given choice must end together in the same state, this is needed to ensure that the composite state which contains the alternatives has one set of terminating roles. If alternatives associated with the choice would have different final states, this means, that the composite state would have different sets of terminating roles which would make the derivation of the design model complicated.

Consider the state machine sm_{15} in Figure 7.11(a) where s_1 is the choice state, and r_1 is the choice role. The generated design model for r_1 (written in Umple) is shown

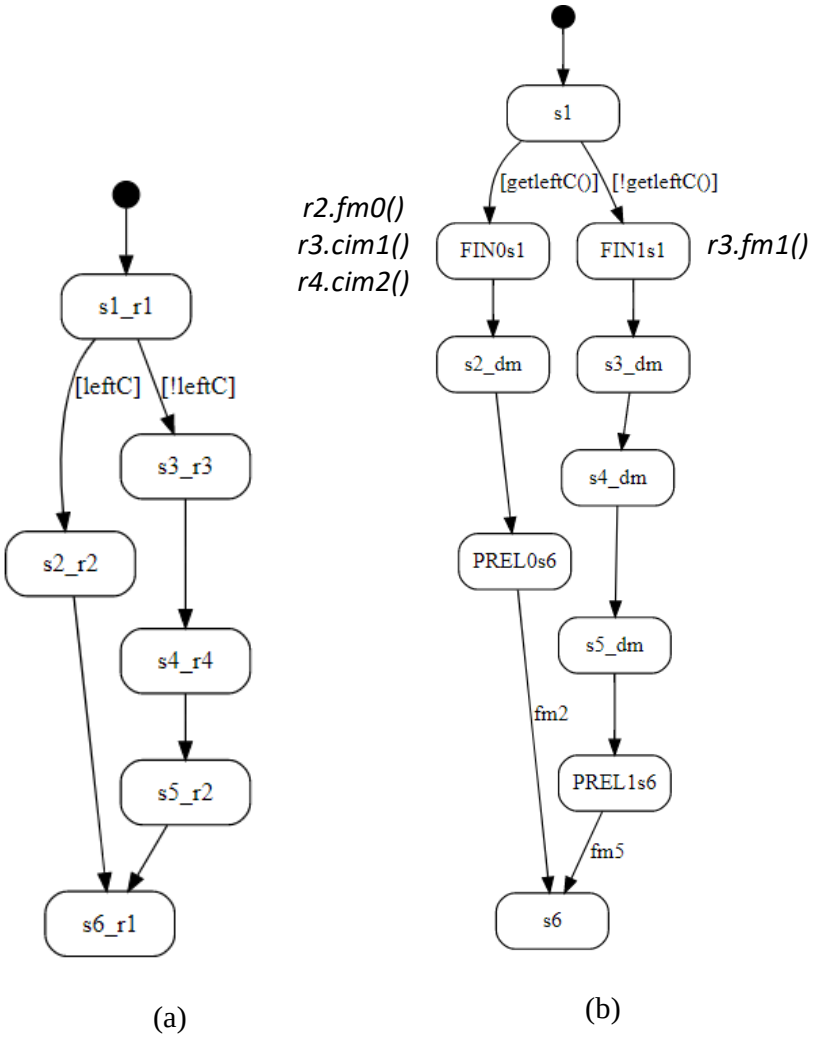


Figure 7.11: (a) State machine sm_{15}_S, (b) Design model for r_1 in (a).

```
class A
{
  boolean flag = true;
  Sm_15 {
    s1 {
      [getLeftC()]  -> FIN0s1;
      [!getLeftC()]  -> FIN1s1;
    }
    FIN0s1 {
      do { r2.fm0();r3.cim1();r4.cim2(); } -> s2_dm;
    }
    FIN1s1 {
      do { r3.fm1();} -> s3_dm;
    }
    s2_dm {
       -> PREL0s6;
    }
    PREL0s6 {
      fm2 -> s6;
    }
    s3_dm {
       -> s4_dm;
    }
    s4_dm {
       -> s5_dm;
    }
    s5_dm {
       -> PREL1s6;
    }
    PREL1s6 {
      fm5 -> s6;
    }
    final s6 {    }
  }
}
```

Listing 7.6: Design model for r_1 in sm_{15} written in Umple

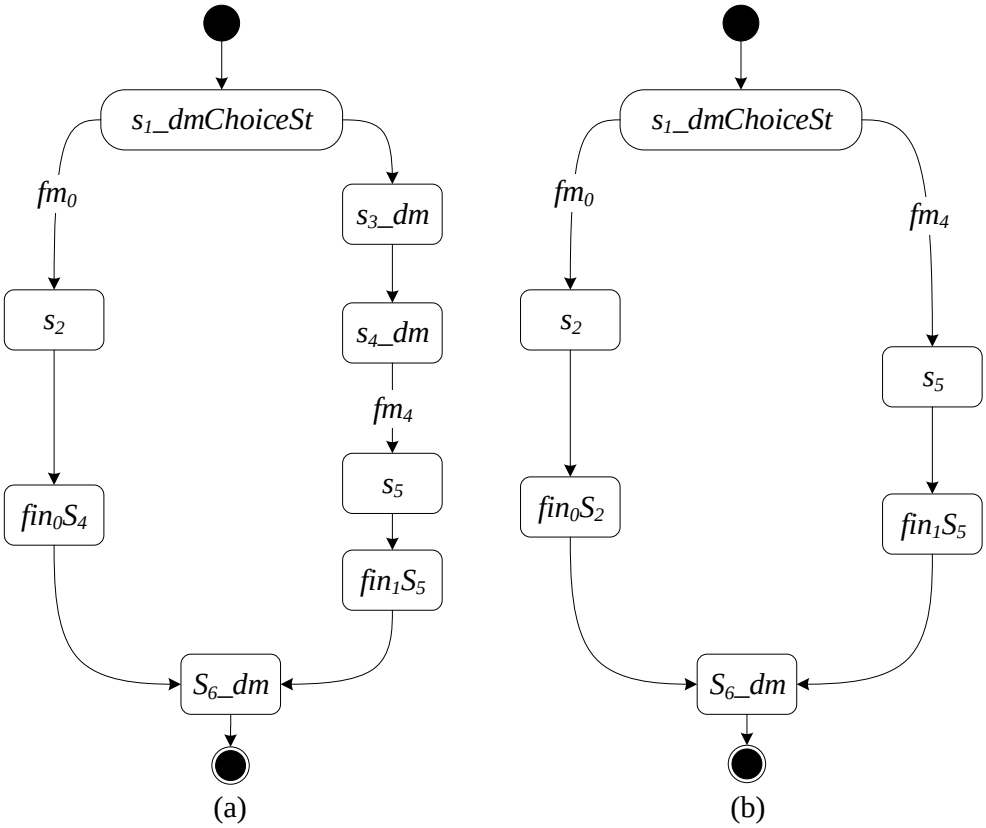


Figure 7.12: (a) Wrong design model, (b) Correct design model, for the role r_2.

in Listing 7.6, and the corresponding state machine (graphical representation generated by Umple Online) is shown in Figure 7.11(b). As we see in the state "FIN0s_1" in the left branch, the following coordination messages are sent (see the do-action for the state FIN0s_1 in Listing 7.6): fm_0 to r_2 because of strict sequence, and the choice indication messages: cim_1 and cim_2 to r_3 and r_4 respectively, since they are not involved in the left branch, and to inform these roles if this branch is selected.

7.5.5.1 Dummy Choice State Problem

It is important to note a problem with choice propagation related to dummy choice states (also called "inactive choice states" [77]). In the design model derivation for a given role r_n, the derivation algorithm, using the projection approach, generates a dummy state for

the states executed by another role r ($r \neq r_n$). If the dummy state is a choice state, the algorithm generates a "dummy choice state" (our tool uses the name "dmChoiceSt" for short). The dummy choice state must be followed by a reception of a message for each outgoing transition in the design model. Therefore, all dummy states after the dummy choice state will be deleted until a transition with a message reception is detected.

Consider the derivation of the design model for r_2 in sm_{15} (the global model sm_{15} is shown in Figure 7.11(a)), we have a dummy choice state since r_2 follows the choice made by r_1 (see the design model for r_2 in Figure 7.12(a)). As we said in Section 5.2, a dummy choice state must be followed by a reception of a message at each outgoing transition. There is no message reception (between $s_1\_dmChoiceSt$ and s_3_dm states) in the right alternative. We solve this problem by eliminating all dummy states until we get a transition with message reception, see, for instance, the correct design model for r_2 in Figure 7.12(b).

The second problem happens when a given role does not participate in some alternatives. The design model of r_3 (see Figure 7.7(b) in Section 7.4.2) is an example, and this problem is solved using a *cim* message as discussed before.

7.5.5.2 The Dummy Choice State Followed by Composite States

Another problem that should be addressed occurs when an dummy choice state, is followed by a composite state. Consider the global model represented by sm_{16} in Figure 7.13(a).

The design model for r_2 in Figure 7.13(b) is invalid since the dummy state is not followed by a reception of a message in the left branch. The solution for such a problem is to add a message reception for the transition of the left branch after the dummy choice (note that the message cim_1 is needed because r_2 did not participate in the right branch). We have two possibilities, either add the reception of the flow message "fm_1" as an event at this transition and remove it from CS_2 since it is preceded by a dummy state, or do the same but for the message "fm_2", see for instance $solution_1$ in Figure 7.14. It is not easy to derive such a solution automatically because of different possibilities.

Another solution is to wait for both flow messages: fm_1 and fm_2, then start the concurrent executions in CS_2 (see, for instance, $solution_2$ in Figure 7.15). This solution

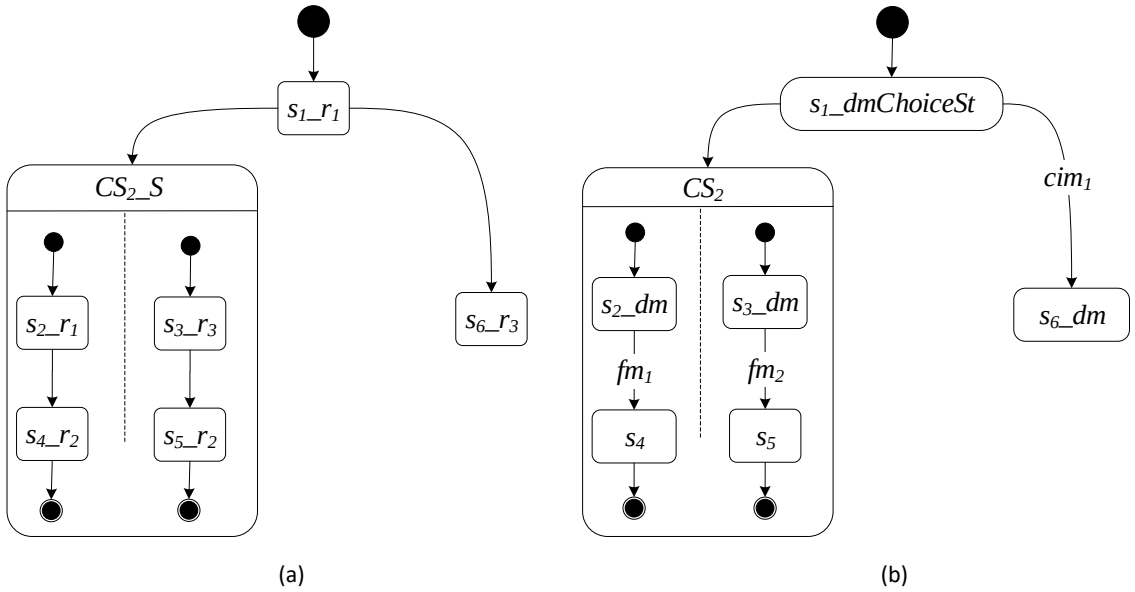


Figure 7.13: (a) State machine sm_{16}, (b) The design model for r_2 in sm_{16}.

could be slower because both messages need to be received for the composite state to start, and this is not necessary. However, it is easy to implement because of the message reception pool [37] supported by Umple. It allows waiting for all required messages to be received before starting the concurrent composite state.

The easiest solution is to receive a $cimC$ message (choice indication message for composite states) by the role for which we derive the design model for (r_2 in this example) and which is involved in a composite state after the choice. The reception of $cimC$ will solve the problem after the dummy choice state. We have implemented this approach, see for instance $solution_3$ in Figure 7.16. The $cimC_1$ message in the figure is sent by the role which made the choice. This solution could involves in general extra messages, but it is easy to derive.

Table 7.2 above shows the required $cimC$ messages at the choice state s_1 in the global model sm_{12}. The messages $cimC_0$ and $cimC_1$ are required by the role r_2 in both branches since it participates in composite states after the choice (and it is not an initiating role for that states)

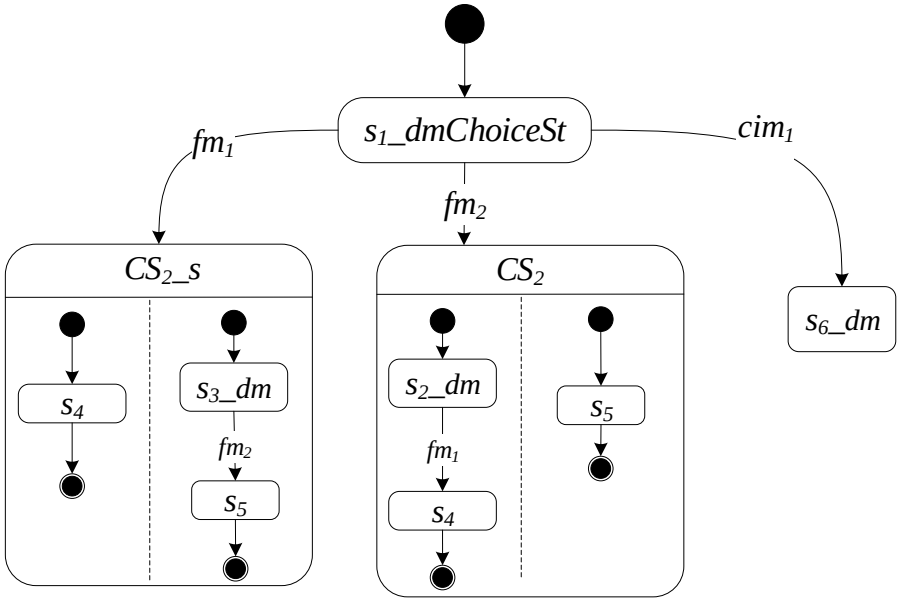


Figure 7.14: $Solution_1$: design model for r_2 in sm_{16} (Figure 7.13(a)).

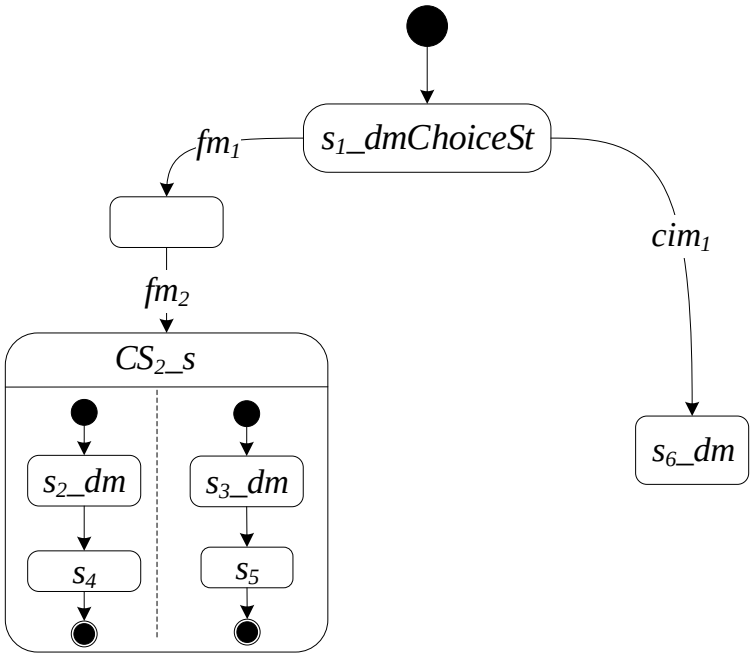


Figure 7.15: $Solution_2$: design model for r_2 in sm_{16} (Figure 7.13(a)).

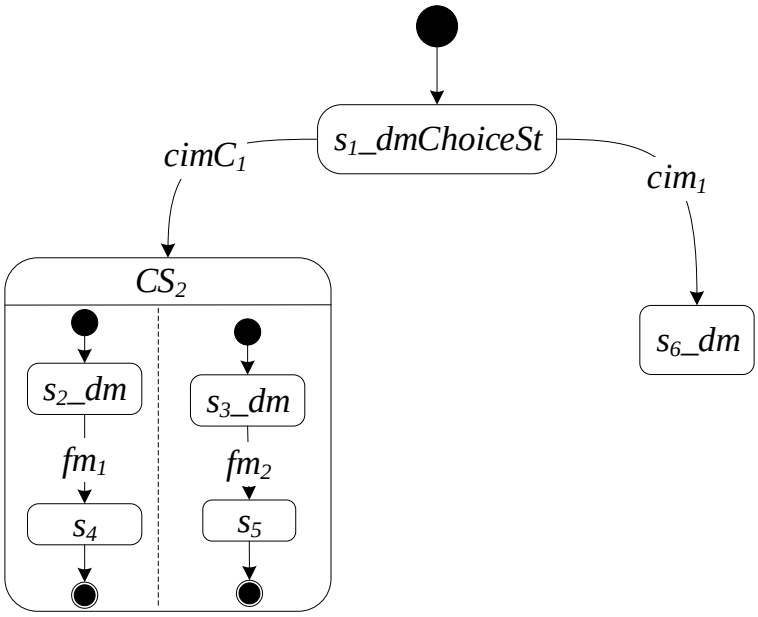


Figure 7.16: $Solution_3$: design model for r_2 in sm_{16} (Figure 7.13(a)).

7.5.5.3 Example for the Derivation of the Choice

Having discussed the problems and assumptions related to the derivation of the design models including a choice, we want in the following to show an example and discuss the design models for the roles r_1 and r_2 in the global model sm_{12} above (see, for instance, Figure 7.8 in Section 7.4.2). The design model for the choice role r_1 is presented in Figure 7.17; the do-actions in the fin states FIN0s_1 and FIN1s_1 represent the sending of coordination messages (flow and choice indication messages). The do-action in FIN0s_1 in the left branch is the following: "$do\ \{r_5.fm_0();\ r_6.fm_1();\ r_2.cimC_0();\ r_4.cim_3();\}$", where the flow messages fm_0 and fm_1 in the do-action are sent to the initiating roles of the next state CS_1 (of the global model sm_{12} above); which are r_5 and r_6 respectively, the message $cimC_0$ is sent to r_2 since it is part of a composite state and message cim_3 to r4 since it is not participating in this branch. The do action for FIN1s_1 in the right branch is "$do\ \{r_6.fm_2();\ r_2.cimC_1();\ r_5.cim_1();\}$".

The design model for r_2 in sm_{12} above is shown in Figure 7.18. Since s_1 is a dummy choice state (see the state "s1_dmChState"), and since r_2 participates in a composite state

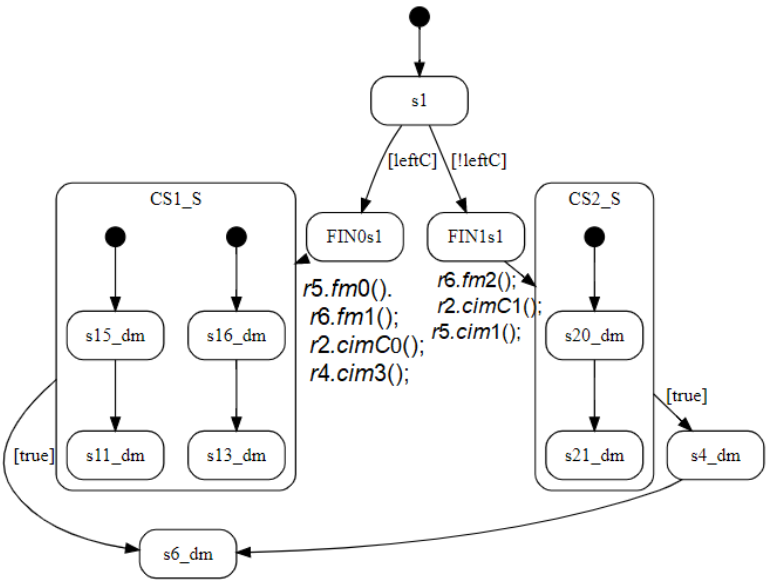


Figure 7.17: Design model for r_1 in the global model sm_{12} (Figure 7.8).

in both alternatives, a *cim* message of type C must be received in both branches. See the reception of the messages $cimC_0$ and $cimC_0$ in the left and right branches, respectively.

