

Specification of design and verification of service-oriented systems

**Robert Marzeta
s040176**

November 2007



Informatics and Mathematical Modelling

Technical University of Denmark
Informatics and Mathematical Modeling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MSc:2007-103

Preface

This report is a 30-ECTS points Master thesis prepared at Informatics and Mathematical Modeling (IMM) department at the Technical University of Denmark (DTU) from June to November 2007. The internal supervisor is Hanne Riis Nielsen (IMM, DTU).

Most of the gratitude goes to my supervisor, Hanne, not only for showing me what research really is and being a great source of advice and hints, but mostly for an outstanding skill and patience of telling me where to stop exploration in order to investigate more important issues. Many acknowledgments are also expressed towards the CPN Tools team at Aarhus University for kindly providing the license of use of the valuable CPN Tools application. I would also like to express my appreciation towards, again, the University of Aarhus, as well as the University of Uppsala for creating such a great verification tool like Uppaal.

Copenhagen, November 19, 2007

Robert Marzeta, s040176

Abstract

As a next step of software development evolution, Service Oriented Architecture is already a widely accepted standard to modern software systems. However, it seems that there are not enough tools and techniques that could allow efficient and precise modeling and verification of SOA. This thesis is an attempt to analyze and address some of those issues.

Firstly, some of SOA's mechanisms are described in the example of a hypothetical test scenario to show how the new approach can be effectively pursued to solve real problems. A number of known technologies and standards are reviewed (UML, Petri Nets, MDA) with respect to SOA applicability. Next, SOA's verification possibilities are assessed followed by the extraction of scenario specific properties.

Secondly, using an application CPN Tools, two design methodologies are followed to create a model realizing the test scenario. Results are compared in terms of clarity, scalability and complexity, which also influences verification suitability. Next, in order to complete partial verification provided by CPN Tools, Uppaal application is introduced. After showing two simple usage examples of modeling components and protocols, both Uppaal's strengths and weaknesses (towards SOA) are described.

Furthermore, an experimental algorithm is presented to transform a CPN Tools Petri net(s) to a state machine(s) compatible with Uppaal. The mapping takes under consideration various Petri net structures and produces significantly extensive models (more than 50 templates and 400 locations). Lastly, algorithm constraints are presented together with a discussion on other possible transformations.

| | |
|--|----|
| IMM-MSc: | 2 |
| Preface | 3 |
| Abstract | 4 |
| List of figures | 6 |
| List of tables | 8 |
| 1. Introduction | 9 |
| 2. Test scenario | 12 |
| 3. SOA | 14 |
| 3.1. Architecture | 15 |
| 3.2. Services | 17 |
| 3.3. Web services | 19 |
| 3.3.1. Message exchange | 20 |
| 3.3.2. Description | 20 |
| 3.3.3. Discovery | 21 |
| 3.3.4. Composition | 22 |
| 3.3.5. Transactions | 24 |
| 4. Modeling | 26 |
| 4.1. Model Driven Architecture | 27 |
| 4.2. UML | 28 |
| 4.2.1. Use case diagrams | 29 |
| 4.2.2. Activity diagrams | 30 |
| 4.2.3. Sequence diagrams | 31 |
| 4.2.4. Communication diagrams | 34 |
| 4.2.5. Component diagrams | 34 |
| 4.2.6. Class diagrams | 36 |
| 4.2.7. State machine diagrams | 37 |
| 4.3. Petri nets | 39 |
| 4.4. Modeling tools | 41 |
| 4.5. Modeling conclusions | 41 |
| 5. Verification | 43 |
| 5.1. Model checking | 45 |
| 5.2. Computation Tree Logic | 47 |
| 5.3. Verification tools | 48 |
| 5.4. Verification properties | 49 |
| 6. CPN Tools | 51 |
| 6.1. Goal sequences | 54 |
| 6.1.1. Methodology | 55 |
| 6.1.2. Implicit scenario detection | 59 |
| 6.1.3. Petri net realization | 60 |
| 6.1.4. Verification | 70 |
| 6.1.5. GSM conclusions | 78 |
| 6.2. Top-down abstraction refining | 79 |
| 6.2.1. Methodology | 79 |
| 6.2.2. Verification | 86 |
| 6.2.3. TAR conclusions | 88 |
| 6.3. CPN Tools conclusions | 89 |

| | | |
|--------|--|-----|
| 7. | Uppaal..... | 93 |
| 7.1. | Component analysis..... | 94 |
| 7.2. | Protocol analysis..... | 96 |
| 7.3. | Uppaal conclusions..... | 97 |
| 8. | Model transformation..... | 99 |
| 8.1. | Element transformation..... | 101 |
| 8.1.1. | Places/Locations..... | 102 |
| 8.1.2. | Transitions..... | 103 |
| 8.1.3. | Sub pages..... | 107 |
| 8.1.4. | Fusion places..... | 108 |
| 8.1.5. | Data passing..... | 109 |
| 8.1.6. | Complete concurrency..... | 110 |
| 8.1.7. | Declarations..... | 111 |
| 8.1.8. | Verification expressions..... | 112 |
| 8.2. | Algorithm..... | 113 |
| 8.3. | User guide..... | 119 |
| 8.4. | Limitations..... | 120 |
| 8.5. | Application details..... | 120 |
| 8.5.1. | Code structure..... | 121 |
| 8.5.2. | Uppaal's data format..... | 121 |
| 8.6. | Methodologies..... | 122 |
| 8.6.1. | 5-step collaboration goal sequence..... | 122 |
| 8.6.2. | Top-down abstraction refining..... | 127 |
| 8.7. | Transformation conclusions..... | 131 |
| 9. | Future work..... | 133 |
| 10. | Conclusions..... | 134 |
| 11. | Appendix A – Glossary..... | 136 |
| 12. | Appendix B – CD contents..... | 137 |
| 13. | Appendix C – Use cases..... | 137 |
| 14. | Appendix D – Verification properties..... | 141 |
| | Goal sequences transformation..... | 141 |
| | Top-down abstraction refining..... | 145 |
| 15. | Appendix F – State space reports..... | 148 |
| | Goal sequence methodology (basic version)..... | 148 |
| | Abstraction refining (basic version)..... | 153 |
| 16. | References..... | 159 |

List of figures

| | | |
|------------|---|----|
| Figure 3-1 | Composition of services from blocks..... | 15 |
| Figure 3-2 | Example SOA framework..... | 16 |
| Figure 3-3 | Web service stack and key dimensions [DMWS]..... | 19 |
| Figure 4-1 | Use case diagram of system behavior..... | 30 |
| Figure 4-2 | Request Service - Activity Diagram..... | 31 |
| Figure 4-3 | Synchronous and asynchronous communication pattern..... | 32 |
| Figure 4-4 | Request Service Sequence diagram..... | 33 |
| Figure 4-5 | Request Service Communication Diagram..... | 34 |
| Figure 4-6 | System - Component Diagram..... | 36 |

| | |
|---|----|
| Figure 4-7 Class diagram of the system | 37 |
| Figure 4-8 State machine of a vehicle component | 38 |
| Figure 4-9 State machine of a protocol for port G2C from garage service's perspective..... | 38 |
| Figure 4-10 State machine of a protocol for port C2G from Central service's perspective.... | 39 |
| Figure 4-11 Graphical classification of Petri Nets [WIKI] | 40 |
| Figure 5-1 Context-free grammar of CTL logic | 47 |
| Figure 5-2 Minimal set of operators for CTL..... | 48 |
| Figure 6-6-1 Initialization of ASK_CTL library | 53 |
| Figure 6-2 Request services process composite collaboration with taxi service..... | 56 |
| Figure 6-3 Goal sequence without a taxi service | 57 |
| Figure 6-4 Goal sequence with a taxi service..... | 58 |
| Figure 6-5 Detailed interactions for the sub-collaborations | 59 |
| Figure 6-6 Mapping of goal sequence elements to HCPN [FCGC] | 61 |
| Figure 6-7 Declarations together with page overview | 61 |
| Figure 6-8 Collaboration goal sequence as Petri net..... | 62 |
| Figure 6-9 Dependency mechanism..... | 64 |
| Figure 6-10 Communication through fusion places..... | 64 |
| Figure 6-11 Authorize Payment interaction | 65 |
| Figure 6-12 Cancel Payment interaction | 66 |
| Figure 6-13 Bank's logic for authorize payment and cancel payment interactions..... | 66 |
| Figure 6-14 Request towing interaction | 67 |
| Figure 6-15 Towing agency logic | 68 |
| Figure 6-16 Request garage interaction..... | 68 |
| Figure 6-17 Request garage interaction..... | 69 |
| Figure 6-18 Garage logic..... | 69 |
| Figure 6-19 Cancel Garage interaction | 70 |
| Figure 6-20 Confirm booking interaction..... | 70 |
| Figure 6-21 Function to search all states to find dead markings | 71 |
| Figure 6-22 Predefined query to find dead markings..... | 72 |
| Figure 6-23 Query to find and display a path between nodes | 72 |
| Figure 6-24 Function displaying a precise description of a first dead marking | 72 |
| Figure 6-25 Description of a marking node number 1086 | 73 |
| Figure 6-26 Verification whether all dead states have tokens in one place | 74 |
| Figure 6-27 Streaming query results to a file | 74 |
| Figure 6-28 Error message while searching state space of a complex model | 75 |
| Figure 6-29 Correct result for a query searching for a specific marking | 75 |
| Figure 6-30 Query to check whether garage can be confirmed even after renting and towing has been rejected in GSM..... | 76 |
| Figure 6-31 Reachability verification | 76 |
| Figure 6-32 Simple Petri net to check liveness formulas | 77 |
| Figure 6-33 Simple test of ALONG formula..... | 77 |
| Figure 6-34 Simple test of EV formula | 78 |
| Figure 6-35 Highest abstraction overview of top-down abstraction refining methodology ... | 80 |
| Figure 6-36 Request Towing collaboration with its interfaces..... | 81 |
| Figure 6-37 Vehicle logic | 81 |
| Figure 6-38 A view at Central's main logic | 82 |
| Figure 6-39 Interfaces and their transitions related to direct interaction with Vehicle..... | 83 |

| | |
|---|-----|
| Figure 6-40 Networks hierarchy | 83 |
| Figure 6-41 Process logic of “reserve repairs” sub-net..... | 84 |
| Figure 6-42 Details of “reserve garage“ sub-page..... | 85 |
| Figure 6-43 Details of garage service | 86 |
| Figure 6-44 Dead markings in top-down abstraction refining methodology | 86 |
| Figure 6-45 Verification query whether final markings have tokens in expected final places..... | 87 |
| Figure 6-46 Reachability test whether booking can be confirmed after towing and both renting car and taxi have been rejected | 87 |
| Figure 6-47 Verification query to check whether for all occurrences of both renting or taxi approval, deposit payment will not be canceled | 88 |
| Figure 6-48 Scenario Elicitation, Scenario Creation and Structuring [PAVT] | 90 |
| Figure 6-49 Ambiguous errors while running simulation | 91 |
| Figure 7-1 Syntax of expressions in BNF [TOU]..... | 93 |
| Figure 7-2 State machine of Vehicle component behavior | 95 |
| Figure 7-3. State machine of a interaction protocol of reserving garage and towing | 96 |
| Figure 7-4 Tree-like structure of related elements..... | 97 |
| Figure 8-1 Petri net of a beginning of 4 concurrent behaviors | 105 |
| Figure 8-2 State machine of a beginning of 4 concurrent behaviors | 105 |
| Figure 8-3 First and main template of a transformed model | 124 |
| Figure 8-4 Concurrent processes that begin at transition “fork” | 125 |
| Figure 8-5 Extended version of towing reservation (reqTow page)..... | 126 |
| Figure 8-6 Highest overview abstraction level after transformation of TAR methodology .. | 128 |
| Figure 8-7 Central’s main logic | 129 |
| Figure 8-8 “Reserve Repair” process logic after transforming TARM model..... | 130 |
| Figure 8-9 “Reserve Car” process logic after transforming TARM model..... | 131 |
| Figure 14-1 Results of a query to find maximum number of reserved garage..... | 145 |

List of tables

| | |
|--|-----|
| Table 1. Sub-role sequences for the Central sub-role (without taxi) | 60 |
| Table 2 Comparison between CPN Tools and Uppaal..... | 99 |
| Table 3 Terminology differences between CPN Tools’ Petri net and Uppaal’s net..... | 102 |
| Table 4 Transformation of a single place | 102 |
| Table 5 Transformation of 1-to-1 transition | 103 |
| Table 6 Transformation of a form transition | 104 |
| Table 7 Transformation of join transition | 106 |
| Table 8 Transformation of many-to-many transitions | 107 |
| Table 9 Transformation of a sub-page | 108 |
| Table 10 Transformation of fusion places..... | 109 |
| Table 11 Transformation of data passing | 110 |
| Table 12 Transformation of a supported type of concurrency | 111 |
| Table 13 Comparison between syntax | 112 |
| Table 14 Comparison between supported logical formulas | 113 |
| Table 15 Possible conversion between string and int data type | 118 |
| Table 16 List of inscription modifications in GSM transformation..... | 122 |
| Table 17 Design specific tests | 144 |

1. Introduction

Globalization creates new demands for a greater flexibility in distributed (software) systems. The main motivations that drive this trend are increasing computerization and business to business (B2B) integration. Systems become more and more relied, not only by providing some time saving functionality, but mostly by protecting customers' assets. However, while systems are upgraded a flawed additional functionality may in fact decrease system's reliability and so – the overall value of a service. The irritation is even bigger when it is not a fancy, but basic and crucial functionality that fails after the 'improvement'. Every bug of this kind undermines the trust in technology and questions vendor's brand.

SOA (System Oriented Architecture) has already become a widely accepted standard to modern software systems. Its flexibility in reusing and binding services together allows maximizing existing business assets as well as creating new components as a solid foundation for next generations. It proposes many techniques to build a clear, flexible and efficient system that can be upgraded and expanded in an evolutionary way. It introduces a notion of services (components) as primary, autonomous building blocks. Services allow cost-effective maximization of IT assets by modularization, reuse and standardization both existing as well as new components. By separating technical details of service implementation from business logic, both of them can evolve at their own pace – technology taking advantage of new developments and business logic responding to business needs.

Despite all the mechanisms that SOA supports, as systems grow more and more complex, their vulnerability to errors, bugs and unpredictable behaviors increases. This is caused mostly by the very nature of service interactions that involve collaboration of many participating and/or dependable (on each other) sub-services and components, in an either interleaving or interrupting scheme. Even subtle errors overlooked in a low-level logic may have a crushing effect on the whole distributed system.

In order to keep up with the increasing complexity, new techniques need to be developed to aid human comprehension of such complex systems. Moreover, they need to be organized into consistent and pragmatic development methodologies to ensure that the system is analyzed and verified as a complete entity. It is known that testing each part of the system separately, does not prove its overall correctness. System verification should preferably be integrated early in the design stage to reduce correction costs. Additionally, tools need to systematically check the design automatically which increases reliability and decreases operation time. Finally, all techniques should be close to industry standards to be usable not only for a computer scientist but also for an 'average engineer' without any desire for complex mathematics.

Thus, the main goals of this thesis are:

- **to analyze current trends and technologies**

Many different SOA mechanisms and practices are well suited to inspire a highly flexible design that can match integration and rapid improvement of future systems. They need to be

investigated in order to propose suitable verification techniques as well as necessary project assumptions.

- **to investigate verification possibilities**

Certain safety properties need to be validated in order to verify that a system ‘does’ what it should. Since manual verification of systems is as error prone as the tested system itself, automation techniques and tools need to be proposed. Not only do they need to be intuitive to use, but also support many of the already accepted SOA techniques. They should also work on a model that contains complete functionality to provide end-to-end view of every service. Those tools may be either:

- created from scratch – that allows achieving desired results and behavior for the cost of development and testing time
- extended from already existing tool – that saves a lot of development time, provided that the tool has an extension mechanisms (not very common)
- integrated from other specialized tools – there is a huge number of efficiently working tools so the main task is to translate the data between them

- **to propose pragmatic development methodology**

An important goal is to give a designer a consistent chain of developing actions from requirements to low-level details in the spirit of a model driven design. A methodology should allow specifying all aspects of a system in systematic and incremental steps with a possibility of verifying them not only individually but also together. While there are many detailed models that describe details of behavior, high level models are very rare. The skill of abstraction is said to come with practice and experience but we believe that a good methodology should speed up the process of efficiently designing correct systems. Another goal is to fill a gap between theoretical, perfection-seeking academic and practical, engineering approaches. Since engineers are naturally skeptical about using formal reasoning, they should be able to take advantage of it even without having to master it. Even though real size systems cannot be guaranteed absolute correctness, a proper methodology is believed to significantly increase their reliability.

Because of the amount of standards and techniques, that the final product of this thesis relies on, a number of issues had to be described in the following chapters. Every chapter refers to the test case in terms of examples for described techniques.

Chapter 2 presents a test scenario based on a Sensoria automotive problem.

Chapter 3 uses a test scenario to describe the idea of essential elements of SOA including its most known implementation – Web services.

Chapter 4 analyses popular modeling techniques such as various UML2 [UML 2.0] diagrams, Petri nets and TLA with respect to applicability in SOA design.

Chapter 5 evaluates verification possibilities for modeling mechanisms discussed in previous chapter as well as introduces model checking and CTL logic concepts. Thereafter, verification properties of a SOA-based system are analyzed.

Chapter 6 introduces CPN Tools as a tool suitable for designing and verifying a SOA system. Different design methodologies are followed by their verification possibilities analysis.

Chapter 7 analyses design and verification possibilities of Uppaal with respect to models of a component and an interaction protocol. Afterwards, an attempt is made to list and address some of Uppaal's limitations.

Chapter 8 presents an algorithm to automatically transform a Petri net created by CPN Tools to a state machine compatible with Uppaal for additional verification.

Chapters 9 and 10 highlight some of algorithm's future development together with a discussion of achieved and possible capabilities.

2. Test scenario

Test scenario is based on a real case scenario from the European project: Software Engineering for Service-Oriented Overlay Computers [SENS]. It presents a realistic test case, not only for academic analysis but also for intuitive understanding that can inspire future improvements and extensions.

Automotive industry

Automotive industry is a rapidly growing market for software services. Almost 80% of innovation cost for a new vehicle goes to software systems [SESO]. Every car is equipped with dozens of sensors and with almost 70 electronic control units and they all require their specific software to operate.

Our scenario [SESO] describes an automatic procedure for an engine failure. It is already a realistic solution in higher class vehicles and should be available in medium class in near future.

Low Oil Level Scenario

Scenario outline:

When a diagnostic system in the car discovers a failure that prevents further driving, based on the car location, a Central “On-the-road” system performs an automatic look-up of:

- garages that agrees to fix the car,
- towing services that can deliver the broken car to the garage,
- replacement car (either rented or taxi) for the driver to get to his/her destination

Scenario details:

Every car is equipped with sensors that monitor the car’s condition and initiate a repair procedure when a serious problem appears. Every car is additionally has a GPS to indicate the car’s location, a reliable communication device based on GSM network, LCD screen and credit card reader. Owner of the service “Central” maintains the whole system.

Let us suppose a serious fault, such as low pressure of oil, is detected in one of the cylinders. In order to prevent further damage a driver cannot continue a trip and is asked to provide a credit card to debit deposit of money to cover possible supportive services. The driver can also specify personal preferences like preferable garage company, replacement car model or no desire for taxi services. Diagnostic data, vehicle’s position and card information are thereafter sent, by a communication device, to Central that is responsible for providing services.

First step a Central does, after receiving a request, is contacting driver’s bank to authorize a deposit payment. If the bank approves the payment, procedure continues; otherwise, the driver receives a service rejection message on his LCD screen.

In case of payment approval, Central matches available garages against cars position and driver's preferences to contact the most suitable garage, with the fault data, in order to make an appointment. If the garage does not approve the request, a next matching garage from the database is contacted. If no garage accepts the request, the driver is informed about that. If the garage approves the appointment, Central books a towing service (in a similar way) from car's position to the garage's location. If, for some reason, there are no towing services available, the garage appointment is cancelled and a new garage has to be booked, hoping that the following towing booking is successful. Only after a successful towing reservation a garage booking is confirmed and both companies contact information are provided to the driver.

Concurrently to garage/towing booking, a replacement car is requested to be delivered to the car's location. Since responsiveness to changes is a property to be tested in the design methodology, there is an extension to the scenario that allows the system to order a taxi if no replacement car is available. Either way, a replacement rented car, incoming taxi or service unavailability information is responded to the driver.

In the worst case, if neither of the services is available, previously reserved deposit is cancelled and the driver gets an apology and is supplied (possibly) with some useful advices/contacts.

The final payment for external services, together with possible insurance is handled by the Central and is not to be considered in the design. As a typical tradeoff between speed and efficiency, the acceptable communication timeouts should not take more than a few minutes. It has to be noted, that from the point that the driver provides the bank card information, the booking process is fully automated.

3. SOA

“SOA is a new way of looking at a very traditional, classic problem, so I don't think of it as brand new.”[ISDD]

This chapter describes SOA paradigm together with its most used techniques. It is necessary to show the underlying technology in order to make proper assumptions while designing and verifying a system.

Service Oriented Architecture (SOA) is a natural evolution of technologies like object-oriented design (OOD) or CORBA¹. For the last few years it has become a standard to meet market's demand for architectural design pattern for large distributed systems. SOA is based on service-orientation design principle with loosely-coupled services, also called components. Even though SOA framework is not a software development process that provides any guidance to designing or implementing a specific architecture, it can incorporate other tested building methodologies.

SOA's main features are:

- flexibility to bind services over physical (enterprise) boundaries with dynamic service binding and look-up
- standardized interfaces bridge incompatible end-point technologies making them accessible for more users
- reusability decreases costs of creating systems
- system maintenance is easier by being distributed close to services

The test scenario system is suitable for SOA for the following reasons:

- it is a new system that needs defining of interfaces and free standards
- building blocks (both external services and car manufacturer) are controlled by unrelated companies with probably various technologies and communication patterns
- it is going to be a long time investment, growing and changing with external services, as well as internal process logic
- new service providers can appear and compete for the customers using the already defined and open standards
- defined communication standards will allow concurrent work on different modules triggering a healthy competition
- with current prices of hardware and internet connection, many external services can afford maintaining their own service servers
- new services may emerge on the already implemented blocks (like a garage website to make an appointment for home users)

¹ www.corba.org

3.1. Architecture

“Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.”[IEEE]

Service Component Architecture (SCA) is a set of specification that describes SOA components with respect to their:

- number
- behavior – general or specific description of functionality
- relationships:
 - communication – both synchronous and asynchronous
 - hierarchy – to allow abstraction in the design process
 - inheritance – to allow reuse
- rules and constraints under which their function can be:
 - broad (high-level)
 - specific (task-level)

Since SOA is based on services, the architecture should support many different views on the system which might help to design them in a structured way.

The main idea of SOA is to create services by combining smaller, loosely coupled blocks (possibly) belonging to different owners. Figure 3-1 descriptively presents a composition of such a system.

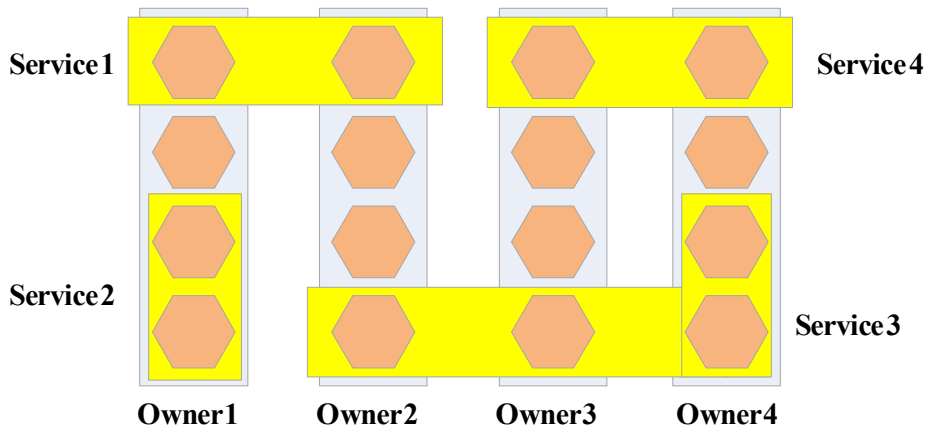


Figure 3-1 Composition of services from blocks

Participants of services can be categorized in three groups:

- provider – owns service implementation and allows others to access it
- consumer (client) – uses an already provided services
- registry (broker) – stores descriptions of web services and allows clients to search for them; operations that a registry should support are:
 - advertising (publication) – that adds a new service description to the database
 - search (lookup) – that allows searching the registry for a certain type of service

- binding – is to locate, contact, and invoke the service based on the binding information in the service description

In our test scenario, a driver is a consumer requesting a bundle of repair services. Central is a provider that maintains servers with running code but also a consumer that requests services from external parties like: bank, garage, towing and renting agencies. In the given scenario, the external parties play only provider roles.

One can easily imagine many configurations and scenarios that could be realized on the framework shown in Figure 3-2. A standardized communication and interface specification is represented by colored triangles. All parties, managed by separate companies, can not only fulfill a common goal but also provide services on their own. Below we can see that, apart from common passenger transportation, a taxi company also offers shopping services where goods are delivered to a customer on demand. Similarly, a towing agency offers various transport requests, a renting agency allows buying cars and a bank provides an account management. All of those services that are not a part of a test scenario can coexist in the same flexible framework allowing many possible configurations.

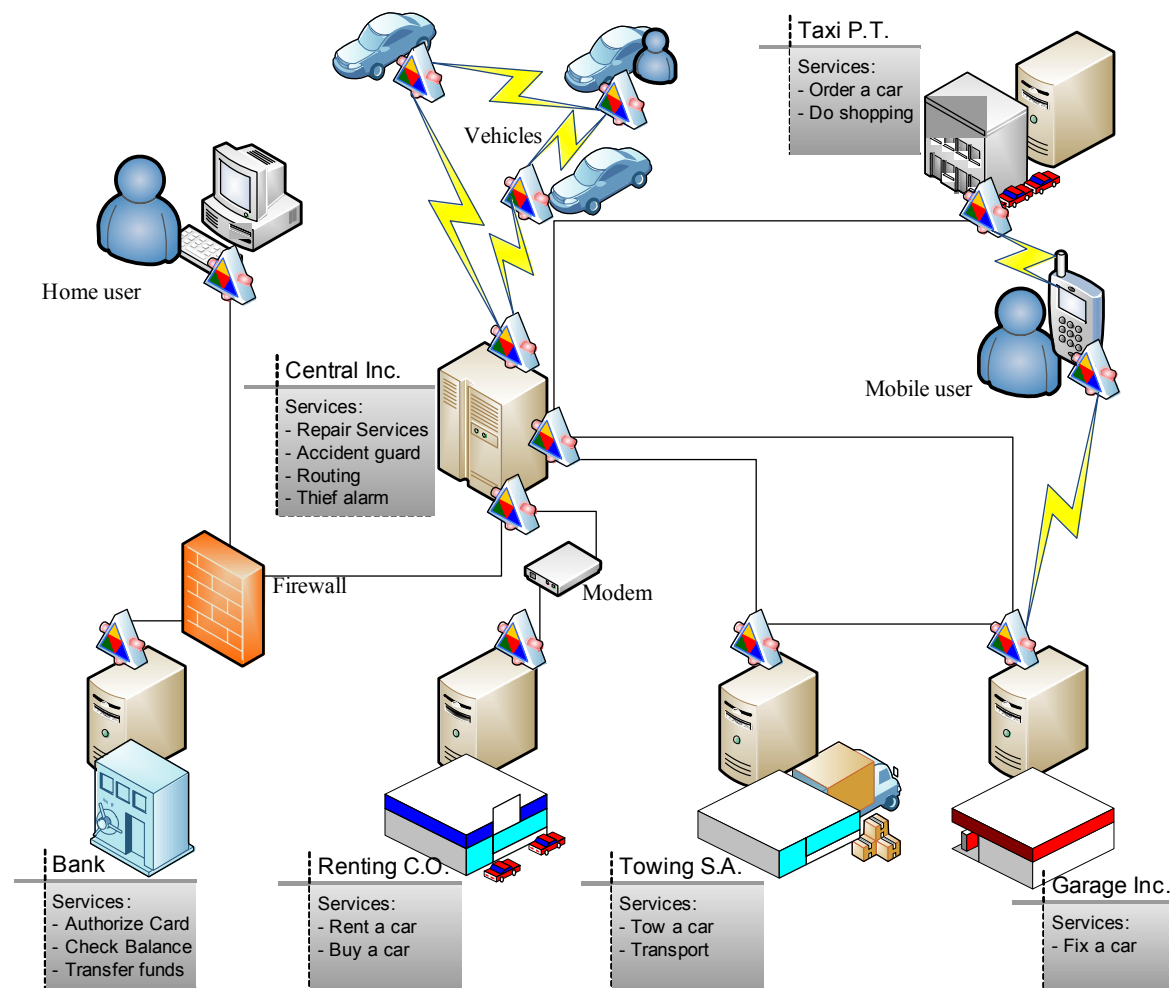


Figure 3-2 Example SOA framework

3.2. Services

“(…) a component is modeled throughout the development life cycle (…)” – UML 2.0 Superstructure Specification

Services (also called components) are the essential part of SOA. Some refer to services as a natural development of software architecture – an evolution after procedural and object-programming.

From a technical point of view, a service should have a well defined interface that separates its description from implementation. None of the executable technologies, like J2EE, .NET Framework, or CORBA objects, are visible outside a component. However, a service can be represented either by a black box hiding internal details or by a white box displaying all or some underlying logic. Apart from how much is displayed, a service consists of:

- data – that represent state or configuration options
- process – that describes a partially ordered activities that define service’s behavior

There are two kinds of interfaces related to a service:

- required – that defines other services the service requires to fulfill its tasks
- provided – that describes functionality the service offers

Services should fulfill following properties:

- reusability – can be used many times in different contexts
- isolated design – it is designed as an autonomous mechanism that significantly decreases development cost by promoting buying components from third parties and out-sourcing. Service developers compete with each other to deliver even better quality of code.
- compositionality – services can be grouped together in hierarchies
- encapsulation – only interfaces are visible to consumers
- isolated ‘life’ – can be upgraded without consideration of the rest of the system

Thinking in terms of services requires a change in terms of design methodology. True power of services is measured in a long run and as such they should be designed with integration and reusability goals in mind. In those terms, an immediate effect achieved from developing a service is not as important as a long-term benefit. Services should also encourage integrating many components in a pursuit of a common goal. Ideally, a service-based system will evolve in an organic way. New and better component should replace old versions adding new functionality while maintaining compatibility with old mechanism. That kind of system would be more reliable by not having to shutdown or reset while upgrading of any of its parts.