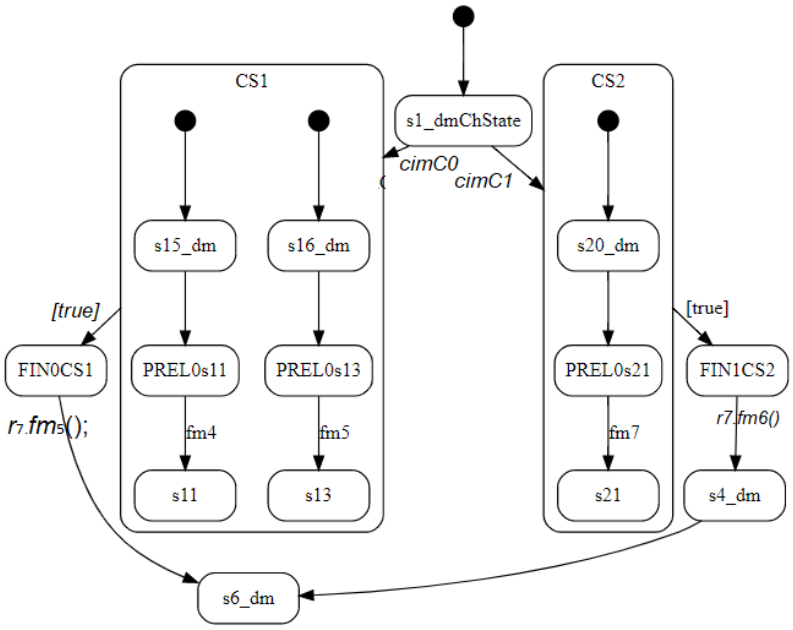


Figure 7.18: Design model for r_2 in the global model sm_{12} (Figure 7.8).

7.5.6 Strict While Loop

The global model in Figure 7.2(e) before (see Section 7.1) is an example of a strict while loop, where r_1 is the loop initiating role, CS_2 represents the loop body, CS_3 is the follow-up, and they are connected by strict sequence. The derivation of a design model for the strict loop is recursive without any additional coordination messages since it consists of alternatives and strict sequence, it is sufficient to apply the rules explained in the previous sections to the occurrences of strict sequences and the alternative which make up the strict while loop.

As an example, consider the design model for the loop initiating role r_1 in Figure 7.2(e) (Section 7.1), the design model for r_1 is presented in Figure 7.19(a). The role r_1 is responsible for sending the coordination message to the roles of the composite states CS_2 and CS_3 through the do-actions of the states FIN0s_1 and FIN1s_1, respectively, see for instance, Listing 7.7). In addition, on the transition between the states CS_2 and the initial state s_1, we see the reception of the flow message fm_0, this message is received by r_1 and sent by the role r_3 (where r_3 is the terminating role for CS_2).

The design model for r_2 is presented in Figure 7.19(b). r_2 receives the following coordination messages: the choice indication message of type C ($cimC_0$) is received because r_2 is part of the composite state CS_2, and cim_0 is received because r_2 did not participate in CS_3.

```
// between s1 and CS2
 FIN0s1 {
      do { r2.cimC0();r3.cimC1(); } -> CS2_S;
    }
// between s1 and CS3
 FIN1s1 {
      do { r3.cimC2();r2.cim0(); } -> CS3_S;
    }
```

Listing 7.7: The do actions for some states in the design model of r_1 Figure 7.19(a)

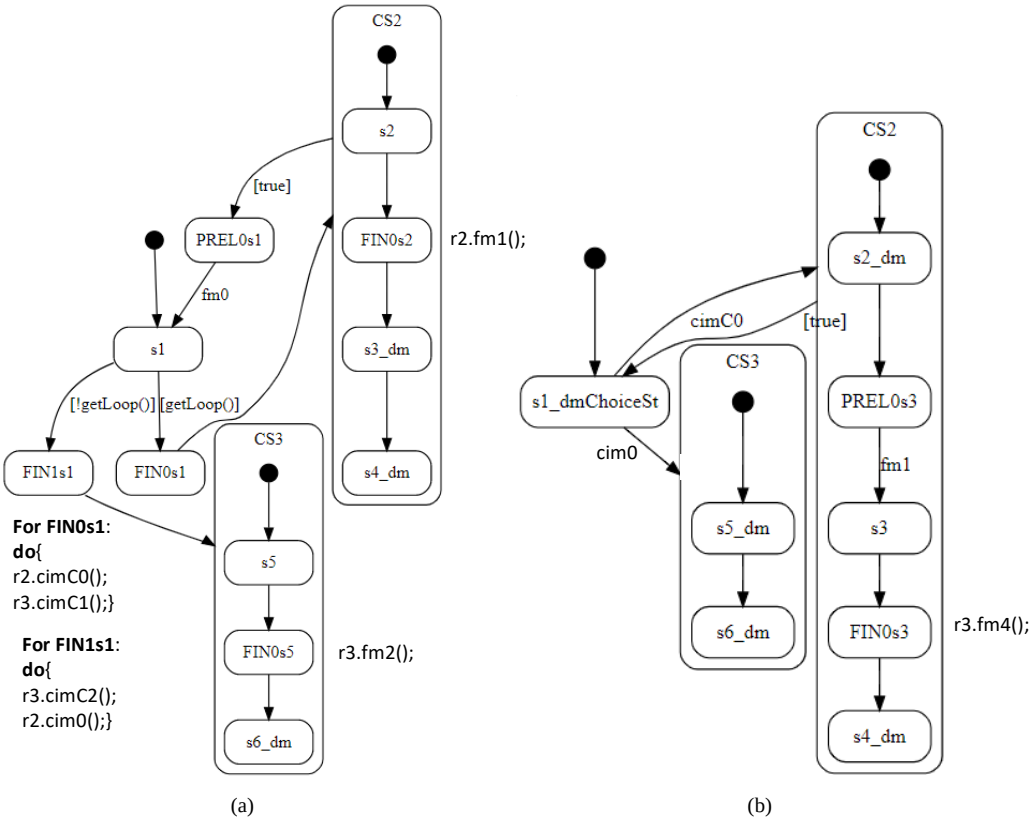


Figure 7.19: Design models for: (a) r_1, (b) r_2, in Figure 7.2(e).

7.5.7 Weak While Loop

The tool supports the automatic generation of a state machine that contains a weak while loop; it implements the derivation algorithm for the weak while loop (see for instance Section 5.2). The global model in Figure 7.2(f) above (see Section 7.1) is an example of a weak while loop, where r_1 is the loop initiating role, CS_2 represents the loop body, CS_3 is the follow-up part. As we said, the suffix "WL" in the state machine or composite state name is used for the detection of a weak while loop. Also, the suffixes "Body" and "Fup" in the composite states are used to identify the loop body and follow-up parts, respectively.

A weak while loop is a special kind of a composite state, and it has two sequencing types. The transitions between the choice state and the body and follow-up states are strict sequences. However, the transition from the Body state back to the initial state is a weak sequence. We assume that the composite state of a weak while loop includes two composite states, Body and Follow-up, and both of them of strict type. The assumption that they are strict sequences is not necessary since both states could have a weak sequence type or being a while loop. However, the tool forces the suffixes "Body" and "Fup" for these states, and it does not accept any additional suffixes ("S", "W" or "WL") for the names of these states.

As we said before in the derivation algorithm for the weak while loop, a termination race could occur at some role, and the addition of a sequence number as a parameter was proposed in the received messages. However, the sequence number is not needed in all messages within the loop when a termination race exists, as described in (Section 4.3) and [77].

In this tool, we did not check for the termination race; the sequence number solution is applied for all coordination messages inside the loop body and in the follow-up part. The derivation rules of the design model for a weak while loop are described before (see, for instance, Section 5.2). For solving the termination race using the message parameter, our implementation algorithm uses Faleh's approach [31, 6] to ensure that all messages are received and processed in the correct order (see also [77]).

In the design model for the loop initiating role (or the choice role), the same rules

used for the derivation of choice and the strict sequence will be used here. In addition, an additional parameter (called sequence number [4, 31, 76]) is sent as a parameter with the coordination messages. It is used to indicate how often the body was repeated. A new state is created called "iniState" to initialize the sequence number to 0 before starting the loop. Two types of coordination messages are sent by the choice role; either flow message because of strict sequence, or a *cim* message if some role participates in the body and not in the Follow-up, or the other way around.

For the design model of the other roles following the choice and starting the execution with the reception of a message, it will be implemented using Faleh's approach (see [31]). A new state called "iniState" is created where two variables a and b, are initialized to -1. These roles will receive two coordination messages:

- A coordination message which triggers a transition to the loop body state: the received message has a sequence number, the role stores the received parameter in variable a, and executes the loop body state. The role then either goes again to the choice state and wait for the next message if $(a \neq b)$, or goes to the follow-up state if $(a = b)$.
- A coordination message which triggers a transition to the follow-up body state: the received message has a sequence number, the role stores the received parameter in variable b, and then goes to a new intermediate state called "INstate". The role then either executes the loop again if $(a \neq b)$, or goes to the follow-up state if they are equal.

An example of the derivation of a design model for the loop choice role in Figure 7.2(f) above is shown in 7.20(a). The role r_1 is responsible for initiating the sequence number and sends it in the coordination messages as a parameter. In FIN0s_1 and FIN1s_1, r_1 sends the required coordination messages to the body and Follow-up states (in this example, r_1 sends the flow messages to the initiating roles for these states), see for instance Listing 7.8. As we see in Listing 7.8, the flow messages fm_0 and fm_1 are sent to the role r_2 with the sequence number $r1Count$.

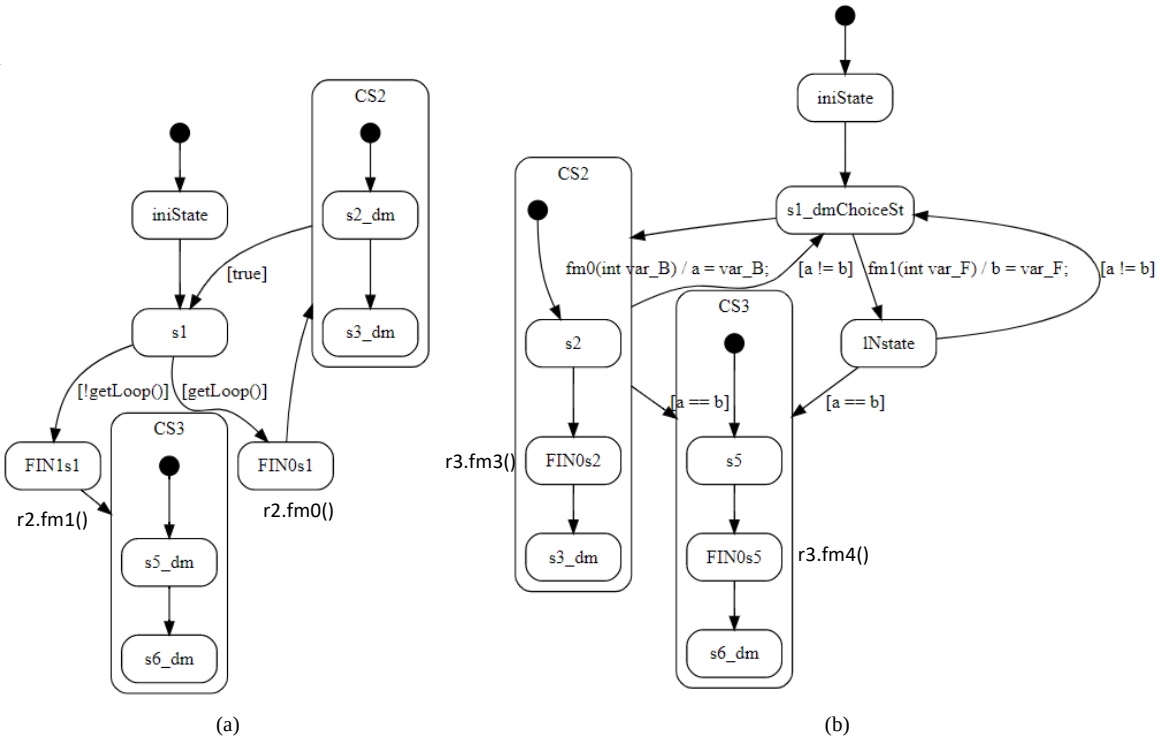


Figure 7.20: Design models for: (a) r1, (b) r2 , in Figure 7.2(f).

In the design model for r_2 (in Figure 7.20(b)), the two variables a and b are initialized to -1 before the loop starts (inside the state $iniState$). The role r_2 receives the coordination messages fm_0 (on the transition to the state CS_2 (or Body state)) and fm_1 (on the transition to the CS_3 (or Follow-up state)) with parameters and stores them in the variables a and b, respectively. The loop terminates if both variables a and b are equal, or otherwise, the role goes back to the initial state for another iteration. Using this implementation with a sequence number, even if the message fm_1 is received before some instance of message fm_0, it will not be consumed until all fm_0 messages are received and consumed.

Figure 7.21 shows the design model for r_3. The flow messages fm_3 and fm_4 are received with a sequence number. These messages are removed from the composite states CS_2 and CS_3, respectively, and placed on the outgoing transitions of the $s_1\_dmChoiceState$ to solve the problem of the dummy choice state. The same implementation (using the sequence number and message parameter) which was described before is applied for the design model of r_3.

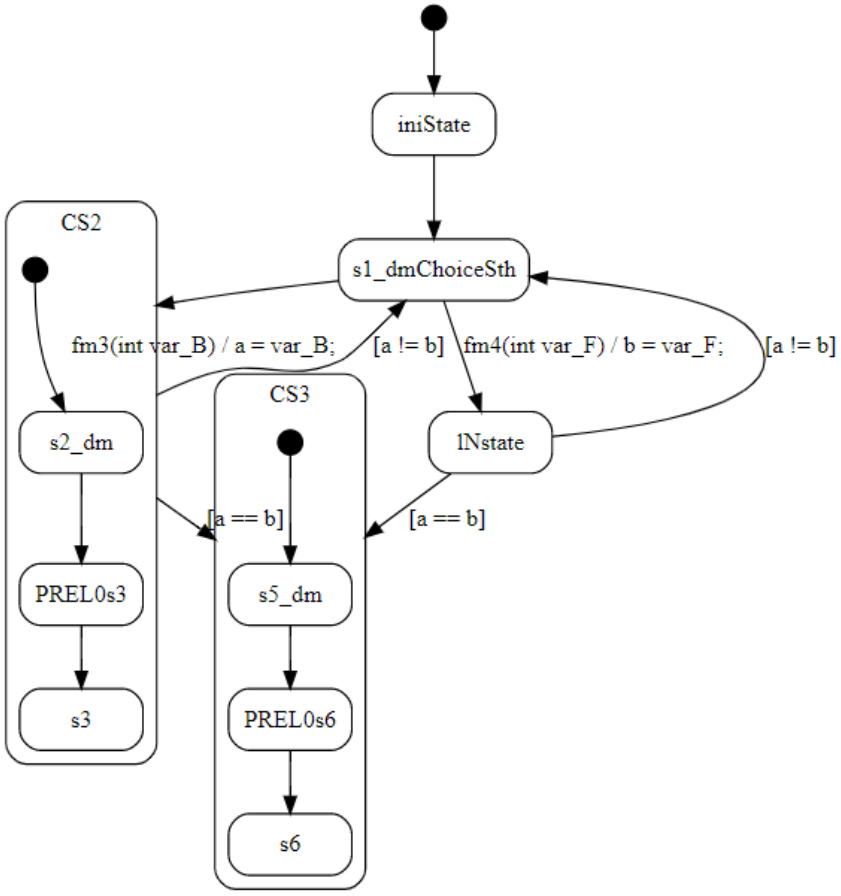


Figure 7.21: Design model for r_3 in Figure 7.2(f).

```
FIN0s1 { // between s1 and CS2
    do { r2.fm0(r1Count); } -> CS2;
  }
FIN1s1 { // between s1 and CS3
     do { r2.fm1(r1Count); } -> CS3;
}
```

Listing 7.8: The do actions for some states in the design model of r_1 in Figure 7.20(a)

A composite state could contain a weak while loop like the state CS_1_WL in sm_{17} (see Figure 7.22). Since CS_1 represents a while loop, the derivation of this state is discussed above. Because of the strict sequence between CS_1 and s_7, flow messages are required. The terminating roles for the weak while loop (or CS_1) $= (TR_{CS_2} \cup TR_{CS_3}) = \{r_2, r_3\}$.

Since CS_1 represents a weak loop, the derivation for this state is discussed above. The

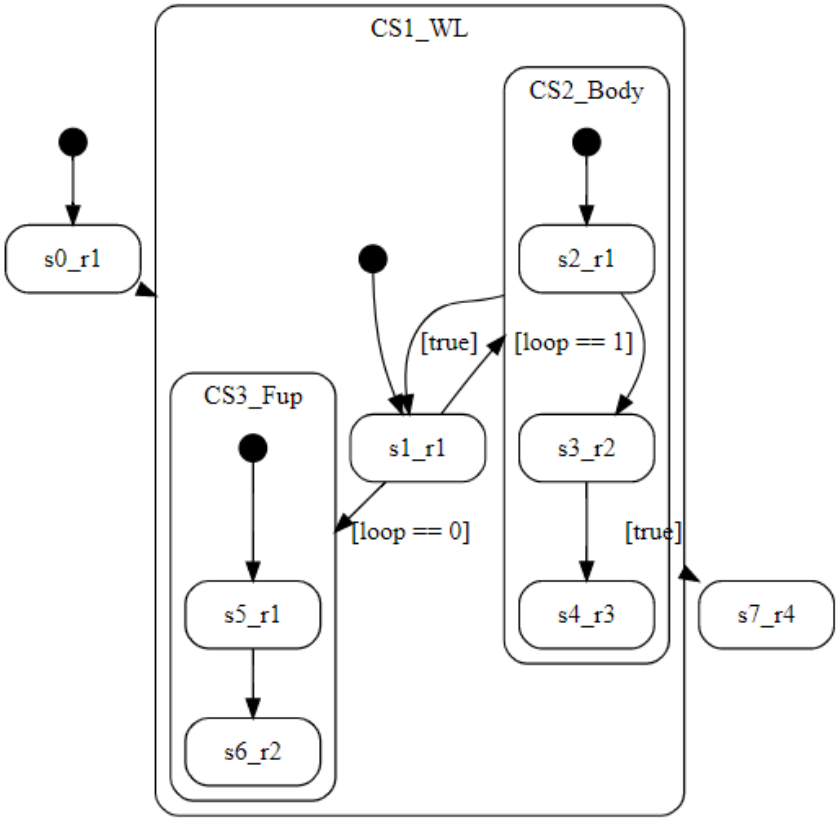


Figure 7.22: Global model sm_{17} contains a weak while loop.

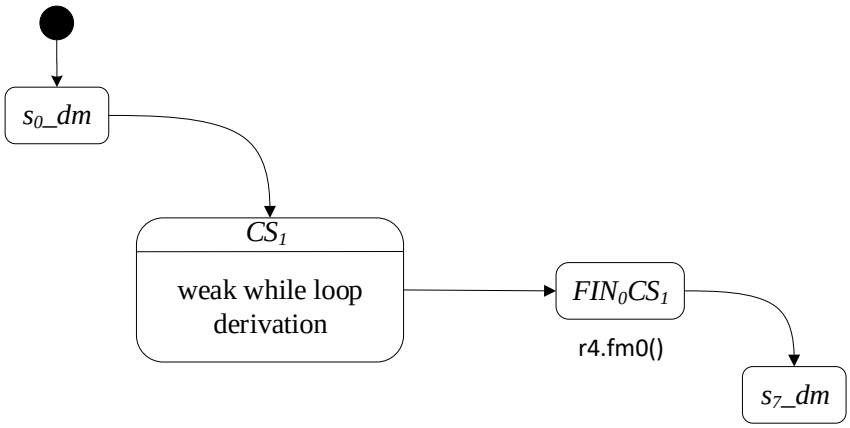


Figure 7.23: Design model for r_2 in sm_{17}.

design model for the role r_2 in sm_{17} is shown in Figure 7.23. Since r_2 is a terminating role for the weak loop in CS_1, it sends a flow message to r_2 ($do\ \{r_4.fm_0();\}$) inside the state Fin0CS_1. Since r_3 is also a terminating role, r_3 will send a flow message (in the design model for r_3) but with different message number.

7.6 Chapter Summary

This chapter presents the overall architecture of the derivation software tool. This tool is developed to implement the derivation algorithm of distributed design models which is described in Chapters 4 and 5.

Our tool does the transformation of a global Hierarchical State Machine Diagram (written in the extended HSM notation as described in Chapter 5) into local UML State Machine Diagrams (one for each role).

These local state machines can be easily implemented by any suitable tools that generate code from a design model written in UML HSM. This will be explained in Chapter 8.

Chapter 8

Evaluation and Testing

In this chapter, we are testing our tool (see Chapter 7) with an illustrative example of a Travel Management System. We will show the generated design models for the different roles which are involved in the system.

In addition, we test the correctness of the generated design models (generated by the tool) for the illustrative example. We have used Zakariapour's distributed implementation environment [85] which is supported by Umple for Java code generation for each design model. The generated code is executed using Zakariapour's environment in order to test the execution order of actions at each role in a distributed environment and see if this conforms to the correct order of actions in the global model.

8.1 Illustrative example: Travel Management System

A Travel Management System (inspired from [83]) is an online service used for hotel and flight reservations. The system is composed of the following roles: Client (C), Management System (M), Hotel System (H), Airline System (A).

The following are the steps of the typical behavior of the Travel Management System:

1. The Client (C) logs into the Management system (M), **M** validates the Client login information, and the Client is confirmed.

2. The Client enters the trip information, including the trip destination, travel dates, and preferences, and **M** confirms the data.