Despite all their advantages, services have drawback, such as:

- design complexity – long-term planning is not easy since it is hard to find and design reusable roles for a service
- expensiveness – requiring existing systems to be adapted to new standards; in some cases, a cost of rewriting specialized applications with a complex access (e.g. through file system or batch data input or output) can exceed the gain

In terms of compositionality services can be divided into:

- atomic – do not rely on other services and their behavior is usually executing data queries and updates or simple transactions
- composite – involve cooperation of two or more services

Both atomic and composite services consist of:

- implementation – an executable code that performs a functionality
- description – contains only information about a web service:
 - syntactic – technical details about accessing technique, data types, input/output of the operations, binding information
 - semantic – additional description of the actual functionality, such as quality of service, policies, taxonomy

Services are closely related to business units and as such should provide measurable profits. They should also describe recognizable business accessing policies and quality assurance like:

- access control – authorizations to use the service
- availability – should estimate probability of outages
- taxonomy – define cost of using a service
- reliability – security, fault control, maximum workload
- security – privacy, integrity regulations, authentication
- performance – throughput, response time
- maintenance cost

In our test case, we can define seven component types with (conceptually) different services they offer. Services that are not required for our scenario are marked with italic.

- Vehicle
 - Display Information – shows messages from the Central to a driver
 - *Provide Location* – reports its current position (theft protection)
 - *Immobilize Engine* – allows remote deactivation of engine (theft protection)
- Central
 - Provide Services – organizes garage, towing, renting or taxi in case of a serious failure
 - *Accident Monitoring* – checks that a driver is alive after an accident happens, if not – notifies a rescue team
- Bank
 - Authorize Card – approves and debits a deposit money for future services
 - Cancel Payment – cancels previously booked deposit
 - *Transfer Funds* – allows transferring money from customer's account
- Garage
 - Book Appointment – reserves a time and place for a car in order to fix it
 - *Sell Parts* – offers a shop functionality for car components
- Towing agency
 - Book Towing – delivers a car to a garage
- Renting agency
 - Book Renting – delivers a replacement car to a driver to cover repair time
 - *Buy cars* – offers a shop functionality to sell cars

- Taxi agency
 - Book Taxi – delivers a taxi to bring an unfortunate driver to its destination
 - *Do shopping* – buys and delivers products on demand

3.3. Web services

Web services [WS] provide a standard of integrating web-based applications introduced by World Wide Web Consortium (W3C). It is the most popular practical example of SOA paradigm. They are defined, described (interface, bindings) and discovered with a use of XML.

The three XML standards, that are key technologies for Web services, are:

- WSDL (Web Services Description Language) – precisely describes a service
- UDDI (Universal Description, Discovery and Integration) – is a registry that allows storing and advertising services descriptions
- SOAP (Simple Object Access Protocol) – that defines a communication protocol

There are five layers of a Web service stack that are descriptively shown in Figure 3-3

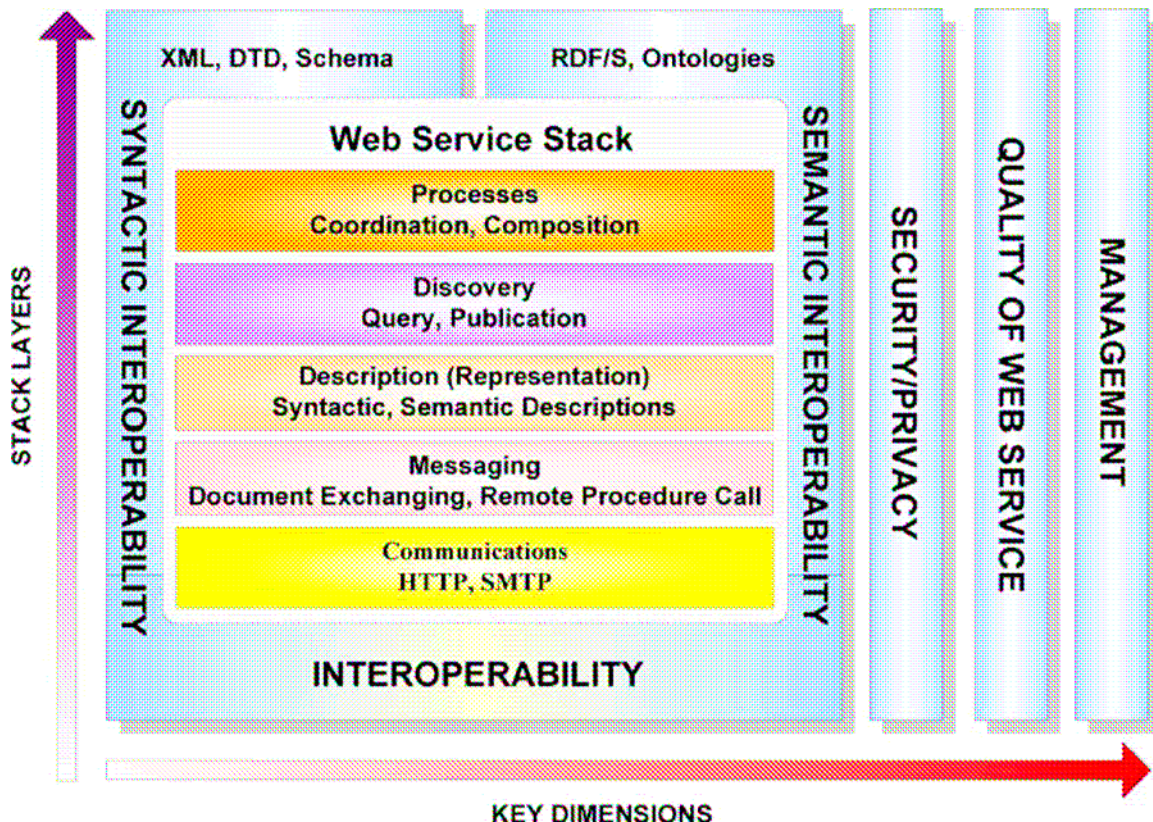


Figure 3-3 Web service stack and key dimensions [DMWS]

3.3.1. Message exchange

SOAP (Simple Object Access Protocol) [SOAP] describes a framework of communication between Web services. It is based on XML data that can be delivered by messaging (HTTP, SMTP) or RPC.

Additionally, SOAP has many extensions that allow specifying transport:

- WS-Routing[WS-ROUT] – can define network nodes a message should visit on the way to destination
- WS-Addressing[WS-ADD] – allows specifying the exact end point a message should be delivered to
- WS-Reliable Messaging[WS-REL] – ensures a reliable message delivery (guaranteed delivery, duplicate elimination and message ordering)

Because of its textual and XML based form SOAP is rather robust in representing data. This can result in communicating large messages that can influence performance of the system. This problem can be addressed by either attaching (possibly compressed) binary data to SOAP envelope or instead of all data – send only its reference.

3.3.2. Description

WSDL (Web Services Description Language) [WSDL] is a standard for Web service syntactic description. Semantic description is not supported. WSDL describes both how to access a Web service and where it is located. In order to achieve interoperability and platform neutrality, WSDL supports XML Schema Definition (XSD) as a canonical type system.

WSDL contains two types of descriptions:

- abstract
 - Type – defines data types with XSD
 - Message – identifies the abstract definition of the transferred data
 - PortType – abstract operations provided by a Web service with transmission primitives like:
 - one-way,
 - request–response,
 - solicit–response,
 - notification
- concrete – information about binding to a concrete service endpoint:
 - communication protocols (SOAP, HTTP)
 - data format specifications
 - network addresses
 - port – single address for binding a service endpoint
 - service – set of related ports

3.3.3. Discovery

UDDI (Universal Description, Discovery and Integration) [UDDI] is a standardized framework that allows discovery and advertisement of Web services. A standardized repository increases the flexibility of a system not only by containing descriptions of all service but especially for their run-time updating capabilities. There is a defined API based on SOAP for searching queries, publishing new services and updating existing ones.

In general, registries can be classified in terms of:

- who owns and maintains them:
 - public – with standardized policies and control distributed among many participants
 - private – usually owned by one unit that adjusts policies to match a specific functionality
- service selection which may either be:
 - static – service end-points do not change
 - dynamic – service location may be found on demand:
 - with reference – forwards a request to service
 - with lookup – all services are explored with every request

Lookup

Web services can be searched by classification they belong to or policies they support (e.g. security). An important parameter in the searching mechanism is a discovery time [WSCA] which should also take fault-tolerance and load-balancing under consideration.

There is still a challenge of how to describe Web services so that they are both human comprehensible and machine interpretable. For that reason research is made in the area of semantic UDDI [ASWS].

Publishing

UDDI registry allows advertising new Web services as well as updating the existing ones for both service information and security policy.

The information about a web service that UDDI stores can be divided into:

- white pages – store information about provider's company and their Web services
- yellow pages – contain a classification of a Web service (type)
- green pages – specify technical details about accessing a Web service

In our case a private registry is owned by the Central which maintains the data, making sure it is updated periodically. External services have obviously access to their information, but the Central has some private records regarding each service which contains company confidential data such as: customer's opinions and complaints, quality and availability reports, etc. Since the advertisement of a UDDI registry is not a requirement in our scenario, it is not mentioned further.

3.3.4. Composition

Composite services are described, registered and secured as atomic services, so from the interface point of view are indistinguishable from them. Running a composite service may require not only access and configuration interfaces, but also a monitoring input to trace the progress of a business process.

From an implementation point of view, a business process can be represented:

- declaratively – techniques like BPEL or UML activity diagrams allow intuitive description of the logic
- programmatically – suitable for more specialized processing can be performed in JAVA, C++, C#, Perl, Python
- other formalisms – such as process algebras, Petri nets, etc

From a predicted running time we can divide them into:

- short running – when the whole process ends in a relatively short time
- long running – when fulfillment of the service may take days or even weeks

Workflow of complex services can be described by:

- orchestration – executable business process that describes coordinated Web services from the perspective of one party
- choreography – describes interactions between multiple Web services with none of them having a full control

Even though systems can be designed in many ways, our test scenario seems appropriate for an orchestration scheme. Firstly, because of the urgency of an already uncomfortable (for the driver) situation when a customer awaits for a resolution, it is a short-time process. Therefore a turn-around time should be measured in minutes. Secondly it is an example of an orchestration with a central coordinator that maintains and monitors both business process and UDDI registry providing (if necessary) also human support when services fail. In addition, the Central could also have a ranking of service providers, preferably influenced by the drivers' opinions that would prioritize choosing best services.

Nevertheless, this type of centrally organized architecture is prone to a typical client-server problem where the central node is a bottle neck of the whole system. The system is vulnerable towards Central failures, outages, or DoS attacks. Luckily, the service oriented nature of the Central allows having one or more backup systems. They could be using various internet connections and be registered, together with the primary box, in several UDDI registries.

Instead of the coordinator orchestrating a garage and towing to be reserved, those actions could also be handled with choreography. In such a case a vehicle system would have to alone perform some of the Central's actions, such as booking garage and renting agency. Garage services could also look for nearby towing by themselves. In the optimistic estimations, such a solution could be even free of charge. However, the most serious possible shortcomings are:

- increased costs for the network communication between the vehicle and external services
- risk of ‘unhealthy’ competition between external services could influence the choices of services they choose to contact
- other typical problems of peer-to-peer systems:
 - difficulties in assuring a quality of service
 - greater complexity of behavior, caused by simultaneous actions, may create unexpected conflicts that need to be resolved by individual nodes
 - difficulties in enforcing any improvements, patches or security policies

Because of all the pros and cons, the orchestration approach is pursued in the later examples.

WS-BPEL

Business Process Execution Language (BPEL) [BPEL] is a widely accepted technology for SOA already integrated in many Web service mechanisms. It is one of the most popular techniques to orchestrate Web services and allows automatic configuration. Choice of services may be driven by many parameters such as cost, failure rate, etc. BPEL allows orchestrating services and structuring activities in hierarchies. It is a business process modeling orchestration language, abstract enough to allow non IT persons understand it. BPEL can also be executed and supports many practical mechanisms such as:

- loops
- concurrency
- data passing
- fault handler,
- compensation
- session tags (correlation sets)

It includes information such as when to wait for messages, when to send messages or when to compensate for failed transactions.

However, BPEL’s textual representation is meant to be understood by machines rather than people. Because of readability issues, it does not fit very well with a description of whole systems, but only partial functionalities. Finally, it does not have any underlying, formal logic basis, but there are attempts to specify BPEL mechanisms with logic suitable for automated verifiers [COWS]

Correlation

Multiple message-based communications between nodes have to be extended with correlation that allows unifying incoming replies with their pending requests. To uniquely identify interacting parties every message needs to be equipped with receiver’s and sender’s network addresses together with end point services, to deliver messages to correct receivers, and timeout indicator. Correlation prevents, for example, an erroneous influence of a towing reply that answers after its timeout has exceeded. Properly implementing timeout monitor in incoming ports should correlate messages behind schedule with expected time and disregard them.

Since our scenario communicates through insecure medium (Internet) and the confidential data character (banking cards, locations, and telephone numbers), all the SOAP messages need to be encrypt (WS-Security). To correct problems with packet duplication,

communication errors or copy-paste attacks, all messages have to be signed (WS-Security), correlated (WS-BPEL) and distributed in a reliable way (WS-Reliability). Correlation set should contain not only addresses of a sender and a receiver, but also precise description of endpoint ports together with a timestamp of the message.

Also WS-Addressing might be used to identify Web services' end points. Due to the complex nature of business process, service orchestration has to be executed in a programmed logic. Authorization mechanism and key distribution is out of the scope of this thesis.

3.3.5. Transactions

As a consequence of a message based interaction by loosely coupled Web services, requested components cannot return values or exceptions, like in tightly coupled systems, allowing a requestor to proceed with its logic. Instead, they need to send an error message back that has to be correlated by the awaiting requestor to a proper request. The possible length of such a long-lasting interaction sometimes leads to using compensations that undo a previously performed operation.

Additionally, some business processes require that either all of the involved parties agree on an operation, or none of them. Those kinds of transactions can also be characterized by an acronym ACID that stands for:

- Atomicity – specifies that either all operations are performed or none of them
- Consistency – defines invariants that need to be respected by transactions
- Isolation – guards that process details are not visible from outside
- Durability – ensures that a successful operation remains and cannot be undone

Atomic actions is a solution for assuring ACID and are described using the example of a two-phase commit protocol. Firstly, the coordinator sends a query to all the parties and awaits their approval; secondly, after receiving all positive replies, the coordinator sends out the 'commit' message to process the query or, if not all of them were positive, a 'rollback' message to undo the previous actions. The biggest disadvantage of this technique is its blocking nature, as all resources are blocked until the coordinator decides to continue or abort. This flaw is especially serious when the coordinator fails. Another drawback is the poor scalability over large unreliable networks. A number of various transaction protocols are supported by WS-Transaction [WS-TRAN].

Another solution to protect transaction logic is through compensations. They are realized by marking a state to revert to, when not all conditions are met while fulfilling a transaction. It has to be noted that some operations, such as sending of an email, cannot be undone. Additionally, it is not defined what happens when the compensation mechanism fails.

Transactions can be protected either by atomic actions or compensations and a choice between the techniques depends on:

- number of involved parties
- maximum time of transaction
- distances between participants
- reliability of connection

- consequence (value) of the blocked resource

Neither atomic nor compensation based techniques can solve all the requirements. In some cases, however, a combination of the two with a coordinator protocol could. It is not clear which approach will become dominant in the future [USWS].

The case scenario involves two different approaches to transaction. Two-phase commit protocol is used with a garage that after confirming an appointment waits until a towing service is booked. Depending on the availability of a towing car, the garage is either confirmed or cancelled but the appointment is blocked until the decision comes. The choice of this solution is justified mostly by the number of blocked resources (1). In case the garage does not receive confirmation because of connection problems, the booking is considered still valid and the towing car can still bring the broken car to that garage. In case the Central server fails right after the towing is ordered, garage and towing services are reserved correctly, but the driver can be confused by not getting any notification. This emergency situation can be cleared out by a human operator. In the worst case the Central crashes down before towing is ordered, freezing the garage appointment unnecessarily. Here again, a garage assistant can contact the Central.

On the other hand, compensation approach is used for deposit payment that is immediately debited when service is requested. In case there are no services available, the deposit is compensated by cancellation. The choice of immediate payment is mostly due to a likeliness of at least some services being fulfilled and no reason to payment cancellation. Additionally, it is more complicated and possibly risky to block resources in a bank, which usually performs many other requests on customer's money.

4. Modeling

In order to design the system one needs to unambiguously and completely describe its structure starting with a problem statement and a set of boundary conditions. Since requirements are usually stated in an informal and textual form with imprecise descriptions of functionalities, one needs to clarify them as well.

From a general point of view, a good modeling technique should ideally be:

- standardized and open – to gain designers' acceptance and encourage popularizing process
- intuitive and precise in identification of requirements – to allow both customers and designers to reach an agreement about what should be accomplished
- clear and precise in describing both static and dynamic properties – to allow developers to clearly know what to implement
- suitable for different modeling styles such as:
 - state-based
 - event-based
 - data-based
- clear in specification of business process orchestration
- based on formal techniques and suitable for automation tools
- intuitive to use:
 - with useful graphical interface supporting design automation, and simulation
 - conformance to known standards and techniques – to attract experienced users and help them to get accustomed
 - compatibility with other mechanisms and tools – this encourages further development of plug-ins or extensions

Systems can be built using the following approaches:

- top-down – begin with a higher abstraction, then specify details of separate parts; this approach encourages outsourcing after the necessary blocks are identified
- bottom-up – begin with small blocks and bind them together as components

The choice depends mostly on designer's preferences and sometimes on system's character.

From a point of view of element structure, a model can be:

- informal (loosely coupled) – providing only descriptions of functionalities (like UML)
- formal – all elements are logically connected; this usually opens possibilities for supporting functionalities such as:
 - simulation – to allow a designer to execute a model
 - verification – either syntax or semantics may be tested automatically, increasing the reliability while decreasing testing time
 - transformations – to map a model into lower or higher abstraction for either design (e.g. abstraction refining), development (e.g. code generation) or verification purposes (see chapter 8)

From the perspective of SOA approach, design technique should support:

- abstraction – support different views on the architecture; capturing only those characteristics that are suitable for appropriate abstraction level
- modularization – to efficiently define and organize parts of complex systems
- scalability – standardized interfaces that allow further extensions
- various data types – to model a real system as close as possible
- various communication schemes – increases design options:
 - one-way,
 - request-response,
 - solicit-response,
 - notification
- compensation – to describe a widely used technique when performing long-running transactions
- performance – to allow making performance predictions even before a system is deployed
- different implementations languages – a designing technique should be abstract enough not to impose any particular technology upon developers
- manageability – control nodes should be implemented according to the design and may alter the behavior of a service; it should be possible to activate and deactivate them during the runtime; they may use control points like[BPDT]:
 - logger – logs all the traffic
 - filter – routes requests according to certain policy and priorities (for example when one service fails)
 - fail – allows termination of incoming requests under exception occasions
 - transformation – to transform messages before they reach destination points

4.1. Model Driven Architecture

Model Driven Architecture (MDA) has been initiated by OMG (Object Management Group) to unify UML models with transformations that can be performed on them. UML 2.0 [UML2] has been specifically created to improve modeling semantic behavior of SOA and MDA leading to more descriptive and precise models. Some of the introduced improvements are:

- complex structures – allows instances of structures to be represented with the roles they play; introduces real object ports instead of only descriptive interfaces
- activities – reusing interactions, control flows
- interactions – may contain reusable parts or their references
- state machines – state has been enriched with modularization possibility

During the design, an abstract model is iteratively refined into lower-level models in a continuing process improvement, increasing the understanding of the system. In some cases, it might be necessary to go back to correct higher abstraction model based on details from lower-level. Those models can be furthermore transformed to verify or create other models. In fact, even implementation can be perceived as a model that abstracts from an underlying system specific machine code. Each model has its purpose, beginning with showing tradeoffs between scope, cost and performance and ending with low-level details required for

implementation. A good model is one that precisely represents important parts and hides insignificant details.

MDA allows not only usage of many existing model checkers, but opens a whole new set of possibilities such as:

- precise specification – to allow not only to clarify behavior, but also, using descriptive simulation, to reach an agreement with a customer upon what should be created; it allows to order third party or out-sourced services with a reliability that they match precisely desired requirements
- scenario testing – before or even during the time a service is manufactured, a model can be tested against new scenarios; testing can be done without the trouble of setting up and configuring real hardware; abstraction from implementation allows for analysis and reproduction of larger scenarios
- automatic verification – to efficiently and reliably verify that a model adheres to certain properties
- automatic code generation – to create a skeleton of implementation where a programmer needs to fill-out only local operations (logic, dB, graphical interface, I/O devices, access control, etc)
- semantic interfaces [CASV,SDCR] that describe services by a whole interaction their ports support to allow:
 - dynamic system binding – unfamiliar services can inspect their ports and figure out how they can interact together
 - automatic compatibility checks – to assure that an upgraded system supports continuously its old services; new components that are replacing the old ones can also be checked whether they offer the previous functionality
- education possibility – model checking is a well documented and a popular technique with many good tools and case studies
- traceability – tightly connected elements allow changing one level of model abstraction and have the changes reflected in another
- identification of design patterns may suggest possible problems and their solutions
- early analysis and assessments of performance allows design of efficient architectures

4.2. UML

Although designed in 1997 by Object Management Group (OMG) for object-oriented techniques, with its expressiveness, UML is still useful for describing SOA. Its popularity is mostly due to flexibility in both showing different aspects of the system and allowing both experienced and beginning designers to present their ideas. Intuitive graphical notation allows sketching designs, discussing ideas and explaining problem domains. The standard is also continuously improved by the OMG and there may be many extensions that can be roughly divided by:

- lightweight – without changing the metamodel
- heavy weight – with metamodel modification

The main goals of UML are to:

- visualize – to understand a system better
- specify – to precisely describe both structure and behavior
- build – possibly create parts of design automatically by tools
- document – to save a current stage of development

UML specifies a set of structural (static) diagrams to catch compositional aspects:

- Class Diagram – presents static types of objects and their relationships
- Object Diagram – shows instances of objects
- Package Diagram – combines related classes into groups
- Component Diagram – shows structure and connection of components
- Deployment Diagram – shows a run-time configuration of static nodes and components that run on them
- Composite Structure Diagram – shows not only internal structure of a class, but also possible collaborations

UML also supports behavioral diagrams that show how a system operates:

- Use-Case Diagram – simple technique for capturing functional requirements that also provides an overview of interactions; very brief and usually not descriptive enough without a use case description but easily understandable by business users
- Activity Diagram – abstract process description suitable for modeling process logic
- State Machine Diagram – describes how state changes in response to events; models a sequence (single thread) which brings it much closer to hardware implementation, thus making it a great input for code generators
- Interaction Diagrams:
 - Sequence Diagram – sequence of interactions between objects
 - Communication Diagram (formerly Collaboration Diagram) – focuses on structural relationships originating from interactions
 - Interaction Overview Diagram – combination of sequence and activity diagrams
 - Timing Diagram – describes interactions focusing on their timing issues

In this thesis, most of the diagrams have been created by a free UML drawing tool: Umlet². Due to its complexity, Sequence Diagram has been created in a commercial IBM Rational Software Modeler³. Component and operation names in diagrams are intentionally abstract so as not to impose any particular implementation technology.

4.2.1. Use case diagrams

Use case diagram depicted in Figure 4-1 specifies use cases that identify all functional requirements. Use cases' descriptions are specified in Appendix C.

² www.umlet.com

³ <http://www-306.ibm.com/software/awdtools/modeler/swmodeler/>

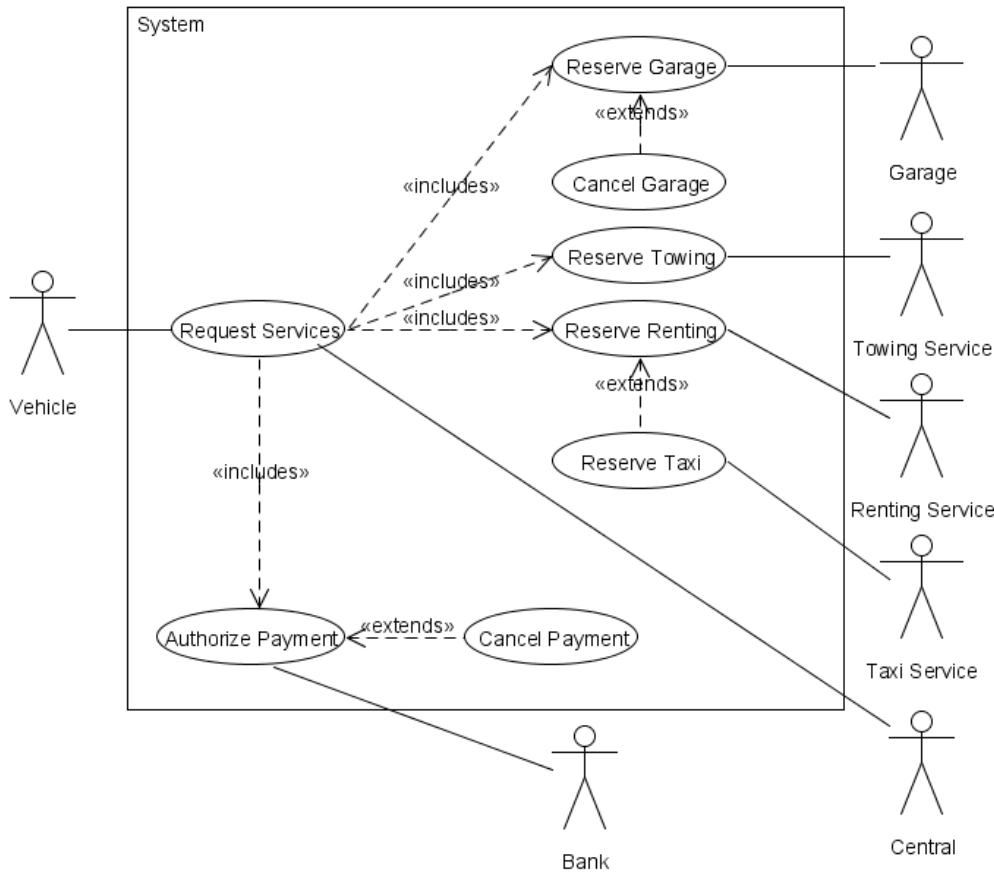


Figure 4-1 Use case diagram of system behavior

Use cases allow specifying requirements preferably involving customer in the process. They also allow a designer to get accustomed with the domain providing a good and systematic source of design ideas such as alternative scenarios, safety conditions, etc. Because of the textual representation, use cases are suitable only for simple process logic. Even with such abstractions like ‘includes’ and ‘extends’, it is impossible to show order of events. Due to the fact that the test scenarios are relatively simple and there was no customer to interact with, test cases could be done following the rule of thumb. A list of all use case descriptions is in Appendix C. However, in a real situation, a nice methodology showing how to gather scenarios is descriptively presented in Figure 6-48. Since use scenario creation is not of a particular focus of this thesis, a reader is referred to the source [PAVT] for details.

4.2.2. Activity diagrams

Having all participants and their use cases identified, it is necessary to describe process logic and order of actions. Activity diagram depicted in Figure 4-2 is suitable for showing the logic of a business operation.

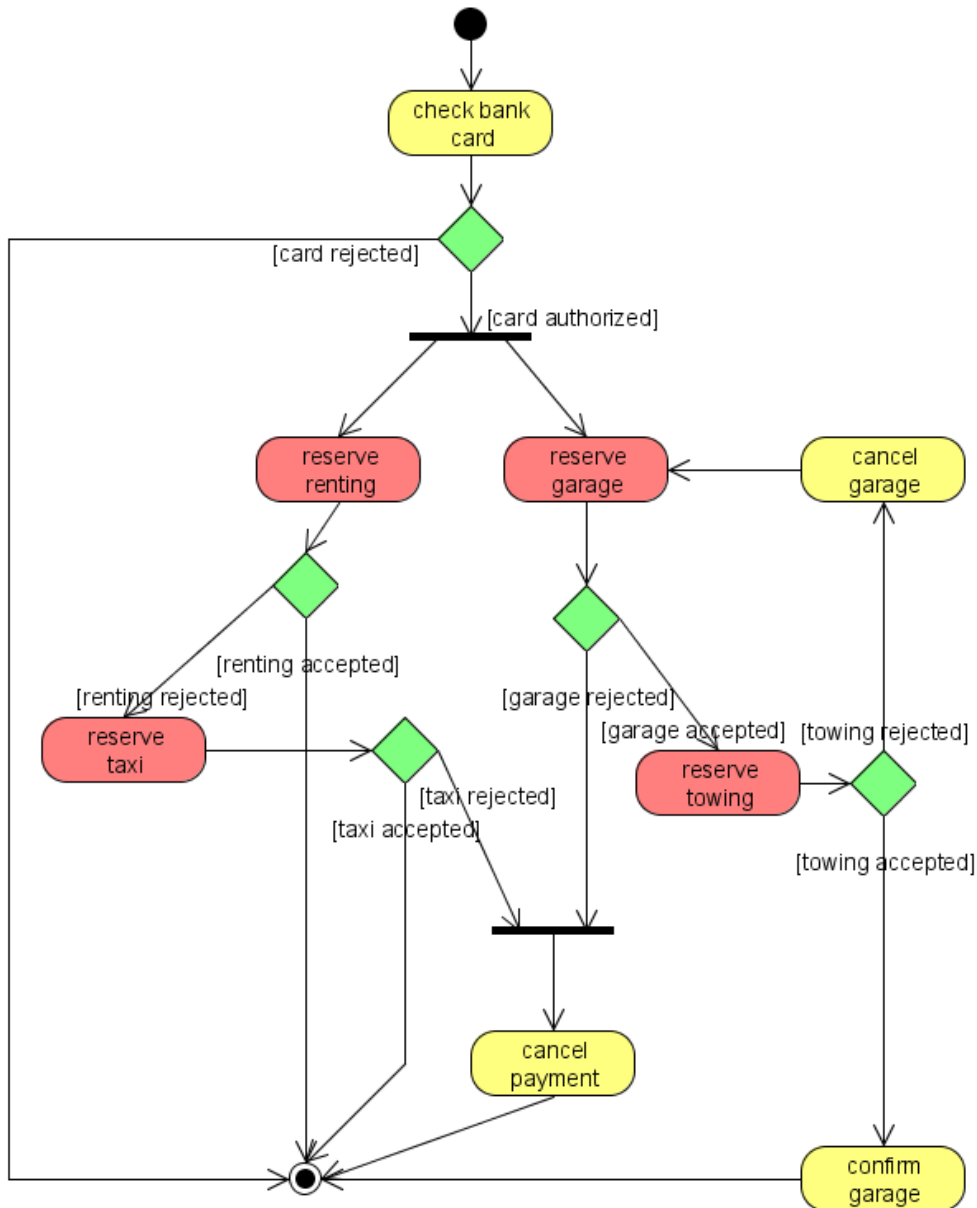


Figure 4-2 Request Service - Activity Diagram

4.2.3. Sequence diagrams

After the process logic is defined, it is possible to specify communication patterns between nodes with either sequence diagrams or communication diagrams. Both of them present the same behavior, but in different forms. A sequence diagram distinguishes the order of messages by placing objects on their life lines but can get unreadable for longer conversations. On the other hand, communication diagram allows convenient localization of interacting nodes. In some cases it allows grouping frequent communication partners together, thus clarifying communication arrows. Hence, sequence diagram can only describe a partial behavior of an interaction and only successful collaborations are presented.