3. **M** sends a query to the Hotel and Airline systems to search for a suitable hotel and flight. Hotel and Airline systems will prepare the options concurrently. **M** will then confirm the available options.

4. **M** then adds its own profits to the searched prices.

5. **M** prepares the choices, and the Client views the choices.

6. The Client either selects one of the choices, performs the payment, and confirms the selection with **M**, and then the Hotel and Airline systems will confirm the reservation, or repeats the search to check other options (different hotel-flight types, dates or destinations).

8.2 Representation of the Travel System Example in our Proposed Global Modeling Notation

In this section, we represent the Travel Management System illustrative example in our proposed extended HSM for describing the global requirements model.

Figure 8.1 shows the graphical representation for the global model, which represents the travel management system written in the extended HSMs notation. The state machine has a strict sequence type, and the suffix "S" in the composite state names also refers to a strict sequence. The suffix in the names of simple states refers to the role associated with the state. For example, the simple state "validateUser" is associated with the Management "M" role. As we said before, the participating roles are C, M, H, and A. The Umple syntax for the global model is shown in Listing 8.1. We add print statements, which describe the executed event (and the role) in each simple state, in order to obtain a trace of the execution, which is useful to verify the generated design models.

1

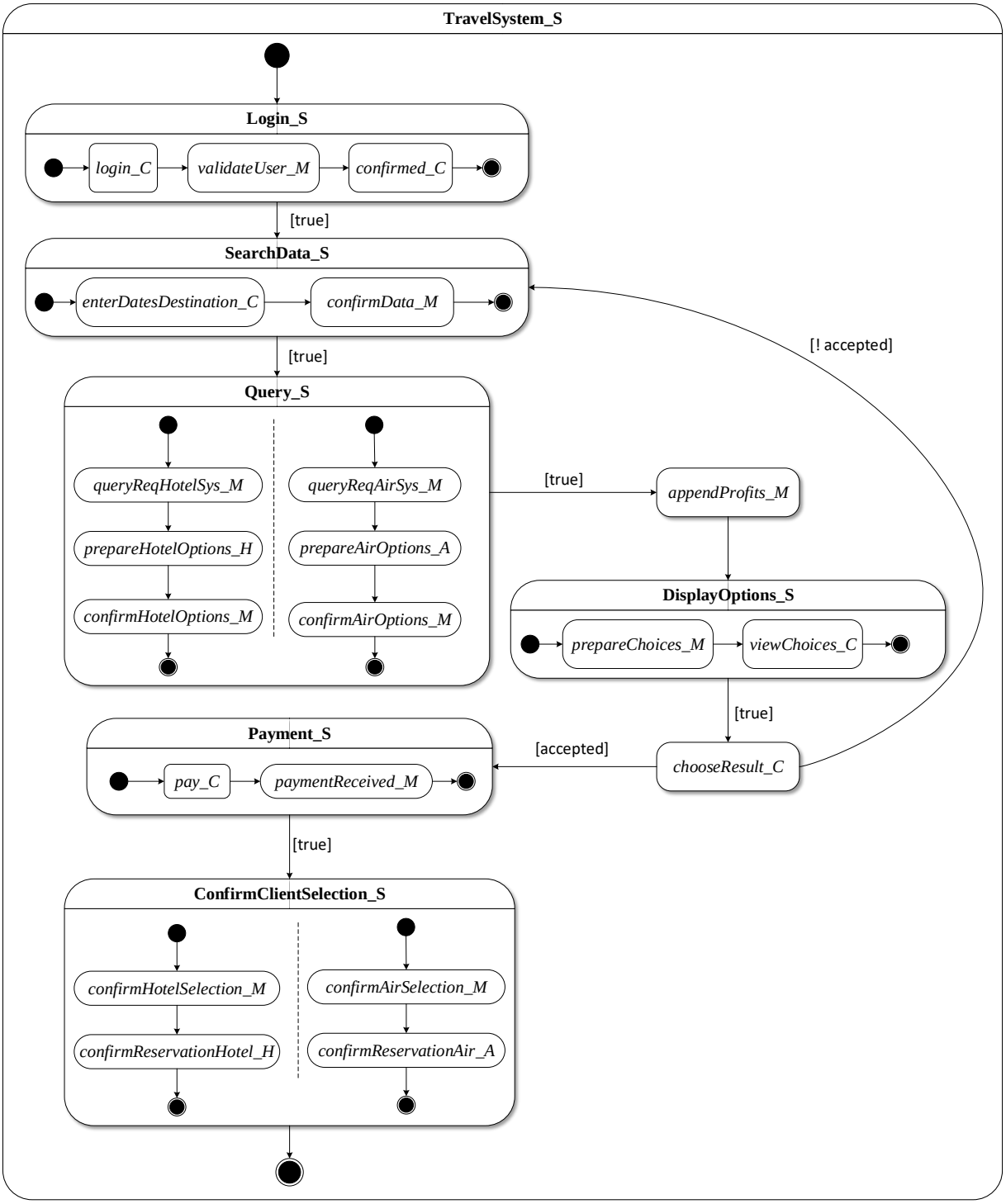


Figure 8.1: Graphical representation for the Travel Management System written in extended HSM notation (inspired from [83])

```
class TravelSystem
{
  TravelSystemSM_S {
   Login_S{
      [true] -> SearchData_S;
      login_C{
          entry / {System.out.println("Client: Login.");}
        -> validateUser_M;
      }
      validateUser_M{
         entry / {System.out.println("Management System: Validate User.")
    ;}
        -> confirmed_C ;
      }
      final confirmed_C {
       entry / {System.out.println("Client: Login confirmed.");}
      }
    }
    SearchData_S{
      [true] -> Query_S;
      enterDatesDestination_C{
        entry / {System.out.println("Client: Enter flight data, dates and
     destination");}
        -> confirmData_M;
      }
      final   confirmData_M {
        entry / {System.out.println("Management System: Confirm data.");}
      }
    }
    Query_S {
      [true] ->   appendProfits_M;
      queryReqHotelSys_M {
        entry / {System.out.println("Management System: Query Request
    Hotel System.");}
        -> prepareHotelOptions_H;
      }
```

```
35      prepareHotelOptions_H
36      {
37        entry / {System.out.println("Hotel System: Prepare Hotel options"
    );}
38        -> confirmHotelOptions_M;
39      }
40      final   confirmHotelOptions_M
41      {
42         entry / {System.out.println("Management System: Confirm Hotel
    options.  ");}
43      }
44     ||
45      queryReqAirSys_M{
46         entry / {System.out.println("Management System: Query Request
    Air System.");}
47         -> prepareAirOptions_A;
48      }
49      prepareAirOptions_A
50      {
51        entry / {System.out.println("Air System: Prepare Air options.");}
52        -> confirmAirOptions_M;
53      }
54      final   confirmAirOptions_M
55      {
56        entry / {System.out.println("Management System: Confirm Air
    options.");}
57      }
58     }
59     appendProfits_M
60     {
61      entry / {System.out.println("Management System: Append profits.");}
62        -> DisOptions_S;
63     }
64     DisOptions_S{
65      [true] -> chooseResult_C;
66      prepareChoices_M{
```

```
67          entry / {System.out.println("Management System: Prepare choices."
    );}
68          -> viewChoices_C;
69        }
70        final  viewChoices_C{
71          entry / {System.out.println("Client: View choices.");}
72        }
73       }
74       chooseResult_C
75       {
76        [!acceptable] -> SearchData_S;
77        [acceptable] -> Payment_S;
78       }
79       Payment_S
80       {
81        [true] -> confirmClientSelection_S;
82        pay_C
83        {
84          entry / {System.out.println("Client: Pay.");}
85          -> receivePayment_M;
86        }
87        receivePayment_M
88        {
89          entry / {System.out.println("Management System: Receive payment."
    );}
90
91        }
92       }
93       confirmClientSelection_S {
94        confirmHotelSelection_M {
95          entry / {System.out.println("Management System: Confirm Hotel
    selection.");}
96           -> confirmReservationHotel_H;
97        }
98
99        final confirmReservationHotel_H
```

```
100       {
101         entry / {System.out.println("Hotel System: Confirm Reservation")
      ;}
102
103       }
104      ||
105       confirmAirSelection_M {
106          entry / {System.out.println("Management System: Confirm Air
      selection.");}
107          -> confirmReservationAir_A;
108       }
109
110       final confirmReservationAir_A
111       {
112         entry / {System.out.println("Air System: Confirm Reservation");}
113       }
114     }
115   }
116 }
```

Listing 8.1: Umple representation for the Travel Management System written in extended HSM notation.

8.3 Usefulness of Representing the Global Model Using Extended HSM

8.3.1 Advantages of State Machines for Modeling Global Requirements and Communication Protocols

Using the UML state machines for the design model is common for systems that interact with their environment and where the order of interactions is important. In addition, many tools support the implementation of a state machine model, Umple [12] is an example.

The HSM notation simplifies the modeling of complex systems. In addition, it formally defines the syntax and semantics of concurrency and nesting concepts that are not supported by simple state machines. As can be seen by Figures 8.5 through 8.9, the distributable design model of the Travel System example can be easily modeled using the HSM notation.

In our work, we also use an extended HSM for defining the global model, as shown in the example of Figure 8.1. Using one kind of modeling paradigm for both, the design and the global models, is advantageous. It makes the derivation algorithm easier, and simplifies the understanding by the developer. Lamarti [5] used Activity Diagram for modeling the global requirements and the design models. However, the availability of code generation tools from a design model written in Activity Diagrams is limited.

8.3.2 Advantages of Using Local Actions and No Explicit Message Passing

In our global modeling notation, we do not use explicit message passing between roles like in MSC (see Section 2.3). Instead, each local action for a given role is represented as an action in the state (written in the *do-action* of the state). Adding messages in the global model could lead to many difficulties such as race conditions, the problem of implied scenarios, and others (see [2] for more details). We should note that the race condition could still happen in the design model between the flow messages which are introduced by the derivation algorithm. However, because of the reception pool at each role, this does not present a problem.

8.3.3 Comparison with Another Notation (Activity Diagram)

HSM allows a hierarchical representation of the global requirements model; this makes the model more readable compared to other notations. Figure 8.2 shows the representation of the Travel Management System example using Activity diagrams [16]. A hierarchical approach is used here, where the suffix "*" indicates that an activity is defined by another

activity diagram. These "internal" Activity diagrams for the activities "Login", "Seach-Data", and "Query" are shown in Figures 8.3 and 8.4. We can see that using Activity diagrams for modeling the Travel System example needs separate diagrams. Therefore, this is an example that shows that the integrated hierarchical representation of HSM enhances the readability of the global model. Hierarchical representation using separate diagrams (such as with Activity diagrams) is a good solution for modeling a very complex system where the global model is too large to fit on a single page. However, we note that Umple and our tool do not support separate diagrams for a model.

It is worth mentioning that Activity Diagrams allow to clearly represent the responsibility for each activity by using swimlanes. However, HSM needs suffixes in the state, as we have shown in this work.

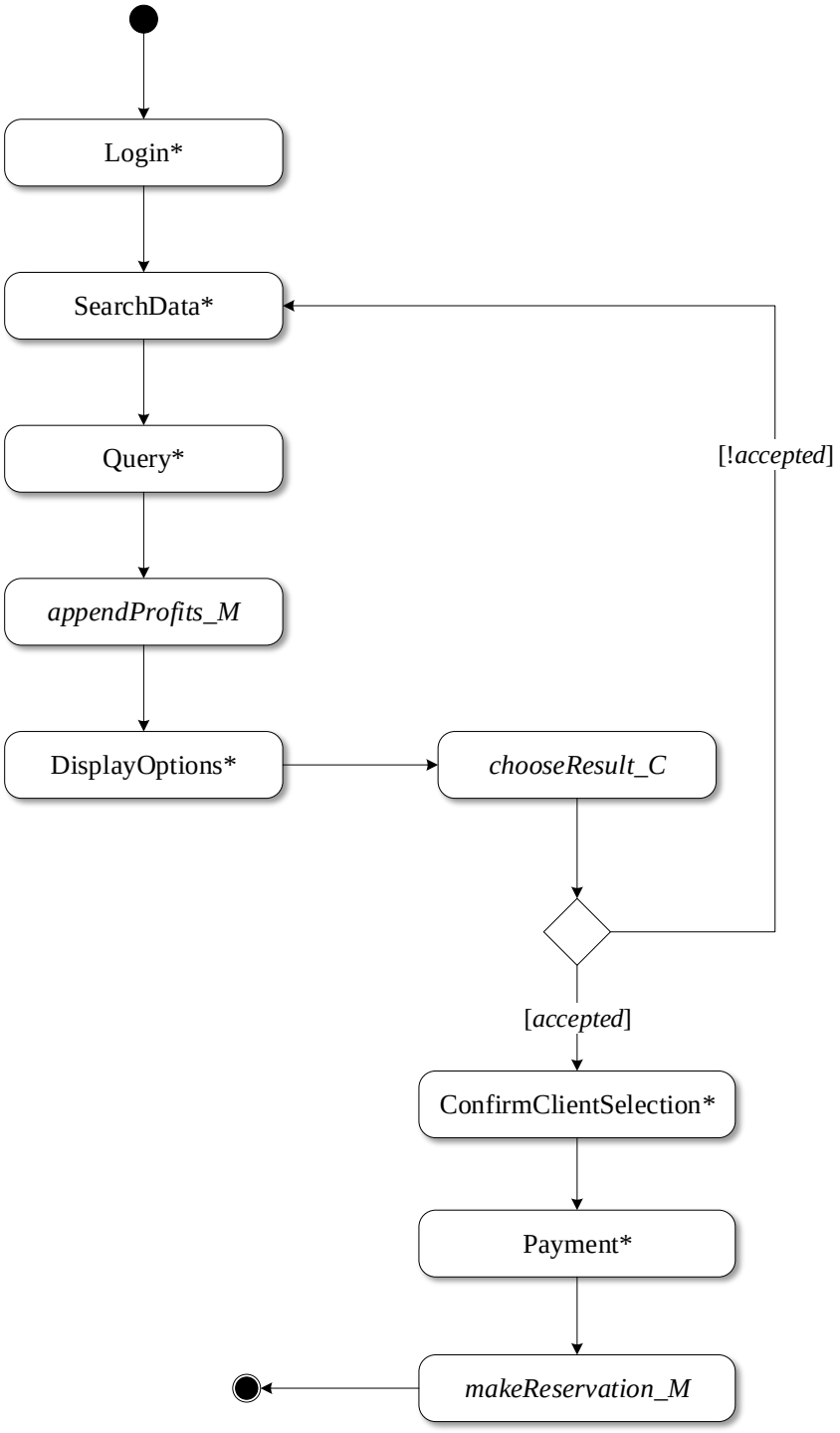
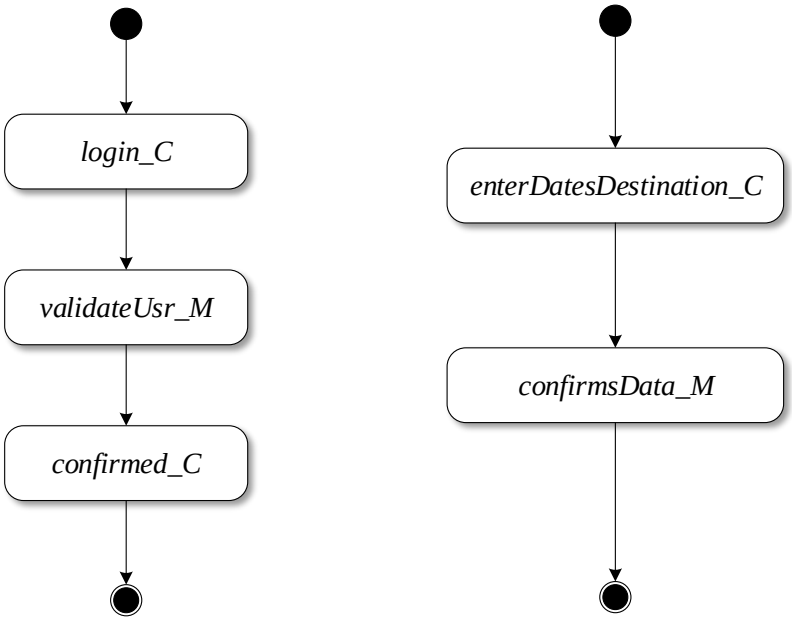


Figure 8.2: The representation of the Travel System example using the Activity Diagram notation.



(a) Login activity

(b) SearchData activity

Figure 8.3: Internal Activity Diagrams for the following activities in Figure 8.2: (a) VisitWebsite*, (b) Seach*.

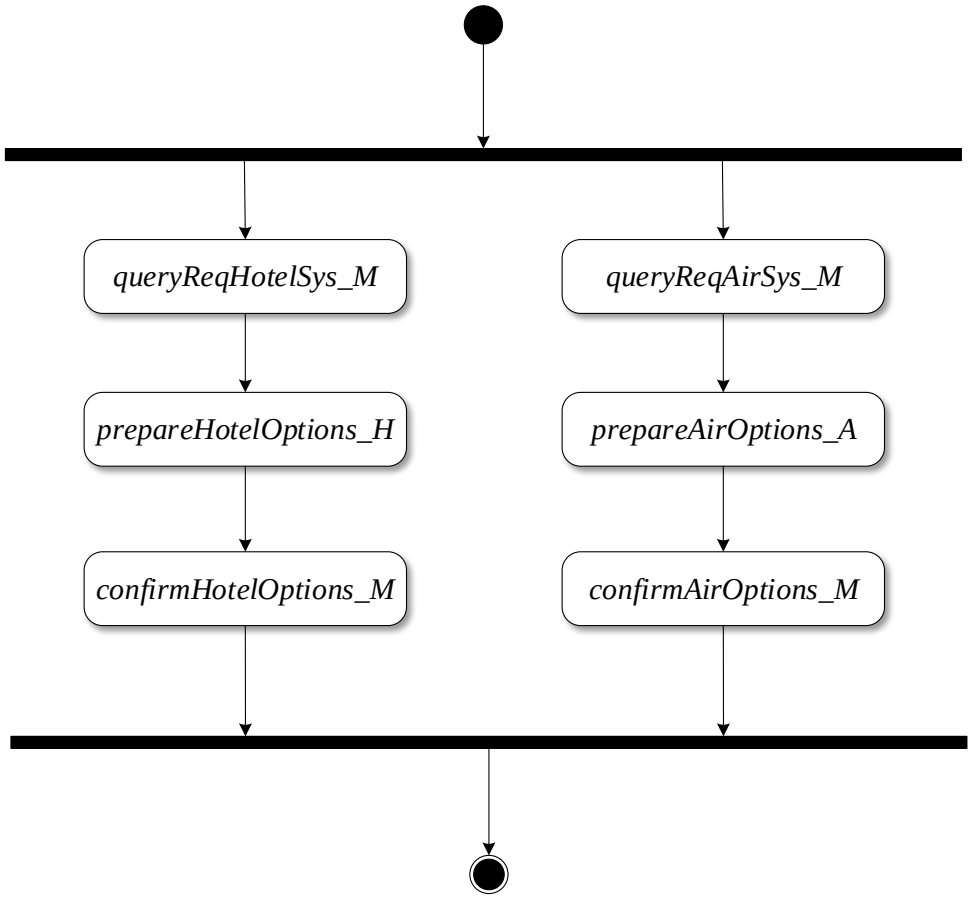


Figure 8.4: Internal Activity Diagram for the Query* activity in Figure 8.2.

8.4 Design Model of the Travel Management System

In this section, we show the graphical representation of the design models that are generated by our tool for the participating roles in the Travel System example.

Figures 8.5 through 8.9 show the graphical representations for the generated design models for the roles: Hotel (H), Airline(A), Client (C), and Management System (M). The corresponding Umple codes for all roles are shown in Section A.1 (Appendix A). Note that the Umple codes for the design models of all roles are generated by our tool, and the graphical representations are generated using Umple Online [20].

Figure 8.5 shows a complete design model for the role Hotel (as generated by the tool), Figure 8.6 shows an improved representation for this design model. The Figures 8.7, 8.8, and 8.9 also show such an improved representation for Airline(A), Client (C) and Management System (M) roles, respectively.

In the improved representations, we did the following modifications to enhance the readability:

- Elimination of dummy states: We replaced a composite state in which the role does not participate, by a simple dummy state (see the states: "Login", "SearchData", and "Payment"). Also, we removed the state machine inside the concurrent composite state when all states are dummy. For example in the "Query" state in Figure 8.6, we removed the dummy states related to the Airline System, and we did the same in the composite state "confirmClientSelection".

- Improve the layout provided by Umple Online: Some messages or guards (*e.g.* [true]) are not placed on the correct transition by Umple Online. For example, the message $cimC_5$ inside the composite state "Payment" in Figure 8.5 should be on the transition between the states "chooseResult" and "Payment".