The object oriented character of UML does not easily model some of SOA requirements. For example, operation calls in sequence diagrams are specified in advance to be either synchronous or asynchronous, while SOA supports also dynamic binding. Showing that communication scheme depends on service preferences requires an alternative execution flow “alt”, as depicted in Figure 4-3. However, for the sake of brevity, all further interactions are modeled as synchronous with a possible result returned by a function call.

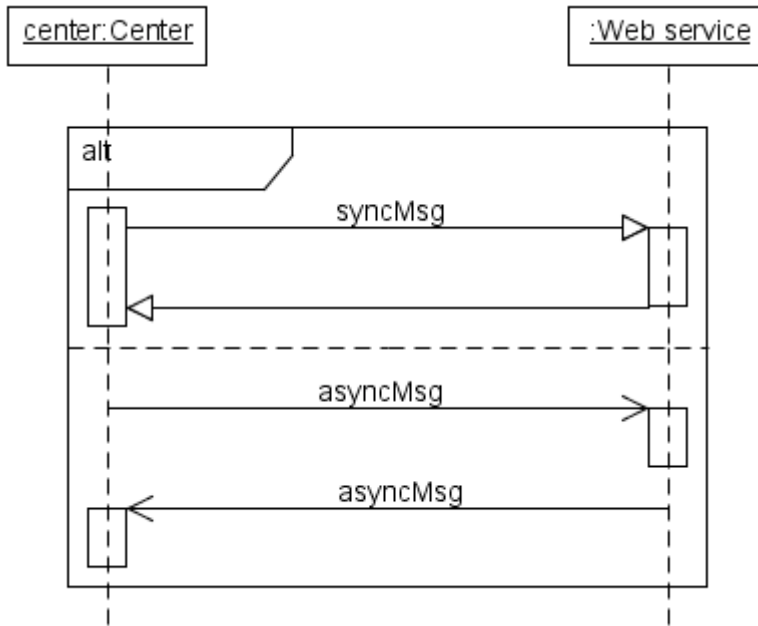


Figure 4-3 Synchronous and asynchronous communication pattern

Sequence diagram depicted in Figure 4-4 helps to clarify precise actions to be taken to fulfill business process. In addition, it is suggesting the workload for interacting elements. The busiest service, apart from the Central, is a garage. It takes part in three different services where two of them are in a loop (the third ‘Confirm garage’ also, but it launches only once before the loops ends). Multiple communications with one node should mark the node for future performance analysis and should also initiate a closer investigation to detect possible conflicting behaviors. It can be seen that despite additional control flow elements, complex interaction diagrams are rather difficult to read.

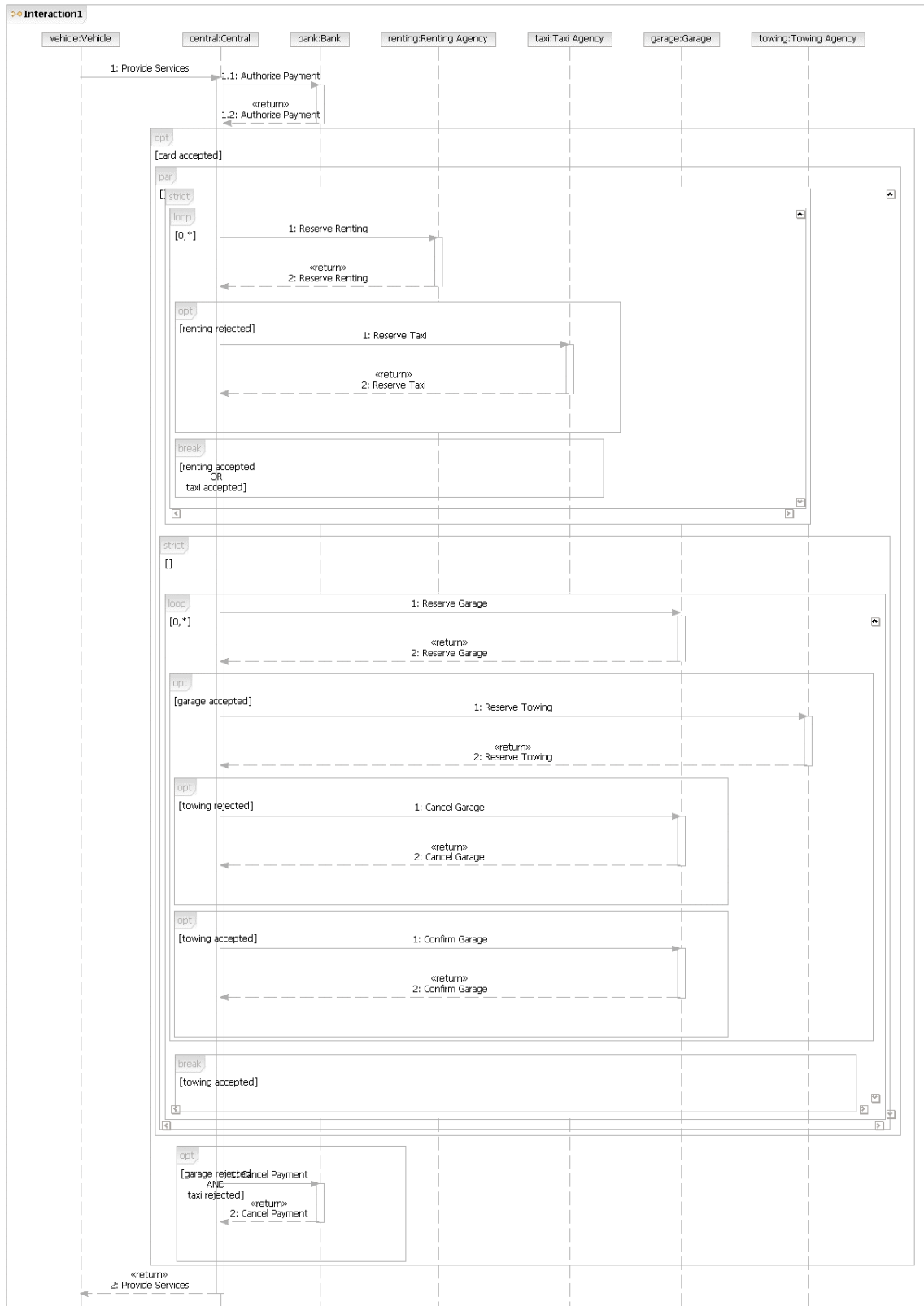


Figure 4-4 Request Service Sequence diagram

4.2.4. Communication diagrams

Communication diagram depicted in Figure 4-5 has been created automatically by IBM Software Rational Modeler [RSM] from the sequence diagram from Figure 4-4. In comparison, it focuses mostly on interactions overview between nodes rather than on their order. The described character of communication clearly identifies our client-server architecture. This is also an example on how different views can be used to verify each others correctness (sequence diagram in this case).

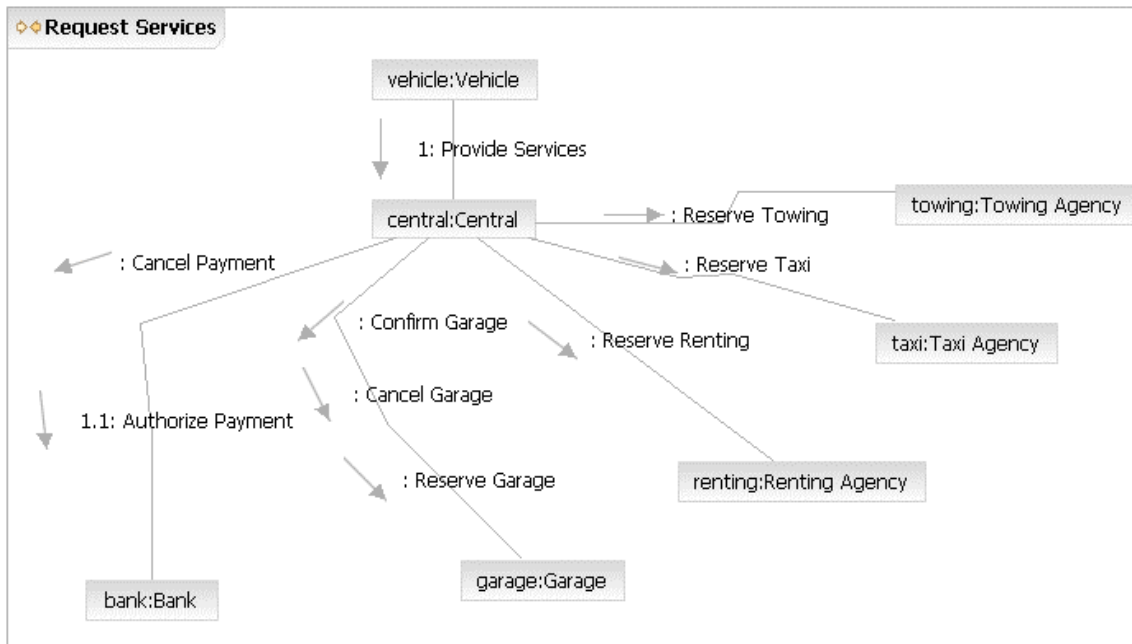


Figure 4-5 Request Service Communication Diagram

4.2.5. Component diagrams

In order to define architecture, it is necessary to specify interfaces and ports for components. For the sake of brevity, port names contain only first letters of components they bind (f.ex. V2C stands for Vehicle-to-Central).

- Vehicle ports
 - V2C – operations that Central requires from Vehicle
 - Services Rejected () – handles reply that credit card is not accepted
 - Services Unavailable () – handles reply that none of the services are available
 - Garage Available () – handles reply that garage (and towing) services are booked
 - Renting Available () – handles reply that only renting service is booked
 - Taxi Available () – handles reply that only taxi service is booked

- Central ports
 - C2V – operations that Vehicle requires from Central
 - Provide Services (failure details, location, credit card details) – request all available services from the Central
 - C2B – operations that Bank requires from Central
 - Card Accepted () – handles reply that payment is approved
 - Card Rejected () – handles reply that payment is rejected
 - Payment Canceled () – handles reply that payment is canceled
 - C2G – operations that Garage requires from Central
 - Garage Accepted () – handles reply that garage request is approved
 - Garage Rejected () – handles reply that garage request is rejected
 - Garage Confirmed () – handles reply that previously requested garage is confirmed
 - Garage Canceled () – handles reply that previously reserved garage is canceled
 - Booking Confirmed () – handles reply that previously reserved garage is confirmed
 - C2T – operations that Towing requires from Central
 - Towing Accepted () – handles reply that towing request is approved
 - Towing Rejected () – handles reply that towing request is rejected
 - Towing Confirmed () – handles reply that previously requested towing is confirmed
 - C2R – operations that Renting requires from Central
 - Renting Accepted () – handles reply that renting request is approved
 - Renting Rejected () – handles reply that renting request is rejected
 - Renting Confirmed () – handles reply that previously requested renting is confirmed
 - C2X – operations that Taxi requires from Central
 - Taxi Accepted () – handles reply that taxi request is approved
 - Taxi Rejected () – handles reply that taxi request is rejected
 - Taxi Confirmed () – handles reply that previously requested taxi is confirmed
- Bank ports
 - B2C – operations that Central requires from Bank
 - Authorize Payment (deposit amount, credit card details) – requests deposit payment authorization
 - Cancel Payment (deposit amount, credit card details) – requests deposit payment cancellation
- Garage ports
 - G2C – operations that Central requires from Garage
 - Reserve Garage () – requests garage reservation
 - Cancel Garage () – requests garage cancellation
 - Confirm Garage () – confirms garage reservation
 - Confirm Booking () – confirms permanently booking reservation
- Towing agency ports
 - T2C – operations that Central requires from Towing
 - Reserve Towing () – requests towing reservation

- Confirm Towing () – confirms towing reservation
- Renting agency ports
 - R2C – operations that Central requires from Renting
 - Reserve Renting () – requests renting reservation
 - Confirm Renting () – confirms renting reservation
- Taxi agency ports
 - X2C – operations that Central requires from Taxi
 - Reserve Taxi () – requests taxi reservation
 - Confirm Taxi () – confirms taxi reservation

Component diagram depicted in Figure 4-6 shows components

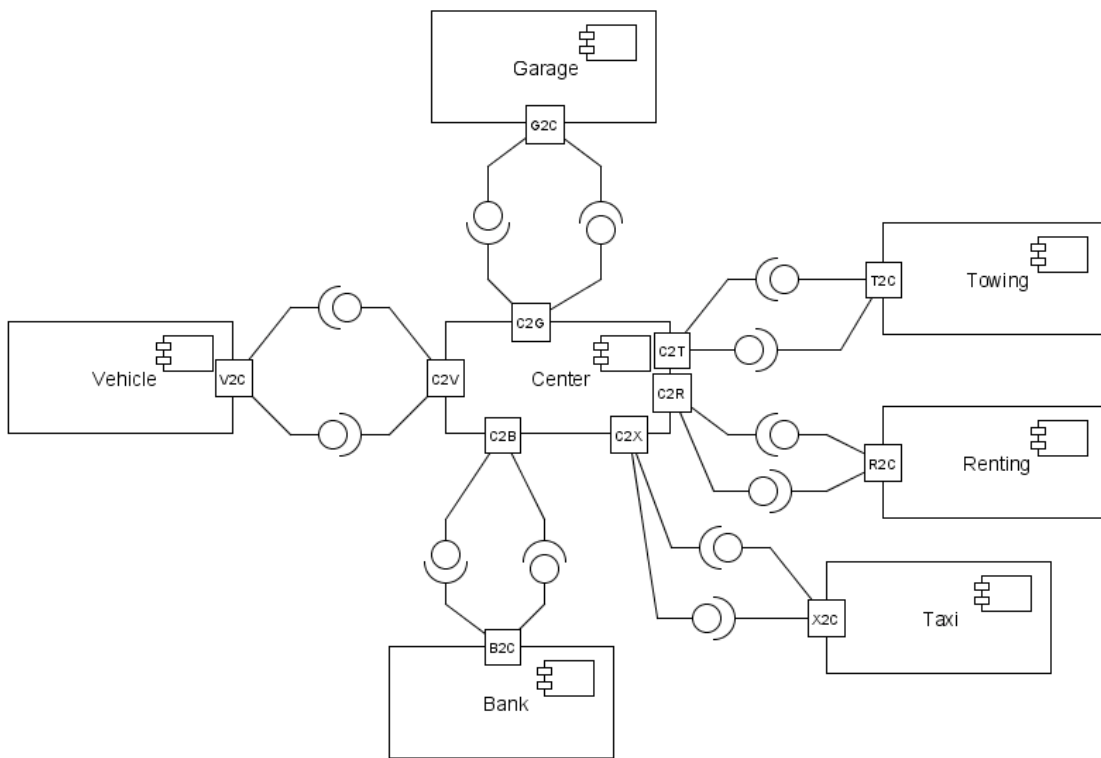


Figure 4-6 System - Component Diagram

4.2.6. Class diagrams

Even though components are not classes, a class diagram, depicted in Figure 4-7, is useful because of OCL constraints attached to methods. However, similarly to operation names, they need to be abstract as well, so as not to impose any particular technology upon developers. The class diagram also shows all the interfaces that each service realizes.

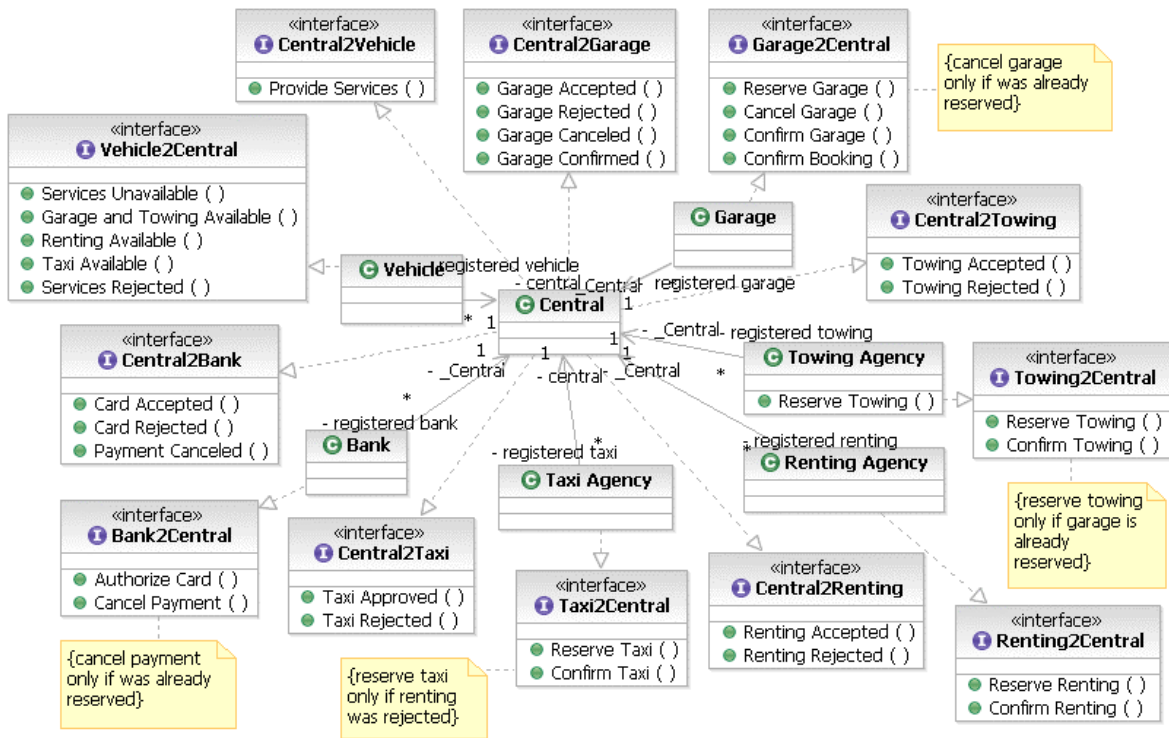


Figure 4-7 Class diagram of the system

4.2.7. State machine diagrams

After components are defined, it is possible to work on them independently or buying them from a third-company. However, even here, specifying them as models can significantly improve readability and compatibility to requirements, especially with respect to communication protocols. State machines can be used to model both component behavior, as presented in Figure 4-8 as well as protocol behavior as shown in Figure 4-9 and Figure 4-10. Because a state machine describes a complete behavior of a model, it can be further verified. The models presented here are verified by Uppaal in chapter 7.

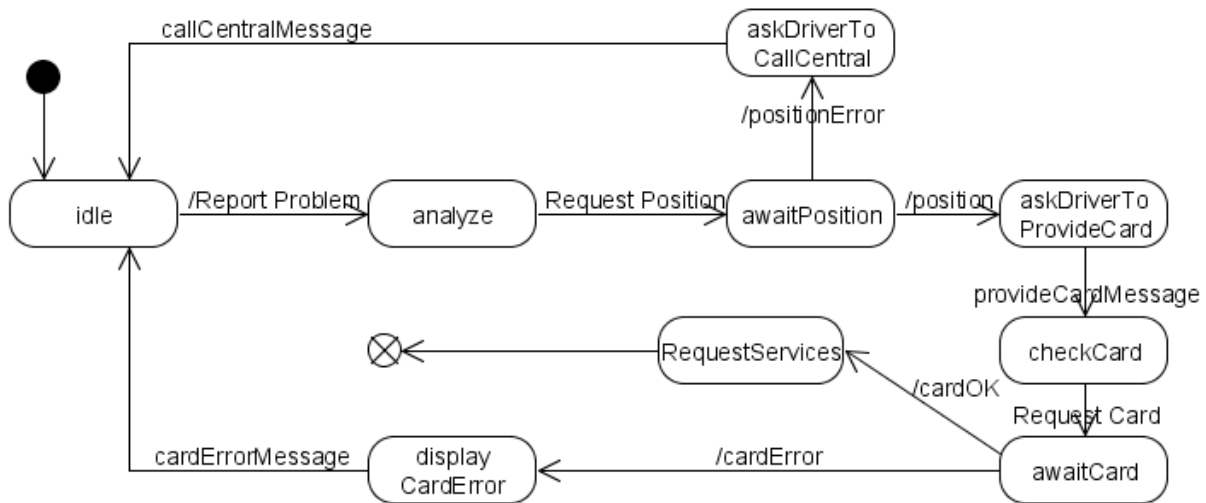


Figure 4-8 State machine of a vehicle component

For the sake of brevity, the state machine shows only behavior of Vehicle component and does not take timeouts under consideration. A complete model description with all additional processes (sensors, GPS, card reader, user interface, logging mechanism and a clock) is modeled in Uppaal in chapter 7.1. Component waits in state “Idle” until it receives a problem notification. Afterwards it checks vehicle’s current location and in case it is not available, a driver is informed to call Central. If the location is obtained, a driver is asked to provide a credit card which is accepted by a card reader. Once this is done, Central is requested to provide all services and the logic ends. Central’s response for the request is handled by another state machine.

Another suitable task for a state machine is testing protocols of interacting ports. A simplified state machine which shows the garage side of the protocol is depicted in Figure 4-9 while Figure 4-10 shows Central’s role in the interaction. Again, complete and detailed protocol is presented in chapter 7.2.

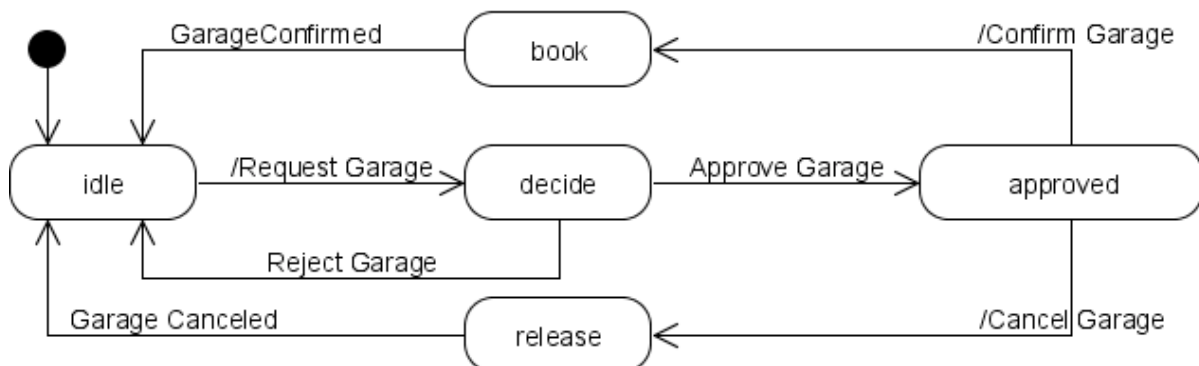


Figure 4-9 State machine of a protocol for port G2C from garage service’s perspective

After receiving a request, a garage service decides whether to approve or reject it. However, even after approval, it needs to wait for either confirmation or cancellation from the Central, to permanently book or cancel the reservation.

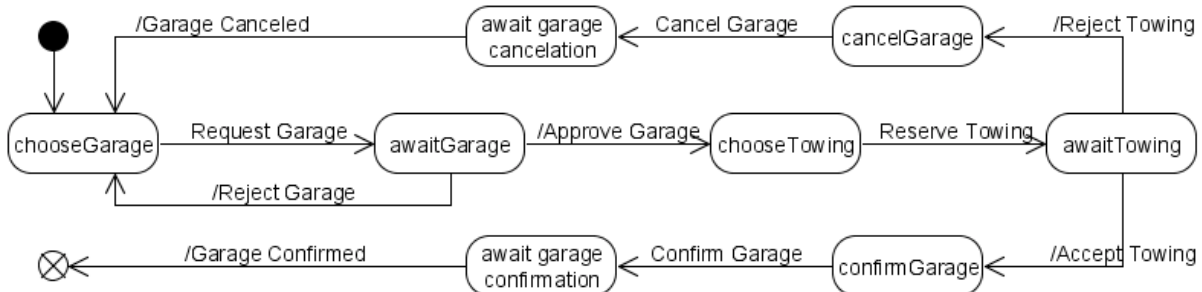


Figure 4-10 State machine of a protocol for port C2G from Central service's perspective

Central's task is more complex as it needs to synchronize both garage and towing services to provide proper service to the vehicle. Firstly, it requests a garage and awaits resolution. Next, if garage appointment is approved, it requests towing, again awaiting resolution. Finally, depending on the response, it confirms previously reserved garage (if positive reply) or rejects it, trying to book another garage.

4.3. Petri nets

Petri net is a formal and graphical modeling language invented in 1962 by Carl Adam Petri. It encourages abstraction and is suitable for modeling systems that are concurrent, asynchronous, distributed, and nondeterministic. Since the Petri net concept has been known for many years, many approaches have been developed on how to construct a system structure and choose communication patterns. This is an advantage for designers that can become skilled in Petri nets by reusing a considerable amount of documentation and already existing tutorial material.

A Petri net consists of following components:

- place – that may contain tokens representing current state of a system
- transition – active element that allows state to change by first getting enabled, and then firing; tokens are removed from transition's input places and placed in output places; transitions may additionally have guards to control activation conditions and assignments that change the data of a token
- arc – a directed connection between a place and a transition

Depending on data a token can pass, Petri nets can be classified [PN]:

- level 1 – places can have only one token containing a boolean value
- level 2 – places can have more than one token containing integer value
- level 3 – places can have more than one token containing high-level (complex data) values

Petri nets can also be classified according to connection patterns [WIKI] as shown in Figure 4-11.

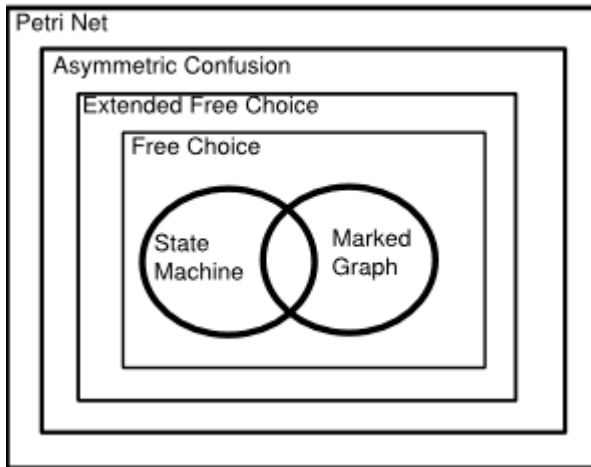


Figure 4-11 Graphical classification of Petri Nets [WIKI]

With the following types of networks:

1. State machines – every transition has exactly one incoming arc, and exactly one outgoing arc. This architecture prevents concurrency but allows non-deterministic decision on which path to choose.
2. Marked graphs – every place has one incoming arc, and one outgoing arc. This architecture allows concurrency, but prevents alternative paths.
3. Free choice – an arc is either the only arc going from the place, or it is the only arc going to a transition. This architecture allows both concurrency and alternative paths, but not at the same time.
4. Extended free choice – it is a Petri net that can be transformed into a free choice graph.
5. Asymmetric choice – both concurrency and conflict are allowed but not asymmetrically.
6. Petri net – allows all configurations.

From a flexibility standpoint, we consider Hierarchical Colored Petri Nets (HCPN) [JEN90] particularly interesting. HCPN is based on a functional language (ML) and supports flexible tokens types (color sets).

Definition (HCPN): “A hierarchical CPN is a tuple $HCPN = (S, SN, SA, PN, PT, PA, FS, FT, PP)$ where S is a finite set of pages (i.e. subnets), SN is a set of substitution transitions, $SA : SN \rightarrow S$ is a page assignment function, PN is a set of port nodes, $PT : PN \rightarrow \{in, out, i/o, general\}$ is a port type function, PA is a port assignment function mapping, for a given substitution transition, its sockets with its subnet’s ports, FS is a finite set of fusion sets, FT is a fusion type function, and PP is a multi-set of prime pages.”[FCGC]

HCPN are very flexible to represent any kind of abstraction, especially to model workflow of services in SOA. They allow modeling mechanisms such as: synchronous and asynchronous data passing, abstraction, composition, concurrency and correlation. They also provide an intuitive mechanism by pattern matching tokens approaching a join transition. An extensive example of HCPN usage is presented in chapter 6.

4.4. Modeling tools

There is a variety of tools that aid designers in their task. One can categorize them according to the standard they support:

- UML
 - Umlet [UMLET] is a free, Eclipse plug-in editor that supports visualization of most of the UML elements
 - IBM Rational Software Architect (RSA) [RSA] is a commercial toolset model creator based on Eclipse; it supports model-driven approach and code generation to several languages
 - IBM Rational Software Modeler (RSM) [RSM] is also commercial toolset included also in RSA; RSM supports model-driven development but without code generation
 - Ramses [RAMZ] is maintained by NTNU and supports syntax and structure verification mechanisms to model a state machine for generating java code
- State machines
 - Uppaal [UPP] with a free license for non-commercial users; allows modeling and verification of state machines; because of the focus on embedded systems, it does not easily support many of SOA key functionalities like abstraction mechanism, concurrency modeling or data passing
- Petri nets
 - CPN Tools [CPNT] offers a free license to create HCPN that are flexible enough to model SOA systems; CPN Tools has an intuitive interface with abstraction mechanism that allows easily creating systems in both top-down or bottom-up approach; there are also several other functionalities that ease designer's work, like syntax checking, partial (or on-the-fly) simulation, flexible ML based type system, etc.; it is elaborated more in chapter 6.

All of the aforementioned tools are suitable for modeling SOA. However, Uppaal capabilities do not allow clear and intuitive representation of mechanisms like concurrency or compositionality.

4.5. Modeling conclusions

UML has proven to be a valuable technique that allows describing a required behavior of a system in various abstraction levels. Because of its intuitiveness, it is helpful for both a designer and a customer to specify requirements. Even though the models are not connected by any formal structure, creating diagrams clarifies various aspects and mechanisms, highlighting many mandatory properties a system should hold. Especially use cases descriptions, listed in Appendix C, and OCL constraints in a class diagram provide a systematic chain of iterations to specify system's behavior and define invariants— see: Appendix D – Verification properties. Those conditions are a necessary input for the formal reasoning presented in chapters 6 and 7.

One can also notice that the scenario extension: 'Taxi service' can easily be added as well as removed from all diagrams, which shows a scalability of the model. The only drawback is that it has to be removed from all diagrams separately (since they are not connected) or else, the views will be inconsistent. Some toolsets, like IBM Software Rational Modeler [RSM],

support removing an element from the whole model (from all views/diagrams) with one command only.

Another interesting formal approach, invented 15 years ago by L. Lamport, is based purely on the temporal logic he invented. TLA+ is very expressive in specifying system behavior and requirements in the same notation. There is additionally a free TLC model checker [MCTS] to verify both security and liveness properties.