- Showing the sending of messages: The sending of coordination messages is performed in the *do action* of the FIN states, and this is not shown in the graphical representation generated by Umple Online (see Figure 8.5). In the improved version in Figure

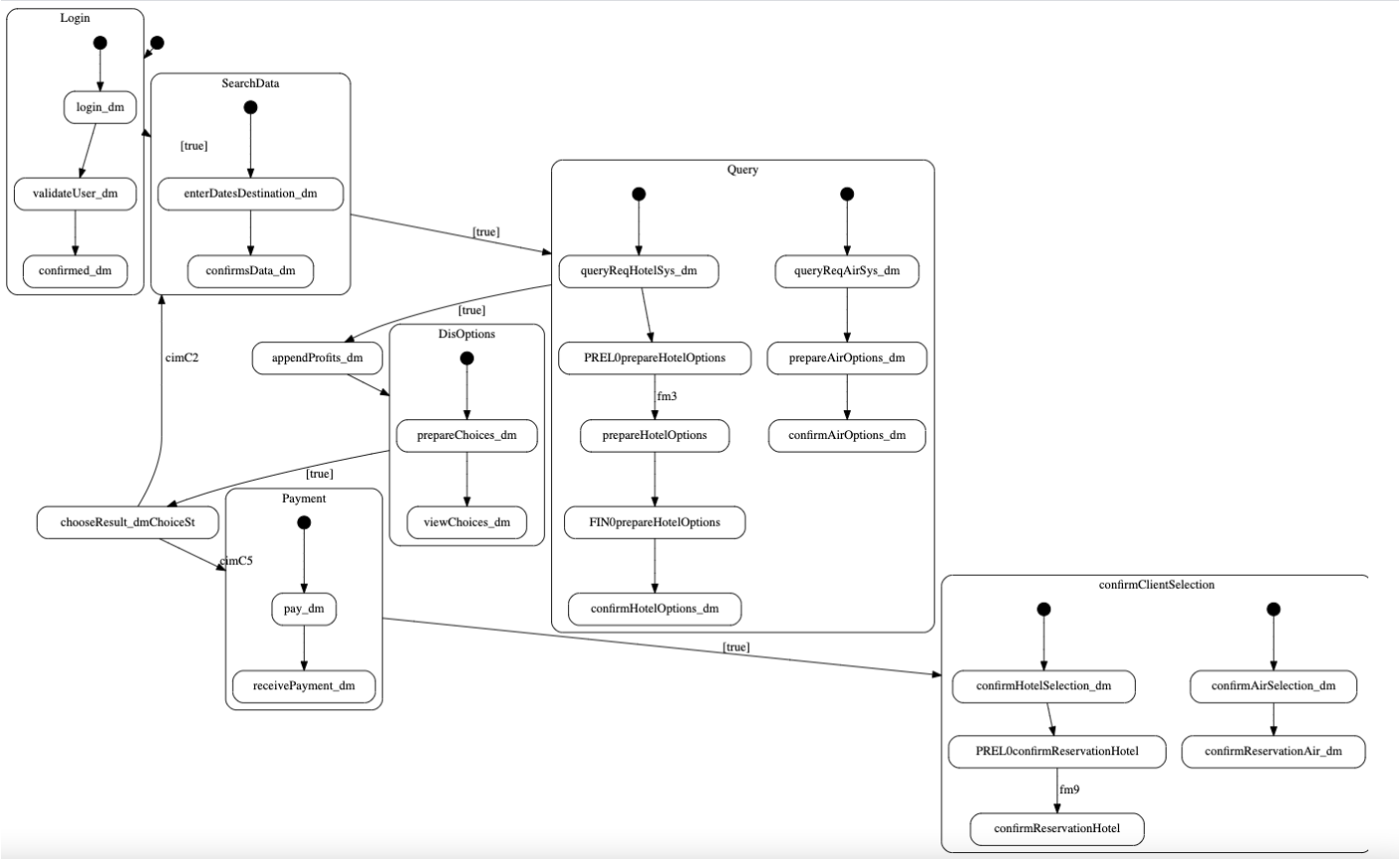


Figure 8.5: The generated design model for the Hotel (H) role.

8.6, we added the sending of messages on the transition after the FIN states. For example, the role Hotel sends the message fm_4 to the Management System Role (written $m.fm_4()$) inside the composite state "Query".

Figure 8.7 shows the improved design model for the Air system (C) role. Figure 8.8 represents an improved design model for the Client (C) role. For example, in the $final$ state "FIN0chooseResult", the Client sends the coordination messages $cimC_0$, $cimC_1$, and $cimC_2$, to the roles Management, Air, and Hotel systems, respectively (where m, a, and h in the Figure refer to the roles Management, Air and Hotel systems, respectively). Figure 8.9 is the improved design model for the Management system (M) role.

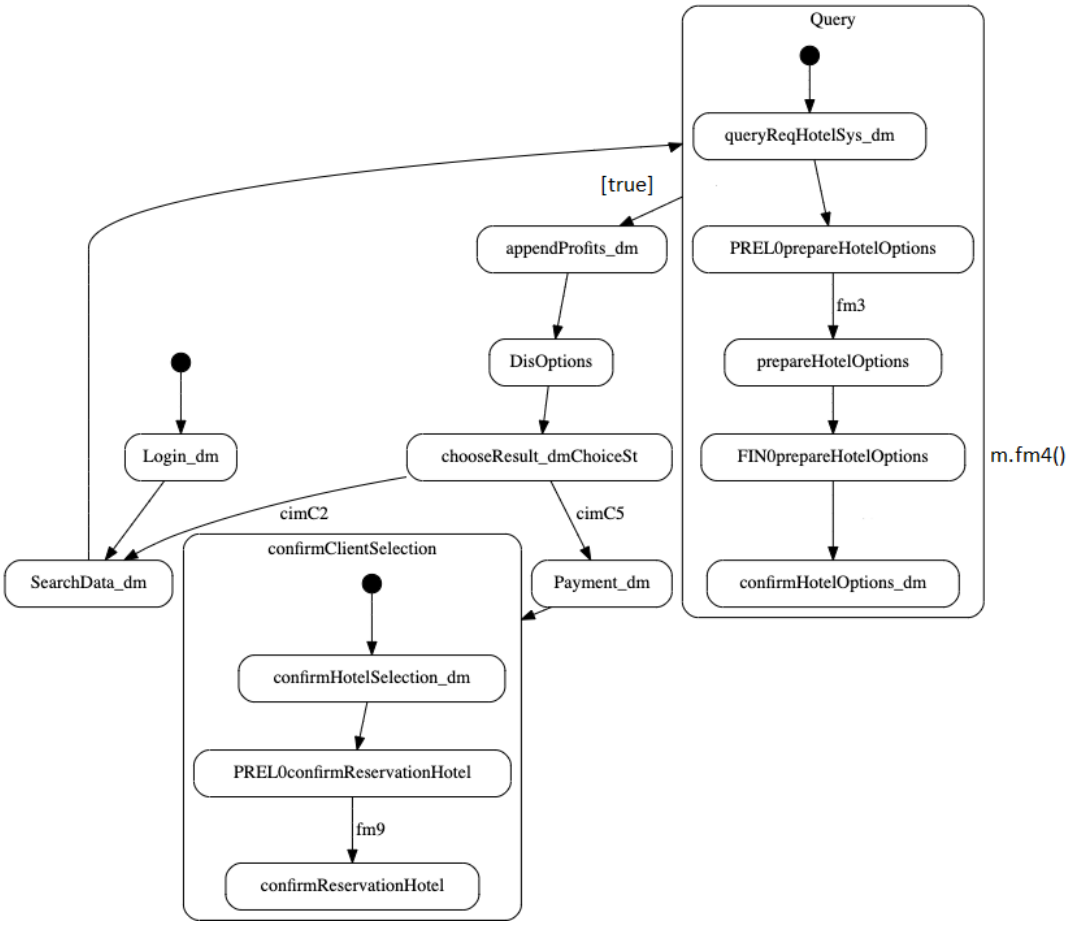


Figure 8.6: The generated design model for the Hotel (H) role (improved version).

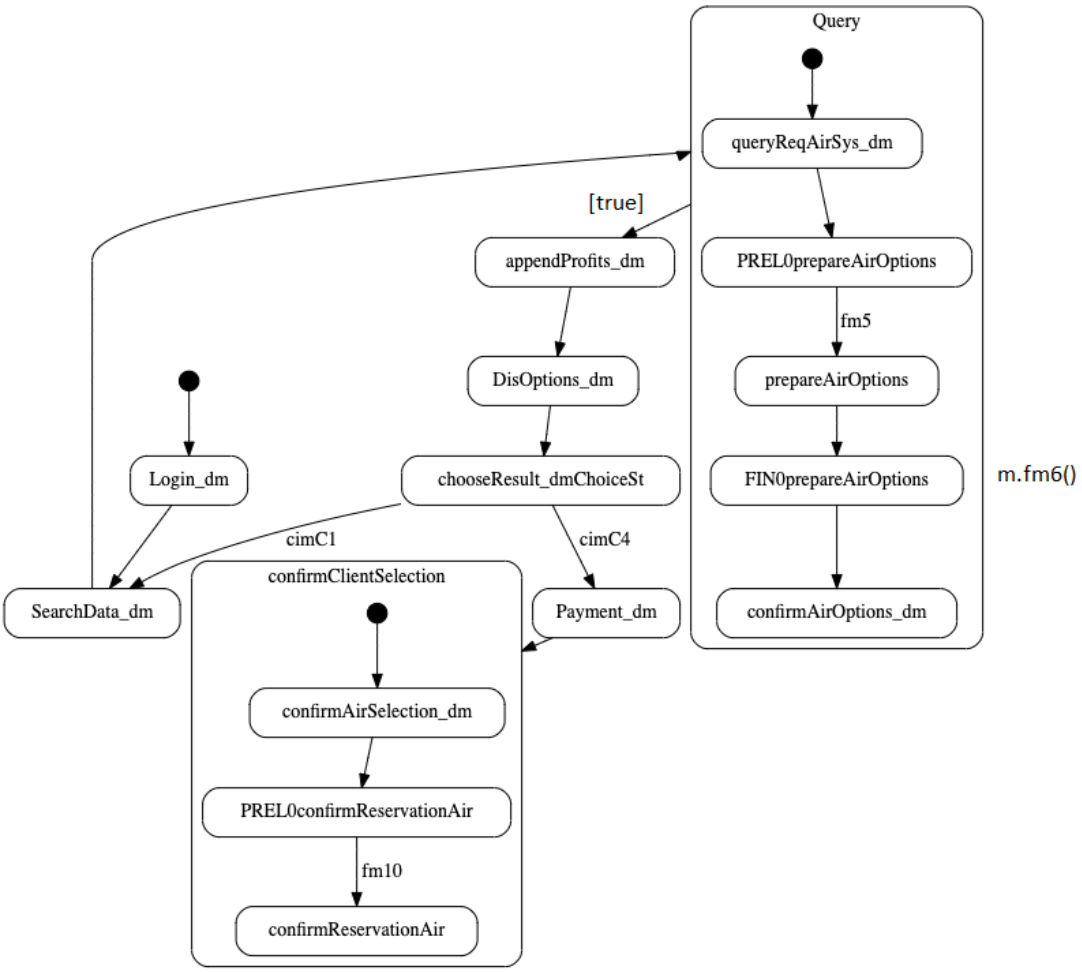


Figure 8.7: The generated design model for Airline(A) role (improved version).

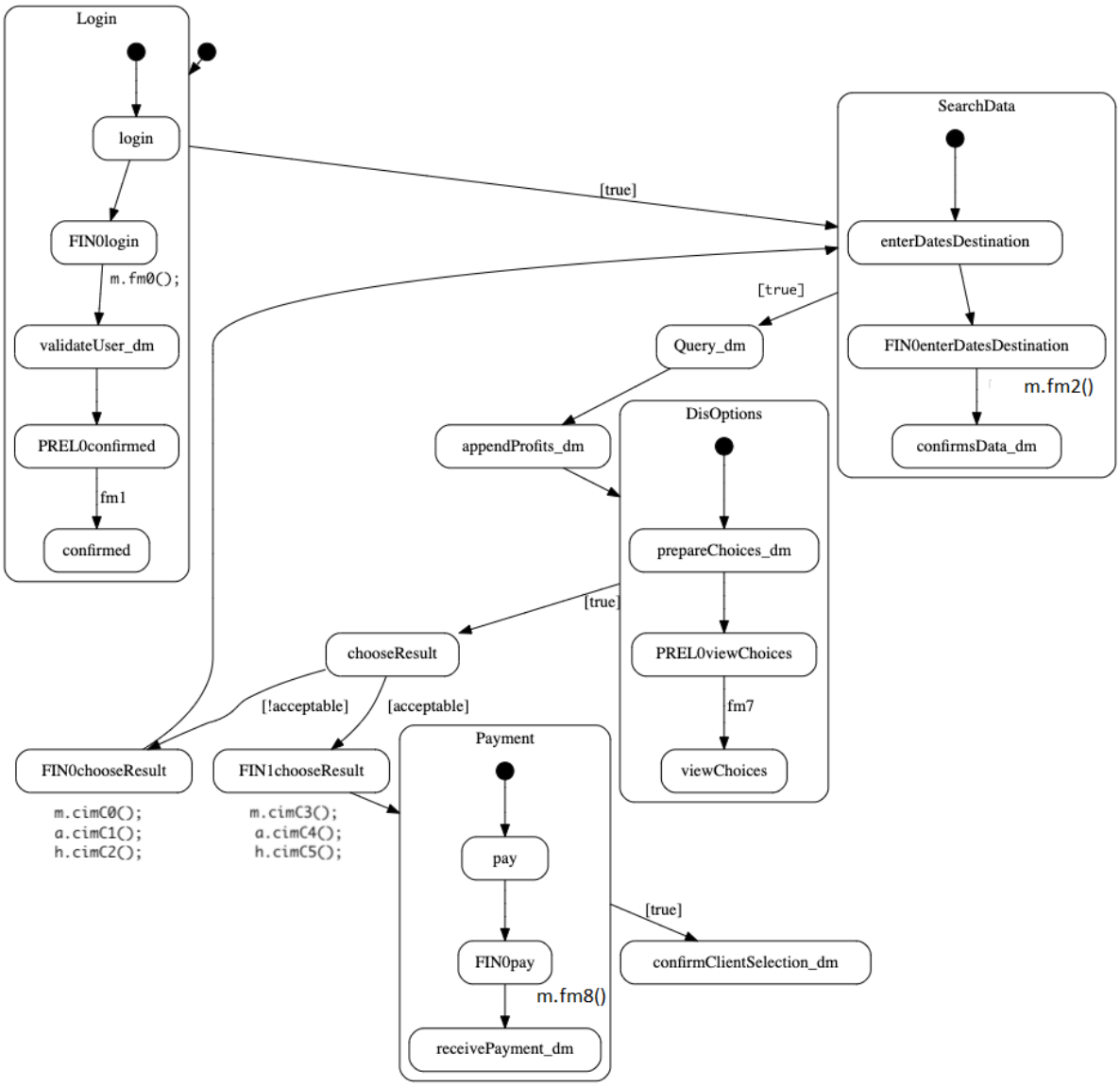


Figure 8.8: The generated design model for the Client (C) role (improved version).

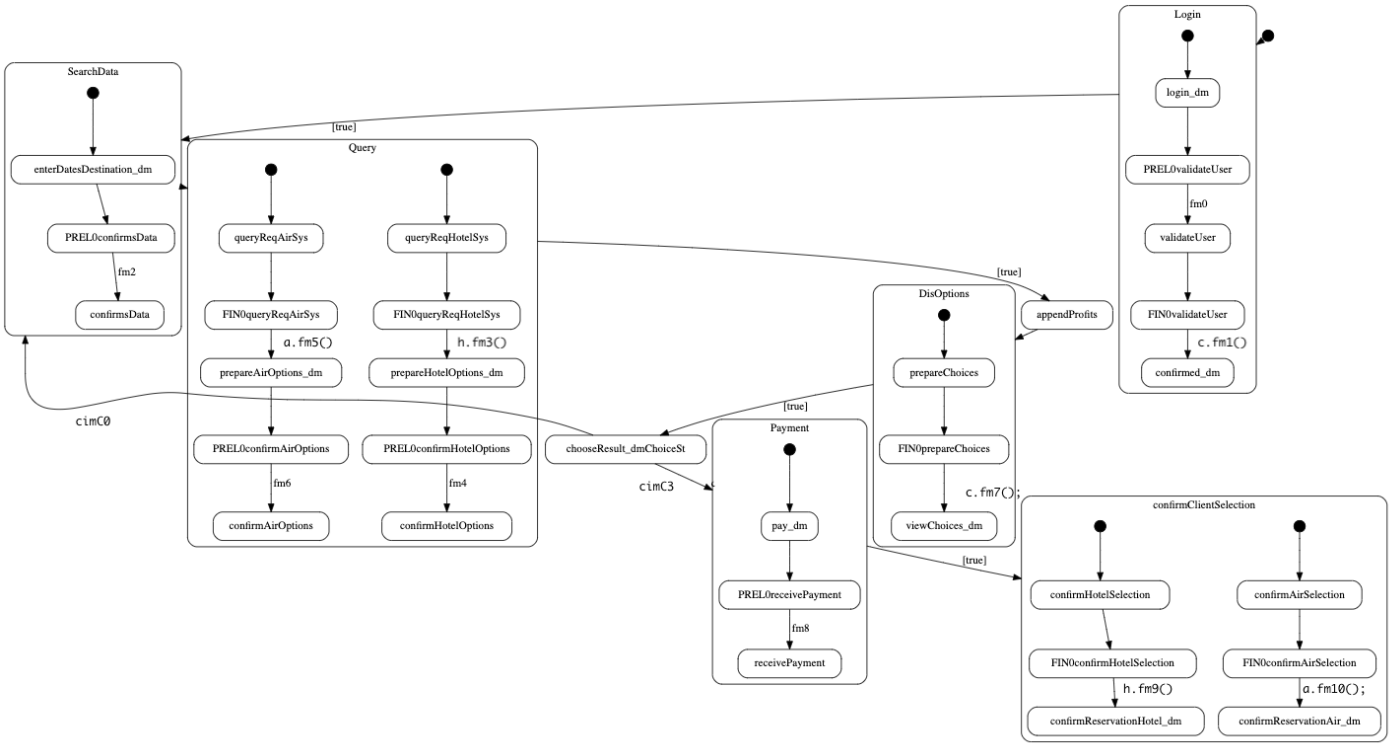


Figure 8.9: The generated design model for the Management System (M) role (improved version).

8.5 Improvements for the Tool and the Global Model

The following improvements could be considered as future work:

8.5.1 Dynamic Sessions Establishment

In the current system, all roles are statically defined. One of the improvements for our tool is to establish dynamic sessions, which means the roles which are active in the application at a given time are dynamic [69, 86, 87]. At each point of execution of the activities in the application, there is a subset of the roles involved. For example, in the Travel System, the Management first establishes a session with the user (Client). Then, several sessions are established as required between the Management system and the reservation agents (Hotel and Airline systems).

Suppose that we have two types of hotels: Budget and Value hotel systems and based on the user selection (Budget or Value), the Management will establish a session with the required system. For example, if the Client preference is a budget hotel, then a session will be created with the Budget hotel only. If the Client accepts one of the choices after the Query state, the Management system will confirm the selection with the Budget hotel. If the result was not accepted, the Budget hotel will be notified, and the session will terminate. In both cases, the Value hotel system will not be involved. In our global model here, the roles are fixed. All roles are active at the same time, and this is a limitation.

8.5.2 Elimination of Dummy States

As we said in the derivation tool chapter (see Chapter 7), the simple state becomes dummy if the role, for which the design model is derived, does not participate in that state. However, our tool keeps these states.

One of the improvements for our tool is the elimination of simple dummy states. Also, the composite state, which contains dummy states only, should be replaced by a simple dummy state.

8.5.3 Data Transfer between the Roles

One of the features which was not described in our global model is the data storage at the different roles and the data flow between them. Data can be described in the global model by defining local variables at each role where the information related to that role is stored [49]. For example, the Hotel system role stores a list of hotels and the state for each one (available or occupied).

Data can be exchanged between different roles as a parameter in the existing flow messages. The role that receives the flow message checks the message parameters (if any) and updates its local variables. For example, the Client sends the trip search data, including the dates and destinations, as parameters in the flow message to the Manager. Then, the Manager receives these data and updates his local variables.

8.6 Travel Management System: Tests and Results

In this section, we test the correctness of the generated design models by the tool for the Travel System example. First, we get the design models for the participating roles which are generated by our tool (see Section 8.4). Then we have used Zakariapour's distributed implementation environment [85] which is supported by Umple for Java code generation for each design model. The generated code will be executed using Zakariapour's environment, we test the execution order of actions at each role in a distributed environment and see if this conforms to the correct order of actions in the global model.

8.6.1 Principles of Zakariapour's System

As we said before, Umple is a model-driven development environment that can be used for code generation. In [85], Zakariapour extended Umple to support distributed application development using Java RMI (Remote Method Invocation) and web services.

The "basic Umple" uses procedure calls to communicate between the state machines of different roles. In Zakariapour's system, each state machine has a queue, and a Remote

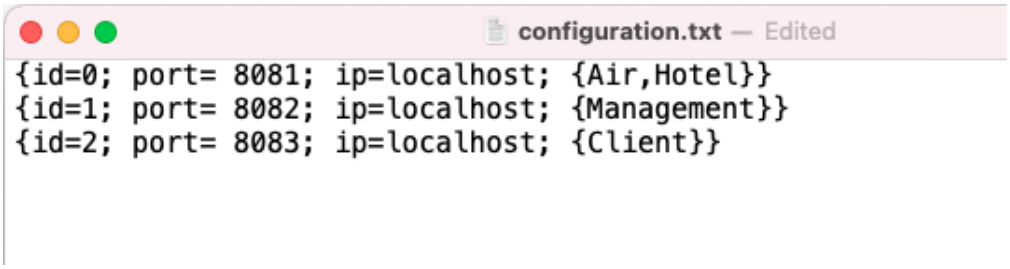


Figure 8.10: Configuration file with three components and four roles

Procedure Call (RPC) is used to put a message into the queue. This RPC is implemented using Java RMI. The sending component makes a RPC to the destination component in order to enter the message into the queue of the destination component, and then the sender continues. The "basic Umple" does not have any distributed implementation, and the state machines, by default, do not have a queue. Zakariapour's system supports two types of queues: FIFO and Pool queue, and we used here the Pool queue (see Chapter 4 for more details).

In Zakariapour's system, a configuration file is needed to indicate the allocated state machines (design models for the roles) to each component (called a "node" in Zakariapour's thesis [85]). Each component represents a Java run-time environment, and we can have several run time environments (each with its own console) on the same machine. The configuration file has information about the number of components (nodes), and for each component, it has the following attributes: component ID, IP address, port number, and the state machines allocated to that component. Figure 8.10 is an example of the configuration file; it has three distributed components and four roles (the roles "Air", and "Hotel" are run on the same component). All components in Figure 8.10 are running on the same machine (ip=localhost). We should note that if the components are running on different machines, a separate copy of the configuration file is required at each machine. Also, we should mention that using Zakariapour's environment, we are limited to four distributed components.

In order to generate a distributable implementation for the roles, Zakariapour defines the classes (the state machines which correspond to the design model for each role) as *distributable* by adding the keyword "distributable" to each class. The example below

(see Listing 8.2) shows the *distributable* Client class. Then, we should combine these distributable classes for all roles in one Umple file, and use Umple Online [20] for the Java code generation.

Zakariapour implements one type of communication protocol between all pairs of roles. At this level, the system could this work can be easily extended such that different communication protocols could be used for different pairs of roles.

```
1 class Client{
2 {
3     distributable;
4     ...
5 }
```

Listing 8.2: An example of the syntax for a distributable class

8.6.2 How to Use Zakariapour's Environment for the Travel System Implementation and Testing

In this section, we explain how we used Zakariapour's environment for a distributed implementation of the Travel Management System and how to run the generated code in a distributed environment.

The Umple file, which contains the design models for the participating roles in this system, is shown in Appendix A (see Section A.2). This file will be used for Java code generation using Umple Online [20]. In this Umple file, we did the following changes compared to the design models generated by our tool (see Section A.1):

1. The code generated by Umple Online for the $do-action$ (used to write a Java code inside the state) does not work properly, therefore, in Listing A.2 we write the Java code in the $entry-action$ of the state instead of $do-action$.

2. Umple does not support a *completion transition* (transition without any triggering event) from a composite state to another state. Therefore, we use a transition with a

guard [*true*], which works like a completion transition (see Section 2.7.4.3). The Java code generated by Umple for such a completion transition does not work properly. To solve this problem, we add a completion transition in the last simple state inside the composite state to the next state. In the Client class, for example (see Listing A.2), the completion transition from the composite state "Login" to the next state "SearchData", we add this transition in the simple state *confirmed*.

We then use the configuration file in Figure 8.10 to run the generated Java code. We run the components of Figure 8.10 in parallel (in separate Java run-time environments). Listing 8.3 shows the commands which are used to run the generated distributable code at each component. For example, the first component (including the roles Airline and Hotel) has the Id=0 in the configuration file (see Figure 8.10). We use the following command to run this component: *java Test* 0 >> *TestResult.txt*, where "Test" refers to the class which contains the *main*() function, "0" refers to the component Id in the configuration file, and the execution output will be written to the text file "TestResult.txt". For the other components, we use the class "UmpleRuntime" + component Id, as indicated in Listing 8.3. In order to see the execution order of actions in a single file, all Java run-time environments write onto the same file.

```
1  java Roles.Test 0 >> TestResult.txt
2  java Roles.UmpleRuntime 1 >> TestResult.txt
3  java Roles.UmpleRuntime 2 >> TestResult.txt
```

Listing 8.3: Running commands

8.6.3 Checking the Execution Order of Local Actions

Global testing at several points of observation in a distributed environment is not an easy task [88, 89, 90]; it is hard to know in which order the events are happening at different points. Verifying the relative order between two events that occur at two different distributed points is difficult; the reason for this is that we do not know what kind of delay is involved before the events are signaled to a central testing unit. One solution is to have one

global clock at all distributed points, and each component makes a log of its executed events with timestamps. At the test unit, we then merge these logs according to the order of the timestamps. The precision of this solution is limited to the precision of the different clocks at the different points of observation. Data exchange between different distributed points is also important to identify the execution order of an event in a distributed environment. If event X receives an input parameter from event Y, and both actions are located at different distributed components, this will help determine the event's execution order. These references also discussed the use of inputs and outputs for testing (see also [3]).

In this Travel System example, we consider only local events which happened at different places, and these events are neither inputs nor outputs. Since we do tracing for the events that are executed by different roles at different places, we can consider these events as output. In our system, we have only two inputs which happen when the user selects one of the search options or accepts the travel reservation.

In this section, we do centralized testing where different distributed components write to the same file. We used the file method *append* (>>) in order to allow different Java run-time environment consoles to write to the same file. However, using the *append* method, only one process can write to the file at a given time and, therefore, concurrency is not possible.

Figures 8.11 shows the execution order of actions for the roles: Client, Management, Hotel, and Air systems. At the beginning of the execution, the components (nodes) try to connect to each other. Each line in the figure starts with the role name (the role which executes the action) followed by a description of the action.

Now we check the actions which are executed at the Client component in Figure 8.11 and compare their order with the global model in Figure 8.1. The first simple states in the global model (inside the composed state "Login_S") which correspond to the Client (C) are: "login_C" and "confirmed_C". Then, the Client executes "enterDatesDestination_C" state inside "SearchData_S". The first three actions executed by the Client in Figure 8.11 are: *Login*, *Login confirmed*, and *Enter flight data, dates and destination*, and this conforms to the global model. The Client then executes the following states in the global model:

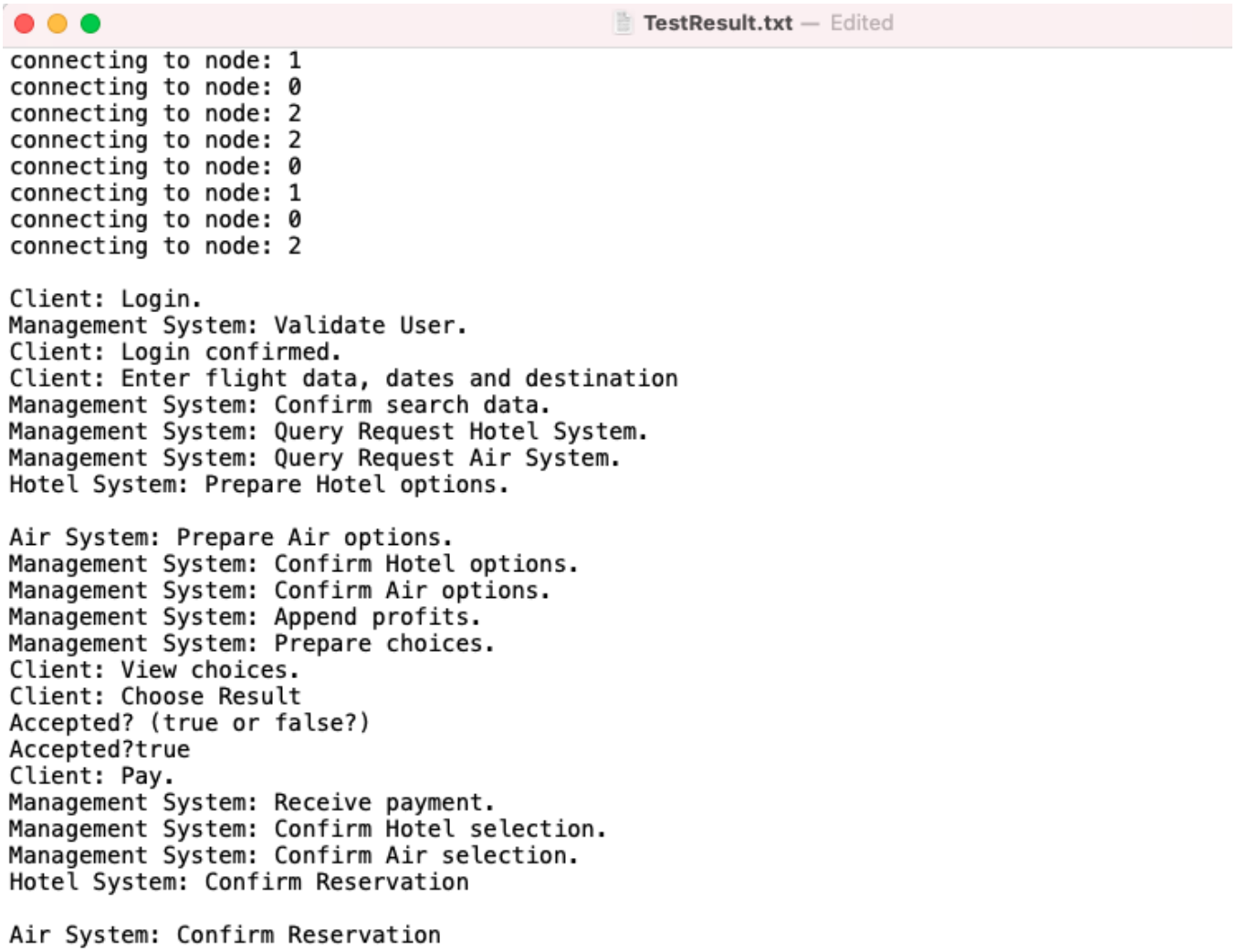


Figure 8.11: Example of execution order of actions for the roles: Client, Management, Hotel and Air systems.

"viewChoices_C", and then the choice state "chooseResult_C". If the Client accepts the result, he will continue for the payment "pay_C", and if the result was not accepted, the Client repeats the search again. In Figure 8.11, we see that the Client executes the actions that correspond to these states in the correct order.

At the Management System component "M" (see the execution order of actions in Figure 8.11), the following actions are executed: *Validate user*, *Confirm search data*, and these actions correspond to the states "validateUsr_M" and "confirmData_M" in the Travel System (see the global model in Figure 8.1). The Management System then performs the concurrent actions *Query Request Hotel System* and *Query Request Air System* and then the actions *Confirm Hotel options*, and *Confirm Air options* (correspond to the concurrent composite state "Query" in the global model). This is part of the Management System actions order.

Hotel system executes the action *Prepare Hotel options* (corresponds to the state "prepareHotelOptions_H" inside the composite state "Query" in the global model) and then the action *Confirm Reservation* (corresponds to the state "confirmReservationHotel_H" inside the composite state "ConfirmClientSelection_S" in the global model). Airline system executes similar actions in the same composite states.

We can see that the action execution order in Figure 8.11 conforms to the global model in 8.1. However, as we said before, this is not a complete test due to the difficulties which are associated with the distributed systems testing. In addition, since we have concurrency, there could be several execution sequences based on different transmission delays or different execution speeds at the different components.

If we want confidence that the generated code always conforms to the global model, the following two points should be assured:

- The correctness of the principles for the derivation of the distributed design models as described in Chapters 4 and 5.
- The fact that the Tool described in Chapter 7 implements these principles correctly.

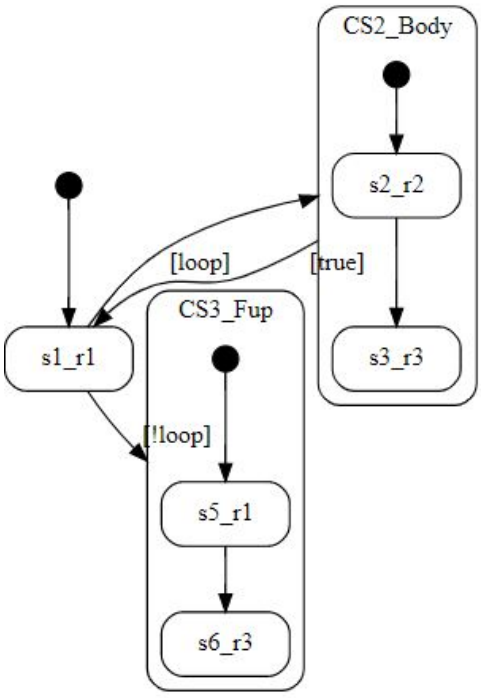


Figure 8.12: Weak while loop example (global model)

8.7 Weak While Loop: Tests and Results

In this section, we show the testing results for the weak while loop, since the Travel System illustrative example above does not include this kind of loop, and the weak loop implementation is complex, involving sequence numbers to solve the *termination race* (discussed in the Sections 4.3 and 7.5.7). Figure 8.12 shows an example of a weak loop (global model).

Figures 8.13 through 8.15 show the improved representation for the generated design models by the tool for the roles r_1, r_2, and r_3. The Umple files of the generated design models for the roles r_1, r_2, and r_3 are shown in Listings B.1.1, B.1.2, and B.1.3, respectively (see Section B.1 in Appendix B).

In the design model for r_1 in Figure 8.13, the guard [loop] (in the global model) is replaced by *getLoop()* (this *getLoop()* is created when the internal representation of the role design model is translated back to Umple syntax). The sequence number $r1Count$ is initialized to 0 in the state *iniState*, and incremented in the states FIN0s1 and FIN1s1. This sequencer number is sent as a parameter in the messages fm_0 and cim_0. In the design

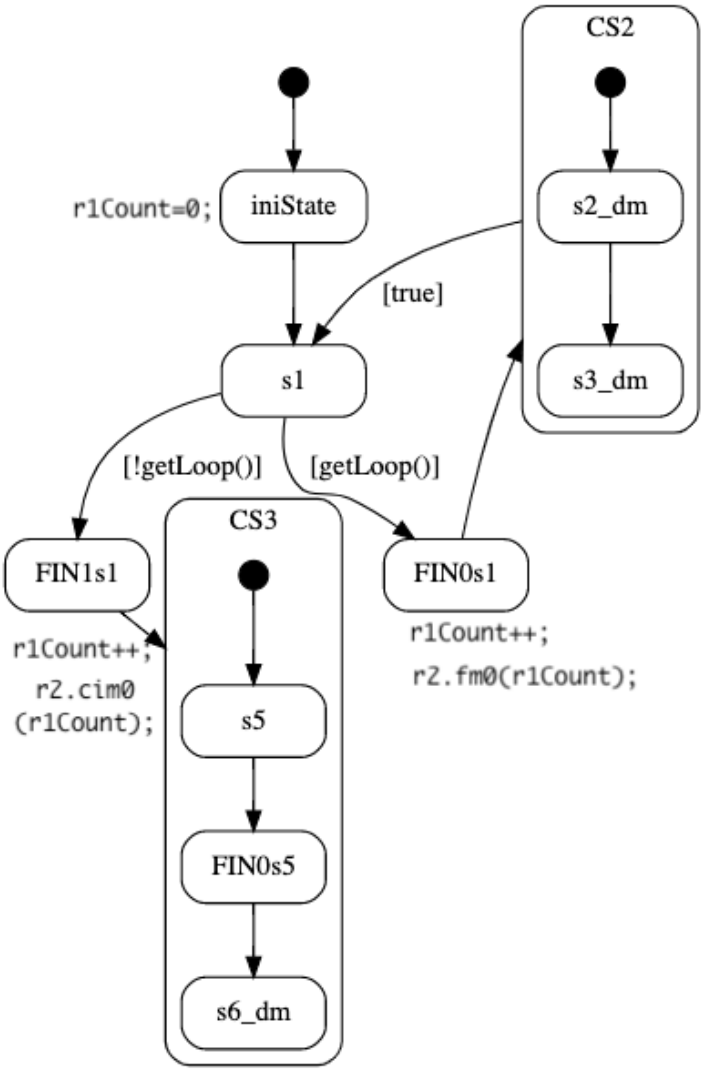


Figure 8.13: Generated design model for the role r_1

model for r_2 in Figure 8.14, the local variables a and b are initialized to -1. The messages fm_0 and cim_0 are received by r_2 with the message parameters var_B stored in the local variable a and var_F stored in b, receptively.

In the design model for r_3 for example (see Figure 8.15 and Listings B.1.3), the role receives the messages fm_2 (sent by r_2) inside the loop, and fm_3 after the loop (sent by r_1). We can see that these messages are sent with a sequence number, the received parameters are stored in the local variables a and b.

For testing the design models, we prepared an Umple file that contains the design models for r_1, r_2, and r_3 together and use this file for Java code generation using Umple

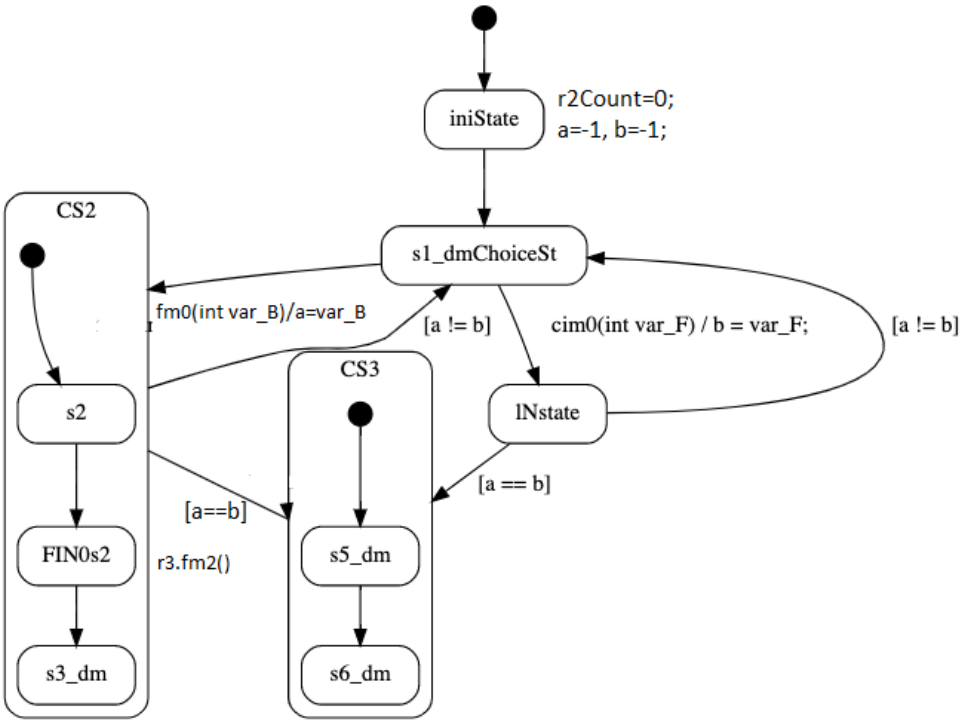


Figure 8.14: Generated design model for the role r_2

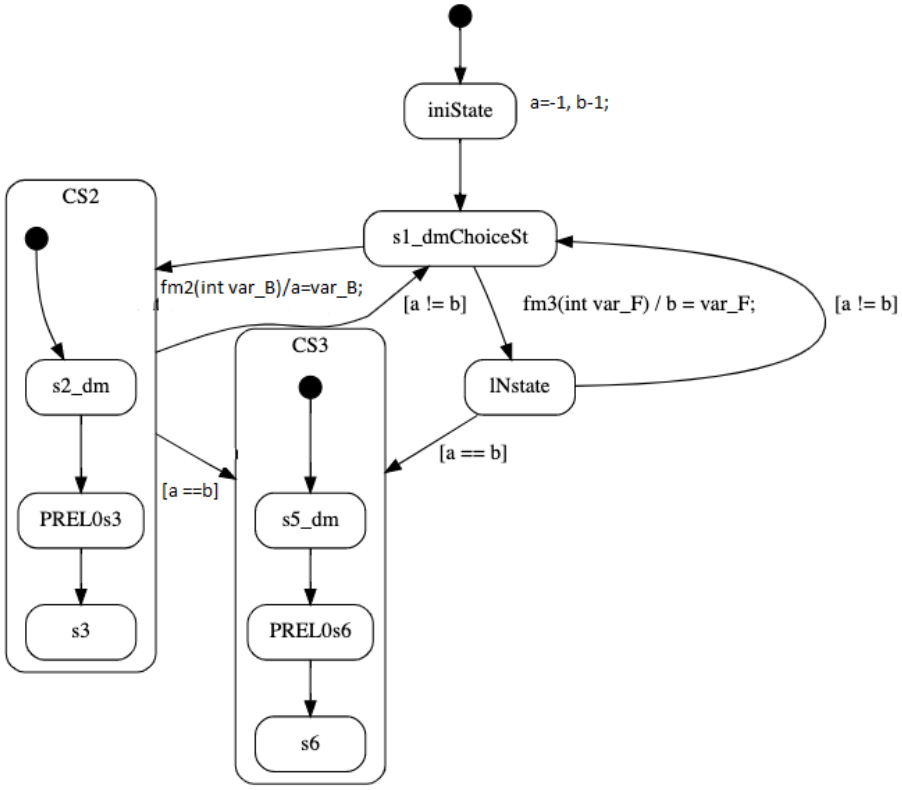


Figure 8.15: Generated design model for the role r_3

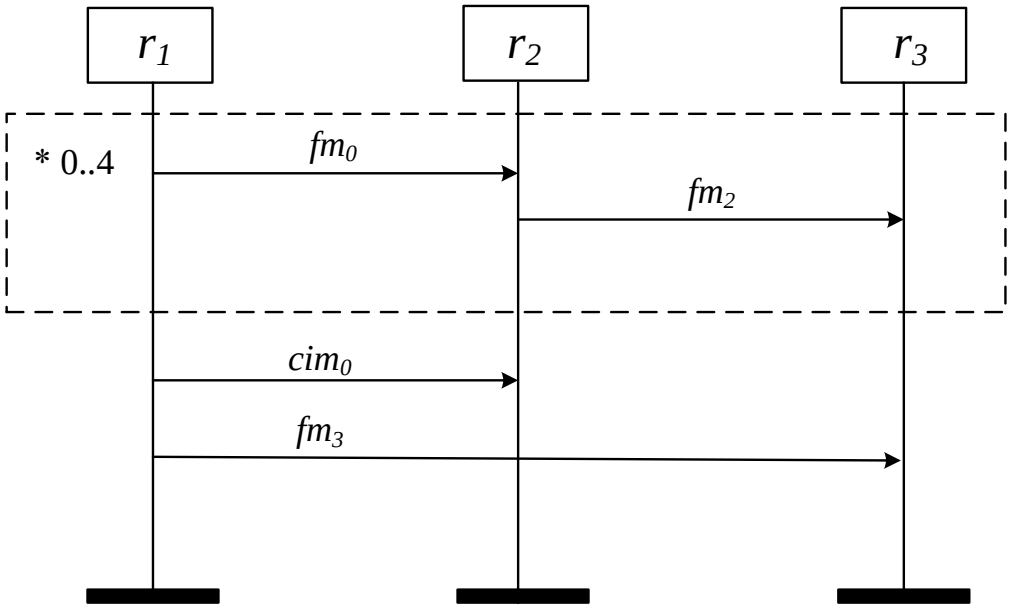


Figure 8.16: A MSC shows the sent and received coordination messages in the design models for the roles r_1, r_2, and r_3.