There is a significant research based on this logic, covering different aspects:

- an extensive 382 page long manual by L. Lamport [SSTL]
- specification of a web services protocol [FSWS]
- verification of UML constructs [TCSS, TVEU, AUSM, SSCC, VURS]
- transformation to TLA+ and further verification of SDL [TFVT]
- verification of a trust model [TLSV]
- comparison with Uppaal in real time model checking [RMCR]
- extension to cover real-time and continuous properties [SHSC]
- a number of case studies [ICS, SSM, HSLI, FSPV, FMTP]

Despite the focus on the typical engineering approach, the technique is quite difficult to use. The text-based specification quickly gets to big to comprehend. It seems that it focuses on expressiveness and by doing so, it loses its intuitiveness. It offers high configurability and is believed to be suitable for specifying protocols between interacting nodes together with their properties and data, such as counters, mutual exclusion, etc. However, TLA seems to be appropriate for more tightly coupled systems like microprocessors, memories, sensors, etc. It would be, however, interesting to compare it deeper with other verification mechanisms on how to automatically generate or transform TLA specifications.

Interestingly enough, there is no widely adopted design technique for SOA. UML is mostly used in object-oriented systems, state machine models focus on real-time systems and Petri nets are researched mostly in academic circles. Despite being a buzzword, SOA does not seem to get a proper tool support⁴ to allow engineers to use its full capabilities in practice. Instead of defining new systems, SOA is being used for correcting flawed architectures, promoting a corrective approach rather than a constructive one [ISDD].

In the absence of a efficient modeling standard, an experiment is to be carried out to exploit combined functionalities of chosen modeling mechanisms (UML, CPN Tools and Uppaal) for designing SOA. It is intended not only to highlight their limitations, but also to reach a consistent methodology on how to describe all aspects of SOA and efficiently verify them.

⁴ <http://searchwebservices.techtarget.com>

5. Verification

After modeling stage, when all system functionalities are described, a model needs to be verified. While there are good tools for checking low- and mid-level models, it is hard to find one for high-levels. In some cases, there may be bugs that cannot be found only by looking at individual components, but only on their combined functionalities.

Despite being based on objects, SOA allows for a new kind of verification. Traditional problems of local applications like buffer overflows, memory and CPU consumption or communication issues are still applicable, but for individual components, designed in OO approach. Nevertheless, SOA allows abstracting away from technical, platform specific details enabling verification of business process. Thus, it is important to target verification at the actual outcome of service interactions rather than how individual processing steps are achieved.

Verification can be performed during following stages of a service lifecycle:

- requirement specification
 - human testing ('peer reviewing') – usually the only solution because of the informal form; it highly depends on a tester, who, due to the amount of details, is often as error prone as the tested specification itself
- design
 - human testing – widely used due to informal nature of popular design techniques like UML
 - automated theorem proving – produces a formal proof through rigorous mathematical techniques, system specification and a set of inference rules; an example for SOA is presented in [COWS]
 - dynamic analysis (model checking) – requires creating and verifying all possible states of an executable finite-state model
- implementation
 - testing – checks a system evaluating output response to valid and invalid inputs; as commented by E. Dijkstra: "program testing can best show the presence of errors but never their absence"
 - black box testing – test cases are derived without knowing inside details
 - white box testing – test cases are derived based on internal structure
- deployment
 - testing – requires a working prototype to apply a number of test data, checking whether the model fulfills its abstract functionality; sometimes end product testing is necessary even after extensive design verification in case:
 - of complexity that prevents a model to be build
 - some parts (like sensors) cannot be modeled
 - component details are restricted (proprietary)
- runtime
 - monitoring – allows checking at run-time that system behaves correctly by investigating:

- order of executed services to detect unexpected sequences
- messages' patterns to detect incorrect behavior, such as duplicated replies
- time when services are queried and their replies allows identifying bottlenecks

Since the test scenario is imaginary and no real system will be implemented, this thesis focuses on verification in the design phase. Additionally, an unquestionable advantage is that early elimination of problems is cost-efficient by not needing to spend valuable resources on multiple implementation-testing-debugging iterations. From the variety of verification techniques, model checking, explained in details in next section, seems to be the most suitable. This choice has been motivated not only by its formal and proven mechanism but also by the amount of SOA research and available automated tools. In addition, the advantage of having components specified as models opens a future possibility of implementing semantic interfaces [CASV, SDCR]. This new technique allows verification of components behavior through their external interfaces, rather than inspecting all their internal states, although limiting significantly a state space of the system.

There are several techniques to create models depending on requirements and the desired features. Three of the most prominent standards are:

- UML
- Petri nets
- State machines

1. UML

Since UML does not have any underlying formal mechanism, verification capabilities are limited to manual testing and OCL. However, OCL is usually a suggestion for a developer on how to implement particular elements, what preconditions to assume, etc. An indirect verification is done through multiple views with usually overlapping parts that provide control by redundancy which makes it possible to compare different views against each other.

The main shortcoming is that those views are not connected by any formal backbone that could allow a system to be verified as a whole. Additionally, there exists a risk that all views are not complete and so do not present all properties of the system. Some editors, like IBM Rational Software Modeler [RSM], allow verifying syntax and element consistency between views but do not analyze interactions between components.

2. Petri nets

Petri nets allow modeling interactions between different components of a system while being able to analyze concurrency issues. There exists a long tradition of research providing a good formal technique for abstract (high-level) verification. There are also model checkers that enable both manual and automated verification. Chapter 6 describes comprehensive usage of CPN Tools, a specialized Petri net suite that includes a model checker.

3. State machines

While Petri nets are designed to model concurrency, a state machine simulates behavior of a single execution thread. This resemblance to low-level hardware realizations brings the

design much closer to real implementation. In addition, there are also a number of various code generators based on state machines as an input [RAMZ, RSA]. Low-level verification is popular and widely investigated due to safety-critical nature of electronic devices in use. Not only can state machines be quickly created by informal UML but also there is also a formal approach that allows their simulation and verification. There are some excellent design and verification tools, such as Uppaal, that support state machine design, enriching them with the necessary formal framework. It allows model checkers to efficiently and reliably verify full or partial state space against safety and liveness properties. An example of a verification of a component and protocol based state machine is in chapter 6.

Because of the lack of formal backbone in UML, only Petri nets and state machines offer satisfactory verification possibilities. However, state machines (as well as their tools) are designed to model small real-time systems rather than SOA. Thus, some techniques such as concurrency or abstraction mechanism are not usually supported. Additionally, because of their precise low-level approach it is possible (especially in timing verification) to experience a state space explosion while modeling big systems. Petri nets, on the other hand, seem to focus more on intuitive design supporting abstraction and concurrency. Since it is easier to realize process logic in Petri nets, they are preferred in the later system design.

5.1. Model checking

Once a model driven architecture is pursued, one needs to find technologies that can support this approach. One of them is model checking, which bases on a finite-state model of a system. Model checking is a widely known and used technique for validating both low-level and high-level designs. In order to use it effectively, one has to learn its mechanism and theoretical background lying underneath.

Model checking techniques support following verification techniques:

- simulation – allows for a quick and informal verification of whether the model fulfills its abstract functionality; usually not suitable for testing all the scenarios but only the most important ones; depending on the triggering mechanism of each simulation step, it can be:
 - manual – which requires human involvement
 - automatic – allows automatic progression of model's process
- state space analysis – allows analyzing statistical information about the whole state space, such as number of nodes, possible tokens in every place, fairness properties or dead transitions
- formal verification – automatically and systematically checks a given property, described in some logical formalism, by inspecting every possible system behavior; it requires some expertise from the designer to use it; it can extract a proof from a model that contains a precise specification of:
 - set of states (locations, places)
 - transition relation (rules of how states change)
 - verification formulas – that define what property to check

Comparing to the popular white and black box testing, formal verification has an advantage that more than one test case can be checked at a time. Comparing to white box testing, input data do not have to be chosen based on the executable code, but according to the business logic, thus allowing non-technical persons to inspect a model. By not relying on the implementation, test conditions do not have to change when model process changes, provided that tested functionality is kept.

However, model checking is not a panacea for every kind of system. It is not suitable for data-intensive interaction which usually generates an infinite state space. It is also difficult to create a model of infinite-state systems such as protocols. There is always an issue of finding a proper abstraction for designed architecture that would clearly show both physical and logical dependencies. Even if a model is created, it is arguable whether it conforms to the real system since model checking is only as good as the model. Additionally, even finite-state models may generate a state space so big that it severely influences verification performance or might be even too demanding to compute. This situation is referred to as state space explosion. In addition, verification properties need to be specified manually and may be incomplete, leading to untested functionalities. Finally, model checking tools should also be formally verified, which leads to a problem how to check a checker.

However, despite its disadvantages, model checking seems appropriate for SOA verification. Because of their loosely-coupled and distributed nature, services usually perform data extensive operations locally and interact usually in simpler communication schemes through their interfaces. That allows a designer to conveniently abstract away from uninteresting (from global point of view) implementation details that could lead to state space explosion. Such compositionally defined services can have elegant process logic with a finite number of states.

A model checker performs a search through all possible states in order to determine whether a given property is false or true. It is very convenient if the result also contains a counter example, for unsatisfied properties, to quickly find the cause of the improper behavior. Depending on the formula and checker's algorithm, the state spaces (also called occurrence graphs, reachability graphs or reachability trees[CTSSM]) can be explored in a forward, backward, breadth-first, depth-first or a combination of those strategies. Reachability properties, for example, can be efficiently checked in a backward depth-first direction towards the initial node and terminate the exploration when it's reached. Safety properties, on the other hand, usually require checking all possible states.

An interesting idea is a partial state space analysis where only a fragment of a state space is generated. The verified computation paths may be chosen either according to earlier defined properties or at random. That allows analyzing systems where full state space is too big or takes too much time to explore. Even though a partial analysis cannot prove general correctness, it can help by identifying errors that would also appear in the full analysis.

State space explosion is the biggest problem in model checking even though currently available tools (CPN Tools) allow to analyze occurrence graphs up to half a million nodes and one million arcs [IPUC]. Depending on a model checker and system specification, a designer can usually reduce a state space by techniques such as:

- resetting unused variables
- grouping states in atomic entities (abstraction)
- eliminating redundant elements

Because of the complexity of today's models, a good tool may use not only a state-of-art logic engine but also an optimization mechanism introduced in a model checker such as:

- over-approximation – defining boundaries of behavior that a whole state space fits in
- under-approximation – defining boundaries of behavior that fits into a whole state space
- heuristics – allows optimized strategy of exploration of a state space
- eliminating symmetries in the modeled system
- grouping and abstracting away from concurrent or strongly connected elements [MCCP]
- creating sub-graphs for isolated modules that would be seen as few (or even one) states

Those additional techniques may in some cases allow verifying state spaces that are too big.

5.2. Computation Tree Logic

In order to specify the required verification formulas, a designer needs to use precise logic formulas with machine understandable and verifiable operators. Among many different possible logic standards, we describe Computation Tree Logic (CTL) that is supported by tools used later on. CTL is an example of a branching temporal logic with a tree-like structure of state space [WIKI].

The language syntax of CTL can be described by context-free grammar presented in Figure 5-1.

$$\begin{aligned} \phi ::= & \perp \mid T \mid p \mid (\neg \phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid (\phi \Leftrightarrow \phi) \mid \\ & AX\phi \mid EX\phi \mid AF\phi \mid EF\phi \mid AG\phi \mid EG\phi \mid A[\phi U \phi] \mid E[\phi U \phi] \end{aligned}$$

Figure 5-1 Context-free grammar of CTL logic

Apart from the logic operators (not, and, or) there are also temporal path operators which specify path choice [WIKI]:

- $A \phi$ – property ϕ has to hold on all paths starting from the current state (all)
- $E \phi$ – there exists at least one path, starting from the current state, where ϕ holds (exist)

Properties of a single path can be additionally characterized by state operators:

- $X \phi$ – property ϕ has to hold at the next state (next)
- $G \phi$ – property ϕ has to hold on the entire consequent path (globally)
- $F \phi$ – property ϕ has to hold somewhere on the path (eventually)

There is a free choice of temporal operators in a formula but path operators (A and E) have to be used in combinations with state operators (X, G or F).

CTL has a minimal set of operators, shown in Figure 5-2, which allows representing (after transformation) all CTL formulas and is useful for model checking automation.

$$\phi ::= \text{false} \mid \vee \mid \neg \mid EG \mid EU \mid EX$$

Figure 5-2 Minimal set of operators for CTL

Apart from the versions of CTL supported by our tools (CPN Tools and Uppaal) there are also other variants and extensions to CTL logic [WIKI]:

- Probabilistic CTL – allows for probabilistic quantification of described properties
- Fair Computational tree logic – focuses on explicitly defined fairness constraints
- Linear temporal logic (LTL) – allows formulating queries referring to time

5.3. Verification tools

Verification tools take a lot of the burden from a designer by checking properties automatically. They support many different standards in terms of input models and verification formulas. The quality of a verification tool can be measured by the size of a problem that a tool can analyze and the verification properties that can be tested. Depending on those two inputs, the verification performance varies.

There is a whole variety of model checkers that allow specifying a system and checking certain verification conditions but the most relevant are:

- Spin [SPIN] – based on a Promela’s specification and supports modeling and simulating of asynchronous distributed algorithms as non-deterministic automata[WIKI]; properties are specified in LTL logic and can detect:
 - deadlocks
 - unspecified receptions,
 - flags incompleteness,
 - race conditions,
 - unwarranted assumptions about the relative speeds of processes
- Uppaal [UPP] – “is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types”[WIKI]; it supports:
 - graphical editor
 - model checking and simulation (also with timed automata)
 - properties can be specified in a subset of TCTL logic that does not support full TCTL, but focuses only on safety and liveness properties
 - generating diagnostic trace of a counter example
- CPN Tools [CPNT] – allows a variety of different analysis:
 - simulation can be done on-the-fly while a system is built
 - automatic monitor highlights all syntax errors
 - state space report generation
 - many state space predefined and customizable queries
 - performance analysis
 - multiple extensions like ASK_CTL enrich the functionalities with a subset of TCTL logic

- Hugo/RT [HUGO] – “is a UML model translator for model checking, theorem proving, and code generation: A UML model containing active classes with state machines, collaborations, interactions, and OCL constraints can be translated into the system languages of the real-time model checker UPPAAL, the on-the-fly model checker SPIN, the system language of the theorem prover KIV, and into Java and SystemC code.”[HUGO]
- TLC [TLC] – is a model checker for specifications written in Temporal Logic of Actions (TLA+); has simulation mode that generates random behaviors instead of state space; it can verify deadlocks as well as liveness properties, also generating a counter example in case of unsatisfied property; verification is, however, less efficient [RMCR] than in Uppaal.

5.4. Verification properties

Because of the countless amount of element configuration, it is impossible to automatically verify systems for all kind of problems. It is still the experience that suggests a designer what kind of problems a particular system may be vulnerable to. The most abstract classification of properties that may be evaluated is:

- qualitative properties – can be observed, but generally not measured; they are usually connected with subjective impressions and in terms of a SOA modeling can describe:
 - clarity of a model
 - ethical aspects related to prioritizing certain requests and users (fairness)
 - tradeoff between reusability and efficiency
- quantitative properties – are the most popular in engineering because they can be measured in a defined units multiplied by a number

In order to classify services from the perspective of expected behavior, following property categories are worth to mention:

- safety – characterized generally as “something bad will never happen” ensure that certain behavior is detected, such as:
 - deadlocks that infinitely stop service execution
 - invariants describe forbidden states
 - service failing to fulfill a task will not harm a system
- liveness – described as “something good will eventually happen” defines desired behavior and may be inspected by inevitability formulas that check whether a certain behavior is going to happen, usually discovering unpredicted executions and infinite loops in process logic
- reachability – “something good will possibly happen”

Those generic properties can apply to all kinds of systems including SOA. Because of their abstract nature, they can specify most of the desired behavior modeled in the design phase.