Online. The Umple file which is used for the code generation is presented in Appendix B (see Section B.2). We added a counter ($nN = 4$) in the design model of role $r1$, in order to execute the loop four times.

The MSC in Figure 8.16 shows the sent and received coordination messages in the design models for the roles r_1, r_2, and r_3 in case of negligible transmission delay (all these messages are sent with a sequence number).

Figure 8.17 shows an example of the execution order of actions for the roles r_1, r_2, and r_3, in the weak while loop. We can see that the messages are received in the expected order. For example, the message fm_3 is received at r_3 after all instances of the message fm_2 are received. Therefore, we note that the message transmission delay is negligible compared to the time of message execution at the different roles.

```
1 after (2)/ {
2     r2Count++;
3     role3.fm2(r2Count);
```

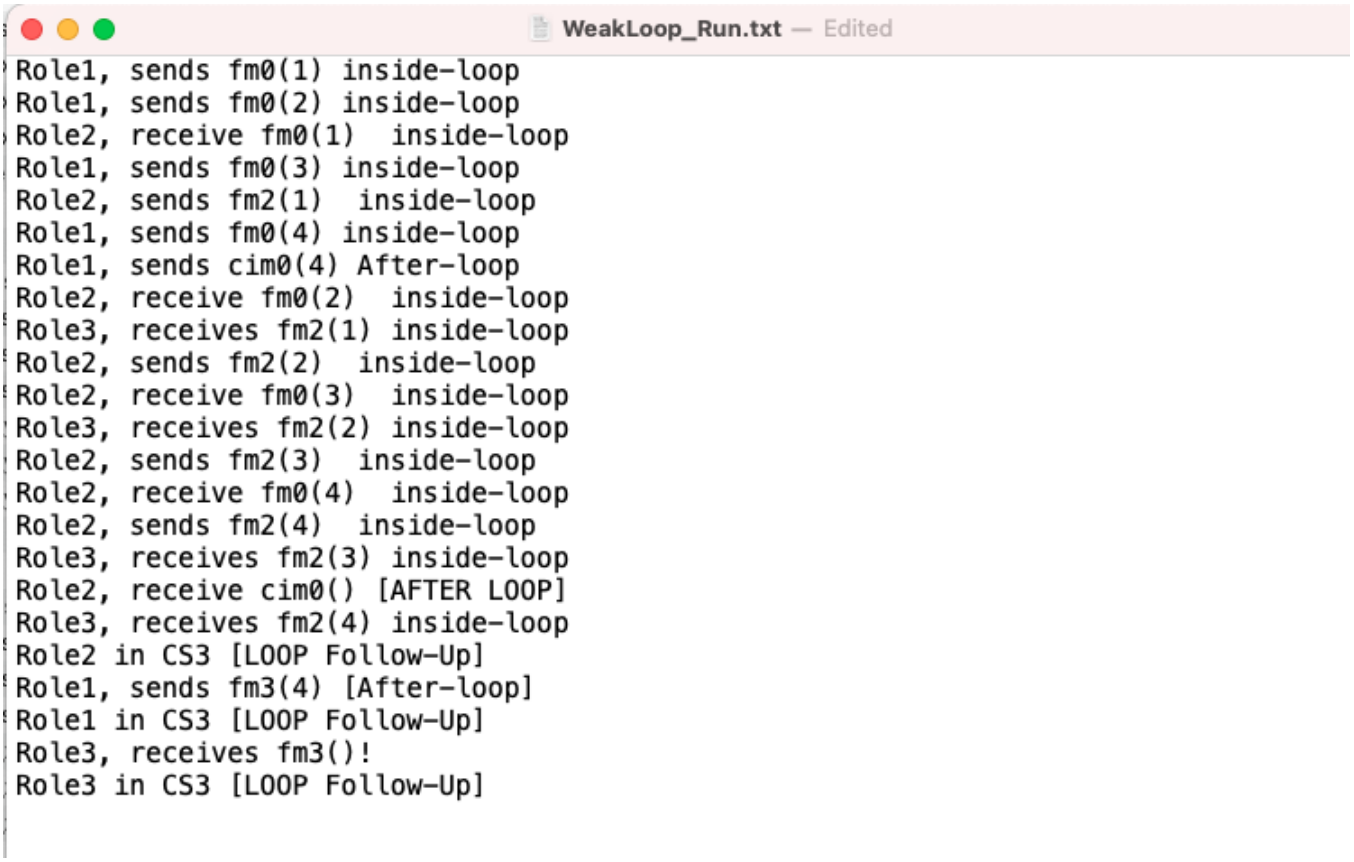


Figure 8.17: The execution order of actions for the roles r_1, r_2, and r_3, in the weak while loop (with no delay)

```
4 }
```

Listing 8.4: Transmission delay example

Since the messages fm_2 and fm_3 received at r_3 are sent by different roles, it could happen that fm_3 is received by r_3 before the last instances of fm_2. If we apply a transmission delay (of two seconds) before sending fm_2 by r_2 (see Listings 8.4), we get the execution order of Figure 8.18. We can see that the message fm_3 is received at r_3 before any instance of the message fm_2. However, fm_3 will not be processed and stays in the reception pool of r_3 until all instances of fm_2 are received and processed.

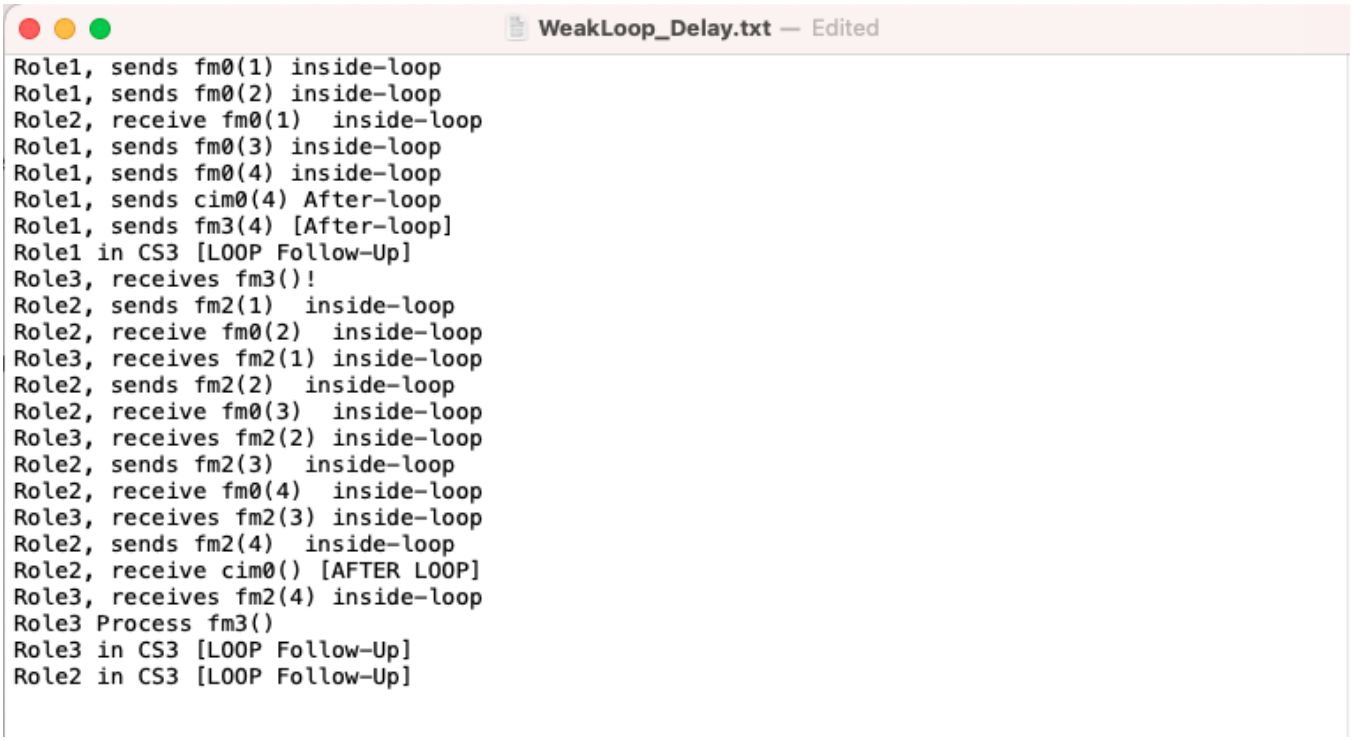


Figure 8.18: The execution order of actions for the roles r_1, r_2, and r_3, in the weak while loop (with delay)

8.8 Chapter Summary

In this chapter, we are testing our tool (see Chapter 7) with a illustrative example of a Travel Management System. We show the generated design models for the different roles which are involved in the system.

In addition, we test the correctness of the generated design models (generated by the tool) for the case study. We have used Zakariapour's distributed implementation environment [85] which is supported by Umple for Java code generation for each design model. It generates a distributable Java code from a UML state machine, which represents the design model for a certain role.

The generated code is executed using Zakariapour's environment in order to test the execution order of actions at each role in a distributed environment and see if this conforms to the correct order of actions in the global model.

Chapter 9

Conclusion and Future Work

9.1 Summary and Contributions

This thesis is concerned with the transformation from a global requirements model, which describes the behavior of a distributed system in an abstract manner by defining the local actions to be performed by the different system roles, to a distributed design model, which defines the behavior of each role separately, including its local actions plus the exchange of coordination messages which are necessary to assure that the actions of the different roles are performed in the required order. This problem is called *realizability* of the global model. In this thesis, first, we consider a global requirements model in the form of partially ordered actions written in the form of a Hierarchical Message Sequence Chart (HMSC) and Collaboration. We study realizability of the global requirements model, and the problems which prevent the global model to be *directly realizable* such as *strict sequence*, *termination race*, *alternatives* if one role does not participate in certain alternative, *competing initiatives* and others. In this context, we improve the derivation algorithm in [4] by minimizing the required coordination messages which are needed by the derived design model in order to conform to the global model. In the derivation of a weak while loop, we show under which conditions the sequence number is not required in the coordination messages. For the derivation of alternatives, we show that the choice indication message (*cim*) is not required in many cases.

In the literature, several notations were proposed to model the behavior of distributed systems such as Message Sequence Chart (MSC), and Hierarchical Message Sequence Chart (HMSC), Collaborations, Activity Diagrams, PO-Charts, Petri-Nets, and others.

In this thesis, we use the Hierarchical Message Sequence Chart (HMSC) [1], Collaboration [7], and UML Hierarchical State Machine (HSM) for modeling the requirements model. HSM is a well-known notation and enables the modeling of complex system behavior concisely; each state is either a simple state or a composite state, which represents another state machine. HSM formally defines the syntax and semantics of concurrency and nesting concepts, which are not allowed by the simple state machine. We introduce certain extensions to HSM in order to describe the different roles that may be involved in the actions performed within one (composed) state and for distinguishing between strict and weak sequencing. The composed state here may involve the actions of different system roles.

We develop an algorithm which does a model transformation from an extended HSM, which represents the global requirements model, into UML HSM, which represent the roles design models, including all required coordination messages between the different roles. The generated state machines can be easily implemented by any suitable tool which supports code generation from a model written in UML HSM. Umple [12] is one of these tools.

Finally, we build a tool which implements our transformation algorithm, where the design models are derived automatically from a global requirements model written in our notation of extended HSMs. In this context, we extend Umple to support our notation for the global model and to derive distributed design models based on our derivation algorithm. Then, a Travel Management System illustrative example is discussed to illustrate the representation of the global model using the extended HSM notation and to demonstrate the correctness of the generated design models by the tool.

The contributions of the thesis can be summarized as follows:

- Improvements to the derivation algorithm in [4], which derives a design models from a global requirements model by minimizing the required coordination messages to

realize the requirements model. In this context, we improve the realizability for the *choice* and *weak while loop*. We show that in many cases, the message sequence number is not required for the realizability of a weak while loop (RQ1), as presented in Chapters 4 and 5.

- Suitable abstract notation for modeling the behavior of distributed applications (global requirements model). Two notations are used: partial order and extended HSMs (RQ2), presented in Chapters 4 and 5.

- An algorithm that generates a model of the behavior of all roles of the distributed application, including the exchange of messages necessary for the coordination of the activities, presented in Chapter 5.

- A study of the problem of competing initiatives in distributed applications and discussion of some distributed design choices for this problem (RQ3), presented in Chapter 6.

- A tool that supports the proposed abstract modeling notation and uses the algorithm to automatically generate the software to implement the application on all its components (RQ4), presented in Chapter 7.

- An illustrative example of a global model using the extended HSM notation and a demonstration of the correctness of the design models generated of by the tool, presented in Chapter 8.

9.2 Threats to Validity

There are several threats to validity that must be assessed to better characterize the limitation of our work.

The main threats are the following:

- The validation is limited to one complex example.

- The solutions do not handle the case where messages may be lost, and where timers are used to conclude that message loss occurred.
- The implementation (Umple code generation) is more deterministic than what is allowed by the global requirements model. For example, the concurrency is constrained by Umple implementation (priority is given to the concurrent state machines from left to right).
- There are limitations for global testing in a distributed environment (as discussed in Section 8.6.3).

9.3 Future Work

There are many opportunities to follow up on the thesis work. We summarize the future work as follows:

- The integration of support for *competing initiatives* (discussed in Chapter 6) in our tool.
- Tool improvements which include (see Section 8.5): dynamic sessions establishment, elimination of dummy states, data transfer between the roles, and reducing the number of *cim* messages.
- Doing further experiments along the lines of those reported in Section 8.6 by: (1) inserting random delays before executing actions, (2) executing the program many times, and (3) measuring the conformance and coverage of the resulting global traces.
- Handling the case where messages are lost and where timers are associated with message reception.
- The following research questions are not discussed in this thesis, and they are considered as future work:

- How can the interruption (see, for instance, [4]) of certain activities be handled in a distributed environment?
- If a non-well-structured activity diagram is used for modeling the global model, can we generate correct distributed design models? (Note: HSM restricts models to be well-structured)

References

[1] R. Alur and M. Yannakakis, "Model checking of message sequence charts," in *International Conference on Concurrency Theory*, vol. 1664 LNCS, 1999, pp. 114–129.

[2] H. N. Castejón, G. v. Bochmann, and R. Bræk, "On the realizability of collaborative services," *Software & Systems Modeling*, vol. 12, no. 3, pp. 597–617, 2013.

[3] G. v. Bochmann, "Conformance Testing with Respect to Partial-Order Specifications," in *IFIP International Conference on Testing Software and Systems*, vol. 9976 LNCS. Springer, Berlin, Heidelberg, 2016, pp. 3–17.

[4] G. v. Bochmann, "Deriving component designs from global requirements," in *Proc. Intern. Workshop on Model Based Architecting and Construction of Embedded Systems (ACES), Toulouse*. Citeseer, 2008.

[5] F. Laamarti, "Derivation Of Component Designs From Global Specifications," Master's thesis, 2011. [Online]. Available: https://ruor.uottawa.ca/handle/10393/28856

[6] N. M. F. Mustafa, "Transformation of UML Activity Diagrams into Business Process Execution Language," Master's thesis, 2011. [Online]. Available: https://ruor.uottawa.ca/handle/10393/20106

[7] T. Israr and G. v. Bochmann, "Performance modeling of distributed collaboration services," *ICPE'11 - Proceedings of the 2nd Joint WOSP/SIPEW International Conference on Performance Engineering*, no. January, pp. 475–480, 2011.

[8] F. Khendek, G. v. Bochmann, and C. Kant, "New results on deriving protocol specifications from service specifications," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4, pp. 136–145, 1989.

[9] H. Ben-Abdallah and S. Leue, "Syntactic detection of process divergence and non-local choice in message sequence charts," in *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1217 LNCS. Springer, 1997, pp. 259–274.

[10] A. Mooij, N. Goga, and J. Romijn, "Non-local Choice and Beyond: Intricacies of MSC Choice Nodes," *Fundamental Approaches to Software Engineering*, vol. 3442 LNCS, pp. 273–288, 2005.

[11] R. Alur, G. J. Holzmann, and D. Peled, "An analyzer for message sequence charts," in *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1055 LNCS. Springer, 1996, pp. 35–48.

[12] Umple, v 1.29.1, 2018. [Online]. Available: http://www.umple.org

[13] G. v. Bochmann and R. Gotzhein, "Deriving protocol specifications from service specifications," *ACM SIGCOMM Computer Communication Review*, vol. 16, no. 3, pp. 148–156, 1986.

[14] V. Pratt, "Modeling concurrency with partial orders," *International Journal of Parallel Programming*, vol. 15, no. 1, pp. 33–71, feb 1986.

[15] ITU-T, "ITU-T Recommendation Z.120 (02/11), Message Sequence Chart (MSC)," ITU, Geneva, Tech. Rep., 2011.

[16] Object Management Group (OMG), "UML 2.5.1 Specification," 2017.

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[18] J.-P. Katoen and L. Lambert, "Pomsets for message sequence charts," *Formale Beschreibungstechniken für Verteilte Systeme*, pp. 197–208, 1998.

[19] J. L. Gischer, "The equational theory of pomsets," *Theoretical Computer Science*, vol. 61, no. 2-3, pp. 199–224, 1988.

[20] Umple online, 2018. [Online]. Available: https://cruise.eecs.uottawa.ca/umpleonline/

[21] Umple meta-model. [Online]. Available: http://metamodel.umple.org/

[22] A. Sultan, "Umple C++ Code Generator," Master's thesis, University of Ottawa, 2011. [Online]. Available: https://ruor.uottawa.ca/handle/10393/26133?mode=full

[23] O. Badreddin, "A manifestation of model-code duality: Facilitating the representation of state machines in the umple model-oriented programming language," Ph.D. dissertation, University Of Ottawa, 2012. [Online]. Available: https://ruor.uottawa.ca/handle/10393/22733

[24] T. C. Lethbridge and R. Laganière, *Object Oriented Software Engineering: Practical Software Development Using UML and Java*, 2nd ed. McGraw Hill, London, 2004.

[25] H. N. Castejón and R. Bræk, "Formalizing collaboration goal sequences for service choreography," in *International Conference on Formal Techniques for Networked and Distributed Systems*, vol. 4229 LNCS. Springer, 2006, pp. 275–291.

[26] H. N. Castejón, R. Braek, and G. v. Bochmann, "Realizability of collaboration-based service specifications," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pp. 73–80, 2007.

[27] A. Mooij, J. Romijn, and W. Wesselink, "Realizability criteria for compositional msc," in *International Conference on Algebraic Methodology and Software Technology*, vol. 4019 LNCS. Springer, 2006, pp. 248–262.

[28] F. Khendek and X. J. Zhang, "From msc to sdl: Overview and an application to the autonomous shuttle transport system," in *In: Leue S., Systä T.J. (eds) Scenarios: Models, Transformations and Tools*. Springer, 2005, vol. 3466 LNCS, pp. 228–254.

[29] B. Mitchell, "Inherent causal orderings of partial order scenarios," in *International Colloquium on Theoretical Aspects of Computing*, vol. 3407 LNCS. Springer, 2004, pp. 113–127.

[30] ——, "Resolving race conditions in asynchronous partial order scenarios," *IEEE Transactions on Software Engineering*, vol. 31, no. 9, pp. 767–784, 2005.

[31] N. M. F. Mustafa and G. v. Bochmann, "Transforming dynamic behavior specifications from activity diagrams to BPEL," *Proceedings - 6th IEEE International Symposium on Service-Oriented System Engineering, SOSE 2011*, pp. 305–311, 2011.

[32] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun, "Infinite-state high-level MSCs: Model-checking and realizability," *Journal of Computer and System Sciences*, vol. 72, no. 4, pp. 617–647, 2006.

[33] M. G. Gouda and Y. T. Yu, "Synthesis of Communicating Finite-State Machines with Guaranteed Progress," *IEEE Transactions on Communications*, vol. 32, no. 7, pp. 779–788, 1984.

[34] M. S. Munir, H. Tanveer, and U. Fatima, "Mixed initiative realizability problem resolution using uml composite states," in *Proceedings of the 2020 9th International Conference on Software and Computer Applications*, 2020, pp. 201–205.

[35] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 623–633, 2003.

[36] ——, "Realizability and verification of MSC graphs," *Theoretical Computer Science*, vol. 331, no. 1, pp. 97–114, 2005.

[37] A. Alghamdi, "Queued and Pooled Semantics for State Machines in the Umple Model-Oriented Programming Language," Master's thesis, University of Ottawa, 2015. [Online]. Available: https://ruor.uottawa.ca/handle/10393/31961

[38] G. v. Bochmann, "Associativity between weak and strict sequencing," in *International Conference on System Analysis and Modeling*, vol. 8769 LNCS. Springer, 2014, pp. 96–109.

[39] P.-Y. Chu and M. T. Liu, "Protocol synthesis in a state-transition model," in *Proceedings COMPSAC 88: The Twelfth Annual International Computer Software & Applications Conference*. IEEE Computer Society, 1988, pp. 505–506.

[40] T. Higashino, K. Okano, H. Imajo, and K. Taniguchi, "Deriving protocol specifications from service specifications in extended fsm models," in *The 13th International Conference on Distributed Computing Systems*. IEEE, 1993, pp. 141–148.

[41] M. Nakamura, Y. Kakuda, and T. Kikuno, "Component-based protocol synthesis from service specifications," *Computer Communications Journal*, vol. 19, no. 14, pp. 1200–1215, 1996.

[42] J.-C. Park and R. E. Miller, "Synthesizing protocol specifications from service specifications in timed extended finite state machines," in *Proceedings of 17th International Conference on Distributed Computing Systems*. IEEE, 1997, pp. 253–260.

[43] J. Al Dallal and K. A. Saleh, "Synthesizing distributed protocol specifications from a uml state machine modeled service specification," *Journal of Computer Science and Technology*, vol. 27, no. 6, pp. 1150–1168, 2012.

[44] J. Al Dallal, "Time assignment for distributed service and protocol UML-based specifications," in *2013 International Conference on Electronics, Computer and Computation (ICECCO)*. IEEE, 2013, pp. 64–67.

[45] ——, "Upss: a tool for synthesizing uml concurrent communication protocol entities," *Advances in Applied Information Science*, vol. 4, pp. 103–108, 2012.

[46] F.-Y. Wang, K. Gildea, H. Jungnitz, and D. D. Chen, "Protocol design and performance analysis for manufacturing message specification: a Petri net approach," *IEEE Transactions on Industrial Electronics*, vol. 41, no. 6, pp. 641–653, 1994.

[47] H. Yamaguchi, K. Okano, T. Higashino, and K. Taniguchi, "Synthesis of protocol entities specifications from service specifications in a Petri net model with registers," in *Proceedings of 15th International Conference on Distributed Computing Systems*. IEEE, 1995, pp. 510–517.

[48] A. Al-Dallal and K. Saleh, "Protocol synthesis using the Petri net model," in *Prof. of 9th Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'97)*, 1997.

[49] H. Yamaguchi, K. El-Fakih, G. v. Bochmann, and T. Higashino, "Protocol synthesis and re-synthesis with optimal allocation of resources based on extended Petri nets," *Distributed Computing*, vol. 16, no. 1, pp. 21–35, 2003.

[50] H. Yamaguchi, K. El-Fakih, G. v. Bochmann, and T. Higashino, "Deriving protocol specifications from service specifications written as predicate/transition-nets," *Computer Networks*, vol. 51, no. 1, pp. 258–284, 2007.

[51] K. El-Fakih, H. Yamaguchi, G. v. Bochmann, and T. Higashino, "Petri net-based protocol synthesis with minimum communication costs," *Journal of the Franklin Institute*, vol. 343, no. 4-5, pp. 501–520, 2006.

[52] C. Kant, T. Higashino, and G. v. Bochmann, "Deriving protocol specifications from service specifications written in LOTOS," *Distributed Computing*, vol. 10, no. 1, pp. 29–47, 1996.

[53] N. Maneerat, R. Varakulsiripunth, D. Seki, K. Yoshida, K. Takahashi, Y. Kato, B. B. Bista, and N. Shiratori, "Composition method of communication system specifications in asynchronous model and its support system," in *Proceedings. Ninth IEEE International Conference on Networks, ICON 2001.* IEEE, 2001, pp. 64–69.

[54] "Information processing systems Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour," International Organization for Standardization, ISO, Standard, Feb. 1989.

[55] L. Hélouët and C. Jard, "Conditions for synthesis of communicating automata from hmscs," in *5th International Workshop on Formal Methods for Industrial Critical Systems*, 2000, pp. 203–224.

[56] L. Hélouët, "Some pathological message sequence charts, and how to detect them," in *International SDL Forum*, vol. 2078 LNCS. Springer, 2001, pp. 348–364.

[57] H. Muccini, "Detecting implied scenarios analyzing non-local branching choices," in *International Conference on Fundamental Approaches to Software Engineering*, vol. 2621 LNCS. Springer, 2003, pp. 372–386.

[58] I.-G. Song, S.-U. Jeon, A.-R. Han, and D.-H. Bae, "An approach to identifying causes of implied scenarios using unenforceable orders," *Information and Software Technology*, vol. 53, no. 6, pp. 666–681, 2011.

[59] F. H. Fard and B. H. Far, "Detection of implied scenarios in multiagent systems with clustering agents' communications," in *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*. IEEE, 2014, pp. 237–244.

[60] C. B. de Melo, A. L. F. Cançado, and G. N. Rodrigues, "Characterization of implied scenarios as families of common behavior," *Journal of Systems and Software*, vol. 158, p. 110425, 2019.

[61] M. Jahan, Z. S. H. Abad, and B. Far, "Detecting emergent behavior in scenario-based specifications using a probabilistic model," in *2020 IEEE Tenth International Model-Driven Requirements Engineering (MoDRE)*. IEEE, 2020, pp. 31–38.

[62] R. Gotzhein and G. v. Bochmann, "Deriving protocol specifications from service specifications including parameters," *ACM Transactions on Computer Systems*, vol. 8, no. 4, pp. 255–283, 1990.

[63] N. Baudru and R. Morin, "Safe implementability of regular message sequence chart specifications." *SNPD*, vol. 3, pp. 210–217, 2003.

[64] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing web services: A survey," *King's College London, Tech. Rep. TR*, 2010.

[65] Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C.K. Liu, S. Thatte, P. Yendluri and A. Y. (Eds.), "Web Services Business Process Execution Language Version 2.0, WS-BPEL TC OASIS," Tech. Rep., 2007.

[Online]. Available: https://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm

[66] M. G. Nanda, S. Chandra, and V. Sarkar, "Decentralizing execution of composite web services," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, vol. 39, No. 10, 2004, pp. 170–187.

[67] "Web services choreography description language version 1.0, w3c working group," Tech. Rep., 2005. [Online]. Available: https://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/

[68] M. Carbone, K. Honda, and N. Yoshida, "A calculus of global interaction based on session types," *Electronic Notes in Theoretical Computer Science*, vol. 171, no. 3, pp. 127–151, 2007.

[69] ——, "Structured communication-centred programming for web services," in *European Symposium on Programming*, vol. 4421 LNCS. Springer, 2007, pp. 2–17.

[70] N. Lohmann and K. Wolf, "Realizability is controllability," in *International Workshop on Web Services and Formal Methods*, vol. 6194 LNCS. Springer, 2009, pp. 110–127.

[71] G. Decker, A. Barros, F. M. Kraft, and N. Lohmann, "Non-desynchronizable service choreographies," in *International Conference on Service-Oriented Computing*, vol. 5364 LNCS. Springer, 2008, pp. 331–346.

[72] N. Roohi, G. Salaün, and S. H. Mirian, "Analyzing Chor Specifications by Translation into FSP," *Electronic Notes in Theoretical Computer Science*, vol. 255, pp. 159–176, 2009.

[73] N. Roohi and G. Salaün, "Realizability and dynamic reconfiguration of chor specifications," *Informatica (Ljubljana)*, vol. 35, no. 1, pp. 39–49, 2011.

[74] Qiu Zongyan, Zhao Xiangpeng, Cai Chao and Y. Hongli, "Towards the Theoretical Foundation of Choreography," in *the 16th international conference on World Wide Web*. ACM, 2007, pp. 973–982.

[75] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot, "A theoretical basis of communication-centred concurrent programming," *Web Services Choreography Working Group mailing list, to appear as a WS-CDL working report*, pp. 65–66, 2006.

[76] M. F. Al-hammouri and G. v. Bochmann, "Realizability of service specifications," in *International Conference on System Analysis and Modeling (SAM 2018)*, vol. 11150 LNCS. Springer, 2018, pp. 127–143.

[77] M. F. Al-hammouri and G. v. Bochmann, "Deriving distributed design models from global state machines requirements," in *International Conference on System Analysis and Modeling*, vol. 11753 LNCS. Springer, 2019, pp. 27–43.

[78] S. Mauw and M. A. Reniers, "High-level message sequence charts," in *SDL'97: Time for Testing*. Elsevier, 1997, pp. 291–306.

[79] A. Barros, M. Dumas, and P. Oaks, "Standards for web service choreography and orchestration: Status and perspectives," in *International Conference on Business Process Management*, vol. 3812 LNCS. Springer, 2005, pp. 61–74.

[80] O. Badreddin, T. C. Lethbridge, A. Forward, M. Elaasar, H. Aljamaan, and M. A. Garzon, "Enhanced code generation from uml composite state machines," in *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, 2014, pp. 235–245.

[81] M. Mahoney and T. Elrad, "Distributing state-charts to handle pervasive crosscutting concerns," in *Proceeding of Building Software for Pervasive Computing Workshop. OOPSLA*. Citeseer, 2005.

[82] Wikipedia contributors, "Structured programming," 2019, [Online; last edited on 4-July-2019]. [Online]. Available: https://en.wikipedia.org/wiki/Structured_programming

[83] I. Toqeer, "Modeling and performance analysis of distributed systems with collaboration behaviour diagrams," Ph.D. dissertation, University Of Ottawa, 2014. [Online]. Available: http://hdl.handle.net/10393/30950

[84] G. v. Bochmann, "Compositional Development of Distributed Applications," Amman, Jordan, 2009. [Online]. Available: http://www.site.uottawa.ca/~bochmann/Curriculum/Pub/Slides/Deriving-3%20-%20Amman.pdf

[85] A. Zakariapour, "Model-Driven Development of Distributed Systems in Umple," Master's thesis, University Of Ottawa, 2018. [Online]. Available: http://hdl.handle.net/10393/37143

[86] R. Hu, N. Yoshida, and K. Honda, "Session-based distributed programming in java," in *European Conference on Object-Oriented Programming*, vol. 5142 LNCS. Springer, 2008, pp. 516–541.

[87] M. Carbone, K. Honda, and N. Yoshida, "Structured communication-centered programming for web services," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 34, no. 2, pp. 1–78, 2012.

[88] B. Sarikaya and G. v. Bochmann, "Synchronization and specification issues in protocol testing," *IEEE Transactions on Communications*, vol. 32, no. 4, pp. 389–395, 1984.

[89] A. Ghedamsi, G. v. Bochmann, and R. Dssouli, "Diagnosing distributed systems modeled by communicating finite state machines," vol. 3, no. 4. Revue Réseaux et Informatique Répartie, 1993, pp. 343—-363.

[90] Z. Pap, G. Csopaki, and S. Dibuz, "On FSM-based fault diagnosis," in *IFIP International Conference on Testing of Communicating Systems*, vol. 3502 LNCS. Springer, 2005, pp. 159–174.

APPENDICES (Umple Files)

Appendix A

Travel Management System Case Study

A.1 The Generated Design Models in the Travel Management System (Umple syntax)

In this section, we show the Umple syntax for the derived design models generated by our tool for the participating roles in the Travel Management System case study.

A.1.1 The Hotel Class

```
class Hotel
{
  Sm_S {
    Login {
      [true]  -> SearchData;
        login_dm {
           -> validateUser_dm;
        }
        validateUser_dm {
           -> confirmed_dm;
```

```
11          }
12          final confirmed_dm {        }
13      }
14      SearchData {
15        [true]  -> Query;
16          enterDatesDestination_dm {
17             -> confirmsData_dm;
18          }
19          final confirmsData_dm {        }
20      }
21      Query {
22        [true]  -> appendProfits_dm;
23          queryReqHotelSys_dm {
24             -> PREL0prepareHotelOptions;
25          }
26          PREL0prepareHotelOptions {
27            fm3 -> prepareHotelOptions;
28          }
29          prepareHotelOptions {
30            entry / { System.out.println("Hotel System: prepare Hotel
    options"); }
31             -> FIN0prepareHotelOptions;
32          }
33          FIN0prepareHotelOptions {
34            do { m.fm4(); } -> confirmHotelOptions_dm;
35          }
36          final confirmHotelOptions_dm {        }
37      ||
38          queryReqAirSys_dm {
39             -> prepareAirOptions_dm;
40          }
41          prepareAirOptions_dm {
42             -> confirmAirOptions_dm;
43          }
44          final confirmAirOptions_dm {        }
45      }
```

```
46    appendProfits_dm {
47       -> DisOptions;
48    }
49    DisOptions {
50      [true]  -> chooseResult_dmChoiceSt;
51        prepareChoices_dm {
52           -> viewChoices_dm;
53        }
54        final viewChoices_dm {        }
55    }
56    chooseResult_dmChoiceSt {
57      cimC2 -> SearchData;
58      cimC5 -> Payment;
59    }
60    Payment {
61      [true]  -> confirmClientSelection;
62        pay_dm {
63           -> receivePayment_dm;
64        }
65        receivePayment_dm {        }
66    }
67    confirmClientSelection {        confirmHotelSelection_dm {
68           -> PREL0confirmReservationHotel;
69        }
70        PREL0confirmReservationHotel {
71          fm9 -> confirmReservationHotel;
72        }
73        final confirmReservationHotel {
74          entry / { System.out.println("Hotel System: confirm Reservation
    "); }
75        }
76    ||
77        confirmAirSelection_dm {
78           -> confirmReservationAir_dm;
79        }
80        final confirmReservationAir_dm {        }
```

```
    }
  }
}
```

A.1.2 The Airline Class

```
class Air
{
  Sm_S {
    Login {
      [true]  -> SearchData;
        login_dm {
           -> validateUser_dm;
        }
        validateUser_dm {
           -> confirmed_dm;
        }
        final confirmed_dm {        }
    }
    SearchData {
      [true]  -> Query;
        enterDatesDestination_dm {
           -> confirmsData_dm;
        }
        final confirmsData_dm {        }
    }
    Query {
      [true]  -> appendProfits_dm;
        queryReqHotelSys_dm {
           -> prepareHotelOptions_dm;
        }
        prepareHotelOptions_dm {
           -> confirmHotelOptions_dm;
        }
        final confirmHotelOptions_dm {        }
```

```
30      ||
31          queryReqAirSys_dm {
32              -> PREL0prepareAirOptions;
33          }
34          PREL0prepareAirOptions {
35            fm5 -> prepareAirOptions;
36          }
37          prepareAirOptions {
38            entry / { System.out.println("Air System: prepare Air options."
    ); }
39             -> FIN0prepareAirOptions;
40          }
41          FIN0prepareAirOptions {
42            do { m.fm6(); } -> confirmAirOptions_dm;
43          }
44          final confirmAirOptions_dm {        }
45      }
46      appendProfits_dm {
47         -> DisOptions;
48      }
49      DisOptions {
50        [true]  -> chooseResult_dmChoiceSt;
51          prepareChoices_dm {
52              -> viewChoices_dm;
53          }
54          final viewChoices_dm {        }
55      }
56      chooseResult_dmChoiceSt {
57        cimC1 -> SearchData;
58        cimC4 -> Payment;
59      }
60      Payment {
61        [true]  -> confirmClientSelection;
62          pay_dm {
63              -> receivePayment_dm;
64          }
```

```
65          receivePayment_dm {         }
66      }
67      confirmClientSelection {
68          confirmHotelSelection_dm {
69              -> confirmReservationHotel_dm;
70          }
71          final confirmReservationHotel_dm {         }
72      ||
73          confirmAirSelection_dm {
74              -> PREL0confirmReservationAir;
75          }
76          PREL0confirmReservationAir {
77            fm10 -> confirmReservationAir;
78          }
79          final confirmReservationAir {
80            entry / { System.out.println("Air System: confirm Reservation")
      ; }
81          }
82      }
83    }
84  }
```

A.1.3 The Client Class

```
1  class Client
2  {
3    Sm_S {
4      Login {
5        [true]  -> SearchData;
6          login {
7            entry / { System.out.println("Client: login."); }
8             -> FIN0login;
9          }
10         FIN0login {
11           do { m.fm0(); } -> validateUser_dm;
```

```
12          }
13          validateUser_dm {
14              -> PREL0confirmed;
15          }
16          PREL0confirmed {
17            fm1 -> confirmed;
18          }
19          final confirmed {
20            entry / { System.out.println("Client: login confirmed."); }
21          }
22      }
23      SearchData {
24        [true]  -> Query;
25          enterDatesDestination {
26            entry / { System.out.println("Client: enter flight data: Dates
    and Destination"); }
27              -> FIN0enterDatesDestination;
28          }
29          FIN0enterDatesDestination {
30            do { m.fm2(); } -> confirmsData_dm;
31          }
32          final confirmsData_dm {        }
33      }
34      Query {
35        [true]  -> appendProfits_dm;
36          queryReqHotelSys_dm {
37              -> prepareHotelOptions_dm;
38          }
39          prepareHotelOptions_dm {
40              -> confirmHotelOptions_dm;
41          }
42          final confirmHotelOptions_dm {        }
43      ||
44          queryReqAirSys_dm {
45              -> prepareAirOptions_dm;
46          }
```

```
47          prepareAirOptions_dm {
48              -> confirmAirOptions_dm;
49          }
50          final confirmAirOptions_dm {        }
51      }
52      appendProfits_dm {
53         -> DisOptions;
54      }
55      DisOptions {
56        [true]  -> chooseResult;
57          prepareChoices_dm {
58              -> PREL0viewChoices;
59          }
60          PREL0viewChoices {
61            fm7 -> viewChoices;
62          }
63          final viewChoices {
64            entry / { System.out.println("Client: view choices."); }
65          }
66      }
67      chooseResult {
68        [!acceptable]  -> FIN0chooseResult;
69        [acceptable]  -> FIN1chooseResult;
70      }
71      FIN0chooseResult {
72        do { m.cimC0();a.cimC1();h.cimC2(); } -> SearchData;
73      }
74      FIN1chooseResult {
75        do { m.cimC3();a.cimC4();h.cimC5();m.cim0();a.cim1();h.cim2(); } ->
        Payment;
76      }
77      Payment {
78        [true]  -> confirmClientSelection;
79          pay {
80            entry / { System.out.println("Client: Pay."); }
81             -> FIN0pay;
```

```
82          }
83          FINOpay {
84            do { m.fm8(); } -> receivePayment_dm;
85          }
86          receivePayment_dm {        }
87      }
88      confirmClientSelection {        confirmHotelSelection_dm {
89              -> confirmReservationHotel_dm;
90          }
91          final confirmReservationHotel_dm {        }
92      ||
93          confirmAirSelection_dm {
94              -> confirmReservationAir_dm;
95          }
96          final confirmReservationAir_dm {        }
97      }
98    }
99  }
```

A.1.4 The Management System Class

```
1  class Management
2  {
3    Sm_S {
4      Login {
5        [true]  -> SearchData;
6          login_dm {
7              -> PREL0validateUser;
8          }
9          PREL0validateUser {
10           fm0 -> validateUser;
11         }
12         validateUser {
13           entry / { System.out.println("Management System: validate User.
     "); }
```

```
14            -> FIN0validateUser;
15        }
16        FIN0validateUser {
17          do { c.fm1(); } -> confirmed_dm;
18        }
19        final confirmed_dm {        }
20    }
21    SearchData {
22      [true]  -> Query;
23        enterDatesDestination_dm {
24            -> PREL0confirmsData;
25        }
26        PREL0confirmsData {
27          fm2 -> confirmData;
28        }
29        final confirmData {
30          entry / { System.out.println("Management System: confirm data."
    ); }
31        }
32    }
33    Query {
34      [true]  -> appendProfits;
35        queryReqHotelSys {
36          entry / { System.out.println("Management System: Query Request
    Hotel System."); }
37            -> FIN0queryReqHotelSys;
38        }
39        FIN0queryReqHotelSys {
40          do { h.fm3(); } -> prepareHotelOptions_dm;
41        }
42        prepareHotelOptions_dm {
43            -> PREL0confirmHotelOptions;
44        }
45        PREL0confirmHotelOptions {
46          fm4 -> confirmHotelOptions;
47        }
```

```
48          final confirmHotelOptions {
49            entry / { System.out.println("Management System: confirm Hotel
    options.  "); }
50          }
51       ||
52          queryReqAirSys {
53            entry / { System.out.println("Management System: Query Request
    Air System."); }
54             -> FIN0queryReqAirSys;
55          }
56          FIN0queryReqAirSys {
57            do { a.fm5(); } -> prepareAirOptions_dm;
58          }
59          prepareAirOptions_dm {
60             -> PREL0confirmAirOptions;
61          }
62          PREL0confirmAirOptions {
63            fm6 -> confirmAirOptions;
64          }
65          final confirmAirOptions {
66            entry / { System.out.println("Management System: confirm Air
    options."); }
67          }
68       }
69       appendProfits {
70         entry / { System.out.println("Management System: append profits.");
       }
71          -> DisOptions;
72       }
73       DisOptions {
74         [true]  -> chooseResult_dmChoiceSt;
75          prepareChoices {
76            entry / { System.out.println("Management System: prepare
    choices."); }
77             -> FIN0prepareChoices;
78          }
```

```
79          FIN0prepareChoices {
80            do { c.fm7(); } -> viewChoices_dm;
81          }
82          final viewChoices_dm {        }
83      }
84      chooseResult_dmChoiceSt {
85        cimC0 -> SearchData;
86        cimC3 -> Payment;
87      }
88      Payment {
89        [true]  -> confirmClientSelection;
90          pay_dm {
91             -> PREL0receivePayment;
92          }
93          PREL0receivePayment {
94            fm8 -> receivePayment;
95          }
96          receivePayment {
97            entry / { System.out.println("Management System: receive
    payment."); }
98          }
99      }
100     confirmClientSelection {        confirmHotelSelection {
101           entry / { System.out.println("Management System: confirm Hotel
    selection."); }
102            -> FIN0confirmHotelSelection;
103         }
104         FIN0confirmHotelSelection {
105           do { h.fm9(); } -> confirmReservationHotel_dm;
106         }
107         final confirmReservationHotel_dm {        }
108     ||
109         confirmAirSelection {
110           entry / { System.out.println("Management System: confirm Air
    selection."); }
111            -> FIN0confirmAirSelection;
```

```
112         }
113         FIN0confirmAirSelection {
114           do { a.fm10(); } -> confirmReservationAir_dm;
115         }
116         final confirmReservationAir_dm {        }
117     }
118   }
119 }
```

A.2 Umple File for Testing the Travel Management System Case Study

For the Travel Management System case study. We prepare an Umple file contains the derived design models in Section A.1 together. The design models are represented as classes and must be *distributable* in order to generated a distributable code for the Travel Management System. We use Umple online [20] to generate a Java code from this Umple file.

```
1  namespace Roles;
2  class Client
3  {
4    1 -- 0..1 Hotel;
5    1 -- 0..1 Air;
6    distributable;
7    boolean acceptable = false;
8    pooled Sm_Client {
9      Login {
10         login {
11           entry / { System.out.println("Client: Login."); }
12            -> FIN0login;
13         }
14         FIN0login {
15           entry / { management.fm0(); } -> validateUser_dm;
16         }
```

```
17          validateUser_dm {
18              -> PREL0confirmed;
19          }
20          PREL0confirmed {
21            fm1 -> confirmed;
22          }
23          confirmed {
24            entry / { System.out.println("Client: Login confirmed."); }
25        -> SearchData;
26          }
27      }
28      SearchData {
29          enterDatesDestination {
30            entry / { System.out.println("Client: Enter flight data, dates
    and destination"); }
31             -> FIN0enterDatesDestination;
32          }
33          FIN0enterDatesDestination {
34            entry / { management.fm2(); } -> confirmsData_dm;
35          }
36          confirmsData_dm {
37        -> Query_dm;
38      }
39      }
40      Query_dm{
41        -> appendProfits_dm;
42
43      }
44      appendProfits_dm {
45         -> DisOptions;
46      }
47      DisOptions {
48          prepareChoices_dm {
49              -> PREL0viewChoices;
50          }
51          PREL0viewChoices {
```

```
52          fm7 -> viewChoices;
53        }
54        viewChoices {
55          entry / { System.out.println("Client: View choices."); }
56      -> chooseResult;
57        }
58    }
59    chooseResult {
60    entry /
61      {
62         System.out.println("Client: Choose Result");
63         System.out.println("Accepted? (true or false?)");
64         Scanner myInput = new Scanner( System.in );
65         acceptable = myInput.nextBoolean();
66         System.out.println("Accepted?"+acceptable);
67      }
68      [!acceptable]  -> FIN0chooseResult;
69      [acceptable]  -> FIN1chooseResult;
70    }
71    FIN0chooseResult {
72      entry / { management.cimC0(); air.cimC1();hotel.cimC2(); } ->
    SearchData;
73    }
74    FIN1chooseResult {
75      entry / { management.cimC3(); air.cimC4(); hotel.cimC5(); } ->
    Payment;
76    }
77    Payment {
78        pay {
79          entry / { System.out.println("Client: Pay."); }
80           -> FIN0pay;
81        }
82        FIN0pay {
83          entry / { management.fm8(); } -> receivePayment_dm;
84        }
85        receivePayment_dm {
```

```
86        -> confirmClientSelection_dm;
87      }
88      }
89      final confirmClientSelection_dm {
90      }
91    }
92  }
93  class Management
94  {
95    1 -- 0..1 Client;
96    1 -- 0..1 Hotel;
97    1 -- 0..1 Air;
98    distributable;
99    boolean qH = false; // query Hotel
100   boolean qA = false; // query Air
101   boolean ccs_H = false; // confirm client selection Hotel
102   boolean ccs_A = false; // confirm client selection Air
103   boolean qryStDone = false;
104   boolean ccsStDone = false;
105   pooled Sm_Manager {
106     Login {
107         login_dm {
108            -> PREL0validateUser;
109         }
110         PREL0validateUser {
111           fm0 -> validateUser;
112         }
113         validateUser {
114           entry / { System.out.println("Management System: Validate User.
      "); }
115            -> FIN0validateUser;
116         }
117         FIN0validateUser {
118            after(1) / { client.fm1(); } -> confirmed_dm;
119         }
120         confirmed_dm {
```

```
121       -> SearchData;
122     }
123     }
124     SearchData {
125         enterDatesDestination_dm {
126            -> PREL0confirmsData;
127         }
128         PREL0confirmsData {
129           fm2 -> confirmData;
130         }
131         confirmData {
132           entry / { System.out.println("Management System: Confirm search
    data."); }
133       -> Query;
134         }
135     }
136     Query {
137         queryReqHotelSys {
138           entry / { System.out.println("Management System: Query Request
    Hotel System."); }
139            -> FIN0queryReqHotelSys;
140         }
141         FIN0queryReqHotelSys {
142           entry / { hotel.fm3(); } -> prepareHotelOptions_dm;
143         }
144         prepareHotelOptions_dm {
145            -> PREL0confirmHotelOptions;
146         }
147         PREL0confirmHotelOptions {
148           fm4 -> confirmHotelOptions;
149         }
150         confirmHotelOptions {
151           entry / {
152       qH= true;
153       qryStDone = qH && qA;
```

```
154       System.out.println("Management System: Confirm Hotel options.  ");
      }
155       [qryStDone] -> appendProfits;
156         }
157      ||
158         queryReqAirSys {
159           entry / { System.out.println("Management System: Query Request
      Air System."); }
160            -> FIN0queryReqAirSys;
161         }
162         FIN0queryReqAirSys {
163           entry / { air.fm5(); } -> prepareAirOptions_dm;
164         }
165         prepareAirOptions_dm {
166            -> PREL0confirmAirOptions;
167         }
168         PREL0confirmAirOptions {
169           fm6 -> confirmAirOptions;
170         }
171         confirmAirOptions {
172           entry / {
173       qA = true;
174       qryStDone = qH && qA;
175       System.out.println("Management System: Confirm Air options."); }
176       [qryStDone] -> appendProfits;
177         }
178      }
179      appendProfits {
180        entry / { System.out.println("Management System: Append profits.");
      }
181         -> DisOptions;
182      }
183      DisOptions {
184         prepareChoices {
185           entry / { System.out.println("Management System: Prepare
      choices."); }
```

```
186             -> FIN0prepareChoices;
187         }
188         FIN0prepareChoices {
189           entry / { client.fm7(); } -> viewChoices_dm;
190         }
191         viewChoices_dm {
192       -> chooseResult_dmChoiceSt;
193   }
194     }
195     chooseResult_dmChoiceSt {
196       cimC0 -> SearchData;
197       cimC3 -> Payment;
198     }
199   Payment {
200         pay_dm {
201             -> PREL0receivePayment;
202         }
203         PREL0receivePayment {
204           fm8 -> receivePayment;
205         }
206         receivePayment {
207           entry / { System.out.println("Management System: Receive
    payment."); }
208       -> confirmClientSelection;
209         }
210     }
211   confirmClientSelection {
212
213     confirmHotelSelection {
214           entry / { System.out.println("Management System: Confirm Hotel
    selection."); }
215             -> FIN0confirmHotelSelection;
216         }
217         FIN0confirmHotelSelection {
218           entry /{ hotel.fm9(); } -> confirmReservationHotel_dm;
219         }
```

```
220         final confirmReservationHotel_dm {        }
221     ||
222         confirmAirSelection {
223           entry / { System.out.println("Management System: Confirm Air
    selection."); }
224            -> FIN0confirmAirSelection;
225         }
226         FIN0confirmAirSelection {
227           entry /{ air.fm10(); } -> confirmReservationAir_dm;
228         }
229         final confirmReservationAir_dm {        }
230     }
231   }
232 }
233 class Air
234 {
235   distributable;
236   pooled Sm_AirSystem {
237     Login_dm {
238       -> SearchData_dm;
239     }
240     SearchData_dm {
241       -> Query;
242     }
243     Query {
244         queryReqAirSys_dm {
245            -> PREL0prepareAirOptions;
246         }
247         PREL0prepareAirOptions {
248           fm5 -> prepareAirOptions;
249         }
250         prepareAirOptions {
251       entry / {System.out.println("Air System: Prepare Air options.");}
252            -> FIN0prepareAirOptions;
253         }
254         FIN0prepareAirOptions {
```

```
255           entry / { management.fm6(); } -> confirmAirOptions_dm;
256         }
257         confirmAirOptions_dm {
258       -> appendProfits_dm;
259
260     }
261     }
262     appendProfits_dm {
263        -> DisOptions_dm;
264     }
265     DisOptions_dm {
266      -> chooseResult_dmChoiceSt;
267     }
268     chooseResult_dmChoiceSt {
269       cimC1 -> SearchData_dm;
270       cimC4 -> Payment_dm;
271     }
272   Payment_dm{
273        -> confirmClientSelection;
274     }
275     confirmClientSelection {
276         confirmAirSelection_dm {
277            -> PREL0confirmAirReservation;
278         }
279         PREL0confirmAirReservation {
280           fm10 -> confirmAirReservation;
281         }
282         final confirmAirReservation {
283           entry / {System.out.println("Air System: Confirm Reservation")
      ;}
284        }
285     }
286   }
287 }
288 class Hotel
289 {
```

```
290 distributable;
291 pooled Sm_Hotel {
292   Login_dm {
293      -> SearchData_dm;
294   }
295   SearchData_dm {
296      -> Query;
297   }
298   Query {
299       queryReqHotelSys_dm {
300          -> PREL0prepareHotelOptions;
301       }
302       PREL0prepareHotelOptions {
303         fm3 -> prepareHotelOptions;
304       }
305       prepareHotelOptions {
306     entry / { System.out.println("Hotel System: Prepare Hotel options."
    ); }
307          -> FIN0prepareHotelOptions;
308       }
309       FIN0prepareHotelOptions {
310         entry / { management.fm4(); } -> confirmHotelOptions_dm;
311       }
312       confirmHotelOptions_dm {
313    -> appendProfits_dm;
314   }
315   }
316   appendProfits_dm {
317      -> DisOptions;
318   }
319   DisOptions {
320       -> chooseResult_dmChoiceSt;
321   }
322   chooseResult_dmChoiceSt {
323     cimC2 -> SearchData_dm;
324     cimC5 -> Payment_dm;
```

```
325     }
326   Payment_dm {
327         -> confirmClientSelection;
328     }
329     confirmClientSelection {
330         confirmHotelSelection_dm {
331            -> PREL0confirmHotelReservation;
332         }
333         PREL0confirmHotelReservation {
334           fm9 -> confirmHotelReservation;
335         }
336         final confirmHotelReservation {
337       entry / { System.out.println("Hotel System: Confirm Reservation");
     }
338         }
339     }
340   }
341 }
342 class Test {
343
344   public static void main(String[] args) {
345       Management _m = new Management();
346     Client _c = new Client(_m);
347       Air _a = new Air(_c,_m);
348     Hotel _h = new Hotel(_c,_m);
349   }
350 }
```

Appendix B

Weak Wile Loop Umple Files

B.1 Generated Design Models for the Weak While Loop

In this section, we show the Umple files of the generated design models for the roles r_1, r_2, and r_3, in the weak while loop in Figure 8.12.

B.1.1 The Role r_1 Class

```
class Role1
{
  boolean loop;
  Integer r1Count = 0;
  Sm_WL {
    iniState {
      entry / { r1Count=0; }
       -> s1;
    }
    s1 {
      [getLoop()]  -> FIN0s1;
      [!getLoop()]  -> FIN1s1;
    }
```

```
14      FIN0s1 {
15        do { r1Count++;  r2.fm0(r1Count); } -> CS2;
16      }
17      FIN1s1 {
18        do { r1Count++;  r2.cim0(r1Count); } -> CS3;
19      }
20      CS2 {
21        [true]  -> s1;
22          s2_dm {
23             -> s3_dm;
24          }
25          final s3_dm {        }
26      }
27      CS3 {        s5 {
28            entry / { System.out.println("after  loop s5"); }
29             -> FIN0s5;
30          }
31          FIN0s5 {
32            do { r3.fm3(r1Count); } -> s6_dm;
33          }
34          final s6_dm {        }
35      }
36    }
37  }
```

B.1.2 The Role r_2 Class

```
1  class Role2
2  {
3    boolean loop;
4    Integer a = -1;
5    Integer b = -1;
6    Sm_WL {
7      iniState {
8        entry / {
```

```
9         r2Count=0;
10        a = -1;
11          b = -1;
12      }
13         -> s1_dmChoiceSt;
14      }
15      s1_dmChoiceSt {
16        fm0(int var_B)/{ a = var_B;} -> CS2;
17        cim0(int var_F)/{ b = var_F;} -> lNstate;
18      }
19      CS2 {
20        [a != b] -> s1_dmChoiceSt;
21        [a == b] -> CS3;
22          s2 {
23            entry / { System.out.println("in loop s2_r1"); }
24             -> FIN0s2;
25          }
26          FIN0s2 {
27            do { r2Count++;  r3.fm2(r2Count); } -> s3_dm;
28          }
29          final s3_dm {         }
30      }
31      lNstate {
32        [a != b] -> s1_dmChoiceSt;
33        [a == b] -> CS3;
34      }
35      CS3 {
36      s5_dm {
37              -> s6_dm;
38          }
39          final s6_dm {         }
40      }
41    }
42  }
```

B.1.3 The Role r_3 Class

```
class Role3
{
  boolean loop;
  Integer a = -1;
  Integer b = -1;
  Sm_WL {
    iniState {
      entry / { r3Count=0;
              a = -1;
              b = -1;
    }
       -> s1_dmChoiceSt;
    }
    s1_dmChoiceSt {
      fm2(int var_B)/{ a = var_B;} -> CS2;
      fm3(int var_F)/{ b = var_F;} -> lNstate;
    }
    CS2 {
      [a != b] -> s1_dmChoiceSt;
      [a == b] -> CS3;
        s2_dm {
           -> PREL0s3;
        }
        PREL0s3 {
           -> s3;
        }
        final s3 {        }
    }
    lNstate {
      [a != b] -> s1_dmChoiceSt;
      [a == b] -> CS3;
    }
    CS3 {        s5_dm {
           -> PREL0s6;
```

```
35          }
36          PREL0s6 {
37              -> s6;
38          }
39          final s6 {         }
40      }
41    }
42 }
```

B.2 Umple File for Testing the Weak While Loop

This section shows the Umple file which contains the design models for the participating roles together (r_1, r_2, and r_3) in the weak while loop (see Figure 8.12).

```
1  class Role1
2  {
3    boolean loop=true;
4    Integer r1Count = 0;
5    Integer nN = 4;
6    pooled Sm_x {
7      iniState {
8        entry / { r1Count=0; }
9         -> s1;
10     }
11     s1 {
12       [nN != 0]/{nN--;}  -> FIN0s1;
13       [nN == 0]   -> FIN1s1;
14     }
15     FIN0s1 {
16       entry / {
17         r1Count++;
18         System.out.println("Role1, sends fm0("+r1Count+") inside-loop");
19     role2.fm0(r1Count); } -> CS2;
20     }
21     FIN1s1 {
```

```
22        entry / {
23          System.out.println("Role1, sends cim0("+r1Count+") After-loop");
24          role2.cim0(r1Count); } -> CS3;
25      }
26      CS2 {
27          s2_dm {
28             -> s3_dm;}
29          s3_dm {
30            -> s1;}
31      }
32      CS3 {
33        s5 {
34            entry / {
35             role3.fm3(r1Count);
36             System.out.println("Role1, sends fm3("+r1Count+") [After-loop]
    ");
37              System.out.println("Role1 in CS3 [LOOP Follow-Up]");}
38             -> s6_dm;
39        }
40        s6_dm {         }
41      }
42    }
43  }
44  class Role2
45  {
46    Integer a = -1;
47    Integer b = -1;
48    Integer r2Count = 0;
49    1 -- 0..1 Role1;
50    pooled   Sm_y {
51      iniState {
52        entry / { r2Count=0;
53      a=-1; b=-1;
54      }
55         -> s1_dmChoiceSt;
56      }
```

```
57      s1_dmChoiceSt {
58        fm0(int var_B)/{
59        System.out.println("Role2, receives fm0("+var_B+")  inside-loop");
60          a = var_B;}
61        -> CS2;
62        cim0(int var_F)/{
63           System.out.println("Role2, receives cim0() [AFTER LOOP]");
64          b = var_F;}
65          -> lNstate;
66      }
67      CS2 {
68          s2 {
69             -> FIN0s2;
70          }
71          FIN0s2 {
72              entry / {
73          r2Count++;
74          System.out.println("Role2, sends fm2("+r2Count+")  inside-loop");
75          role3.fm2(r2Count);
76              }
77              -> s3_dm;
78          }
79          s3_dm {
80            [a != b]  -> s1_dmChoiceSt;
81            [a == b]  -> CS3; }
82      }
83      lNstate {
84        [a != b]   -> s1_dmChoiceSt;
85        [a == b]   -> CS3;
86      }
87      CS3 {
88        s5_dm {
89           entry / { System.out.println("Role2 in CS3 [LOOP Follow-Up]"); }
90             -> s6_dm;
91        }
92        s6_dm {        }
```

```
93      }
94    }
95  }
96  class Role3
97  {
98    1 -- 0..1 Role2;
99    1 -- 0..1 Role1;
100   Integer a = -1;
101   Integer b = -1;
102   Integer r3Count = -1;
103   pooled Sm_z{
104     iniState {
105       entry / { r3Count=0;
106     a=-1; b=-1;
107     }
108        -> s1_dmChoiceSth;
109     }
110     s1_dmChoiceSth {
111       fm3(int var_F)/{
112          System.out.println("Role3, receives fm3()!");
113         b = var_F;}
114          -> lNstate;
115       fm2(int var_B)/{
116            System.out.println("Role3, receives fm2("+var_B+") inside-loop
      ");
117          a = var_B;}
118          -> CS2;
119       }
120     CS2 {
121         s2_dm {
122            -> PREL0s3;
123         }
124         PREL0s3 {
125            -> s3;
126         }
127         s3 {
```

```
128             [a != b] -> s1_dmChoiceSth;
129             [a == b]/ {
130              System.out.println("Role3 Process fm3()");
131             }  -> CS3;
132       }
133     }
134     lNstate {
135       [a != b] -> s1_dmChoiceSth;
136       [a == b]  -> CS3;
137     }
138     CS3 {
139       s5_dm {
140           entry / { System.out.println("Role3 in CS3 [LOOP Follow-Up]");
      }
141            -> PREL0s6;}
142       PREL0s6 {
143            -> s6;
144       }
145       s6 {   }
146     }
147   }
148 }
149 class Test {
150   public static void main(String[] args) {
151        Role3 c = new Role3();
152       Role2 b = new Role2(c);
153       Role1 a = new Role1(b,c);
154   }
155 }
```