Additionally, a service may have properties that are important from the perspective of:

- service requestor – focuses on performance and reliability of a service against (possible) usage cost
- service provider – as a typical business unit always evaluates cost of maintaining a service against profits

A requestor has usually certain expectations regarding a service which may be in general referred to as quality of service (QoS):

- performance – is an example of a dynamic program analysis that investigates data in a running model in order to determine possible optimizations (response time, delays, speed, throughput, memory usage)
- fault-tolerance – shows procedures to handle both invalid requests as well as internal errors; since services are often dependant on each other, even one failing node can be a bottleneck for the whole system
- security – determines a trust a user puts into requestor's company
- compatibility – can be typically defined as adherence to popular standards and includes clear interface description
- consistency – neither behavior nor final states should change unexpectedly

In terms of evaluation of systems security can be define by:

- confidentiality – protects user's data before unauthorized access; it is especially important because of the insecure communication medium (Internet) that may be exploit for a man in the middle, replay or cut and paste attacks; confidentiality contains other parts like:
 - authentication – identifies correctly valid users
 - encryption – prevents disclosure of information to malicious parties
 - privacy – prevents leakage of personal information
- integrity – protects data against unauthorized modification, whether by accident or deliberate
- availability – ensures that a service is always accessible for users

A mixture of those properties defines a reliability of a service and, as a result, user trusts in it.

Apart from the user-centric properties, a service provider has also requirements regarding a service:

- scalability – how easy it is to extend a system in future
- upgrading – describe procedures that need to be followed to change an already running system; it is desirable not to interrupt providing service and should be transparent to customers
- manageability – shows additional control nodes that can be implemented in the system and may change service behavior in the run time
- boundedness defines a maximum workload that a service can efficiently serve; it may influence service's reliability if it is not sufficient; it is also related to defining necessary procedures when the usage limit is reached such as prioritizing some requests or random service denials
- reusability – evaluates how easy it is to use the service in another system configuration and can be influenced by
 - compositionality – possibility to combine functionalities of different services to complex services
 - specialization – possibility to restrict the usage of services by defining its desired features
- absence of self loops – to avoid unnecessary actions and circular dependencies that may affect performance

- resources usage – characterizes a cost of running a service such as database maintenance or power consumption

As explained in the beginning of this chapter, since the verification is performed on the design phase, not all properties are suitable. Some of them, like delays and throughput may be estimated but are best to measure in a running system. Others, like security policies or fairness of fault tolerance could be introduced in the design, but since they have not been defined in requirements and are out of the scope of this thesis. Therefore, the further approach focuses on verification of safety, liveness and reachability properties as sufficient to analyze a model in the design phase. It can be accurately argued that liveness properties prove also reachability tests and could replace them. However, in many cases, reachability properties can be easily and efficiently not only formulated, but also verified. They usually require fewer resources to compute and may be used even when a complete state space is too big to handle. Thus, reachability properties can be used as a pre-verification before the other – more demanding queries.

6. CPN Tools

CPN Tools is a successful application that has been created in 2002 and is continuously upgraded by the CPN Group at the University of Aarhus. CPN Tools supports a flexible High-level Color Petri nets (HCPN) which allows to efficiently organizing a model with intuitive abstraction and flexible token (in terms of data type). Some of distinct design modeling styles are to be later investigated together with their verification consequences.

It has proven to be a good tool for modeling and analyzing various kinds of systems. There are many published papers and test cases which show how to use CPN Tools with respect to:

- payment transaction protocol [REPT]
- edge router discovery protocol for mobile ad hoc networks [SVER]
- mapping and execution of UML models[RVEU]
- code generation for an access control system [ACGC]
- executable code generated from a Petri net through an intermediate ML code [ACGC] or directly to Java [RENEW]
- and many other industrial use [EIUC]

Most prominent of CPN Tools features are:

- intuitive abstraction mechanism by sub-pages
- flexible logic system based on a general-purpose ML (metalanguage) functional programming language
- on-the-fly simulator – allows to conveniently simulate a net while it is being created
 - manual – triggering of transitions requires user actions
 - automatic and semi-automatic – can trigger randomly a number of simulation steps
- intuitive graphical interface – a designer can personalize a workspace with pages and binders; has efficient and context sensitive marking menus

- model editing – is enriched with many time-saving mechanisms such as suggestion of color-set type for a place
- syntax checking – automatically verifies correctness of a created model
- state space analysis – reveals statistical information about not only the whole state space but may also highlights possible problems with respect to:
 - dead transitions – pointing to never used operations
 - dead markings – suggesting a probable design error if there are too many of them
- built-in queries – investigate a previously generated state space; in addition one can also formulate custom queries in a flexible and straightforward ML code
- partial state space exploration – possibility of selecting a start point, of desired state space in a chosen place of simulation execution, to allow partial verification
- performance analysis – allows monitoring multiple executions in order for further data analysis
- many extensions – enrich the functionality with additional visualization and verification functionalities using plug-ins such as:
 - ASK_CTL [DAM] – introduces CTL logic in state space verification
 - BRITNeY Suite [BRIT] – animation framework supporting 2D and 3D state space visualization, message sequence charts, etc.
 - Graphviz [GRAP] – is a graph visualization software that automatically draws graphs from structured data such as a state space

Verification

As a crucial functionality of a tool, CPN Tools allows several verification techniques:

- syntax checking – bugs are highlighted in net structure showing inconsistencies such as incorrect inscriptions and guards
- reachability properties can be verified by:
 - manual simulation until desired state is achieved
 - analysis of a state space with either predefined ML queries or ASK_CTL plug-in
- boundedness – defines how many tokens may be present in a certain place; it depends on token-generating elements like forks and is descriptively shown in a state space report
- liveness can be checked by:
 - manual simulation – by checking all possible executions to be sure that all execution paths are covered; the technique is reliable only for simple models
 - analysis of state base with ASK_CTL extension
- safety invariants can be discovered by
 - manual simulation – a designer can inspect the behavior partially while monitoring conformance to desired properties; complete analysis is reliable only in small networks
 - analysis of state space report – some simple invariants, like deadlocks, are revealed by dead transitions
 - ASK_CTL – allows automatic state space verification with a subset of TCTL logic and is explained in details later
- performance analysis – is carried out by analyzing information based on the behavior on a model during an automatic simulation; additional delay and random variables allow for statistical estimation of:

- average buffer workload
 - resource utilization
 - end-to-end delay
 - system reliability by introducing service or communication failures
- statistics – that describe state space size (nodes and arcs) and its generation time
 - fairness dependency – determines an overview of possible transition execution relations and can point out design flaws such as infinite loops

ASK_CTL

A state space (SS) generated by CPN Tools can be used not only to generate a standard report or to be inspected by standard queries, but also to verify much more specified formulas defined in ASK-CTL. ASK_CTL [DAM] is a library that extends verification options with a CTL-like temporal logic and makes it possible to reason about both state and transition attributes. It is interpreted over the state spaces in the CPN Tools by a model checker, which verifies an ASK-CTL formula, returning a corresponding truth value (true or false).

ASK_CTL is built in CPN Tools, but using it requires loading the library by evaluating a query shown in Figure 6-6-1. After activation, the right part of the figure shows available operators including their data type.

```
opening ASKCTL
datatype A = ...
val NF : string * (Node -> bool) -> A
val AF : string * (Arc -> bool) -> A
val TT : A
val FF : A
val NOT : A -> A
val AND : A * A -> A
val OR : A * A -> A
val EXIST_NEXT : A -> A
val FORALL_NEXT : A -> A
val EXIST_UNTIL : A * A -> A
val FORALL_UNTIL : A * A -> A
val MODAL : A -> A
val EXIST_MODAL : A * A -> A
val FORALL_MODAL : A * A -> A
val POS : A -> A
val INV : A -> A
val EV : A -> A
val ALONG : A -> A
val eval_node : A -> Node -> bool
val eval_arc : A -> Arc -> bool
val it = () : unit

use (ogpath^"/ASKCTL/ASKCTLloader.sml")
```

Figure 6-6-1 Initialization of ASK_CTL library

Most basic path quantification operators are:

- EXIST_UNTIL (ϕ, φ) – is true if there exists a path, starting from a current position in a state space, that ϕ is true for each state along the path until the last state of the path where φ must hold
- FORALL_UNTIL (ϕ, φ) – is true if for all paths, starting from a current position in a state space, that ϕ is true for each state along the paths until the last state of the paths where φ must hold

It can be noticed, that four new convenient operators can be derived [DAM] from two path quantification operators by replacing their first arguments with true value (TT):

- POS (φ) \equiv EXIST_UNTIL (TT, φ) – is true if it is possible to reach a state where φ is true (reachability)
- INV (φ) \equiv NOT(POS(NOT(φ))) – is true if φ is true in all states (invariant)
- EV (φ) \equiv FORALL_UNTIL (TT, φ) – is true if φ becomes true eventually, in a finite number of steps (eventual)
- ALONG (φ) \equiv NOT(EV(NOT(φ))) – is true if there exists a path where φ is true for every state; the path is either infinite or ends in a dead state

A model checker verifies all four operators from the current place in a state space.

Transitions model operators are analogous to state operators but with transition arguments instead. Despite the fact that there are some more operators available, we have chosen only those as relevant to our verification purposes. A reader can study details of ASK_CTL in the manual [DAM].

Next, a few methodologies are presented to show approaches on how to design complete models with rather different priorities. The first methodology focuses on the logic from the perspective of one party whereas the second pursues a top-down design with all parties. Both methodologies are analyzed with respect to their clarity and verification suitability. However, for the reasons explained later, extensive model testing for both methodologies is presented in chapter 8.6.

6.1. Goal sequences

Goal sequences based on UML 2.0 collaborations [UML2.0] is an interesting approach presented in research [UUCC, CASS, and FCGS]. A high-level methodology described there can be used to specify the requirements even before the formal design process. Additionally, it can lead to detecting implied scenarios – scenarios that may be present in the service implementation even though they have not been defined in the specification.

The main features of proposed collaboration-based design are:

- provision of a high level and abstract view on interacting parties while preserving precision
- specification of a service as a collaboration which can be combined into even larger services compositionally and incrementally
- description of partial functionalities involving interactions among participating roles played by objects [CASS]
- interaction diagrams focus on service goals [MPSG] and interfaces, instead of elements to abstract away implementation details

Nevertheless, as mentioned in [UUCC], the technique cannot be used for describing goal sequence dependencies in overlapping collaborations. It is also proposed [CASS] that protocol state machines, traditionally used to describe interactions, should be replaced by port state machines and should be included in interface description.

6.1.1. Methodology

Goal sequence methodology (GSM) has a consistent 5-step process [CASS] that “takes the designer gradually from a more abstract specification to a more detailed one” producing a goal sequence diagram similar to a UML activity diagram. In order to evaluate the methodology, five steps are pursued together with Petri net realization presented in [FCGS].

Step 1. Identify the main roles of the service under specification

For the sake of brevity and figure clarity, following roles are sometimes abbreviated (in brackets).

- Vehicle (V) – representing logics in the car
- Central (C) – arranges all services
- Bank (B) – handles card payment
- Renting Agency (R) – arranges a replacement car
- Garage Agency (G) – arranges car repairs
- Towing Agency (T) – arranges delivery of car to garage
- Taxi Agency (X) – arranges delivery of taxi

There is also a driver role that provides a credit card but it can be omitted since it is the only human action that initiates the process.

Step 2. Define the collaborations that each service role has with any of the other roles.

Here, isolated service behaviors are identified by defining collaboration for mutual and logically consistent interactions between roles. For the sake of simplicity, only two-party collaborations are considered. Multi-party collaborations can be decomposed into multiple two-party collaborations. Additionally, collaborations should have identifiable goals that can be defined in predicates formulated for properties of collaboration.

Collaborations between services:

- between Vehicle and Central
 - Request Services (RS)– car forwards details of the failure together with its location and credit card
- between Central and Bank
 - Authorize Payment (AP)– Central asks a Bank to validate the deposit payment
 - Cancel Payment (CP)– Central requests cancellation of deposit payment
- between Central and Renting Agency
 - Request Renting (RR)– Central requests a replacement car
- between Central and Garage
 - Request Garage (RG)– Central reserves a garage
 - Cancel Garage (CG)– Central requests cancellation of garage reservation
 - Garage Confirm (GC)– Central confirms a previously reserved garage
- between Central and Towing Agency
 - Request Towing (RT)– Central reserves a towing car to vehicle’s location
- between Central and Taxi Agency

- Request Taxi (RX)– Central reserves a taxi to car’s location

Step 3. Specify the service as a single composite collaboration

This step allows producing an overview of the information from steps 1 and 2. Figure 6-2 shows collaboration diagram representing structure of the service decomposed into smaller sub-collaborations which are bound to specific roles. Even without interaction details, a typical client-server architecture is visible.

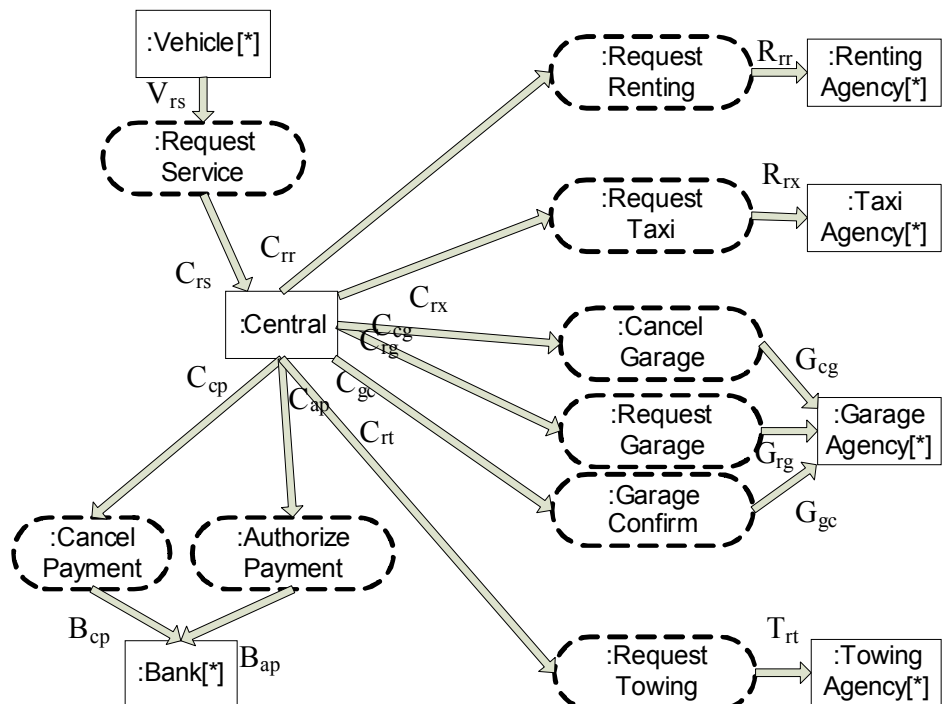


Figure 6-2 Request services process composite collaboration with taxi service

Step 4. Describe dependencies by means of a collaboration goal sequence

Collaboration goal sequences depicted in Figure 6-3 describe how process logic is attached to collaborations, showing their order of execution and dependencies. The diagram is a minor notation enhancement [CASS] to UML activity diagram. It shows a collaboration use that is active for a consequent state. A passing token represents the flow of control among activities as in activity diagram. Collaboration becomes active when it receives a token, either from an initial or suspended state. A suspended collaboration then should resume its suspended behavior (marked by an empty circle in entry point). After releasing a token, a collaboration use can either end or be suspended to wait for appropriate inputs. Suspended collaboration uses have exit points marked with an empty circle in comparison to crossed-circle for terminating activities.

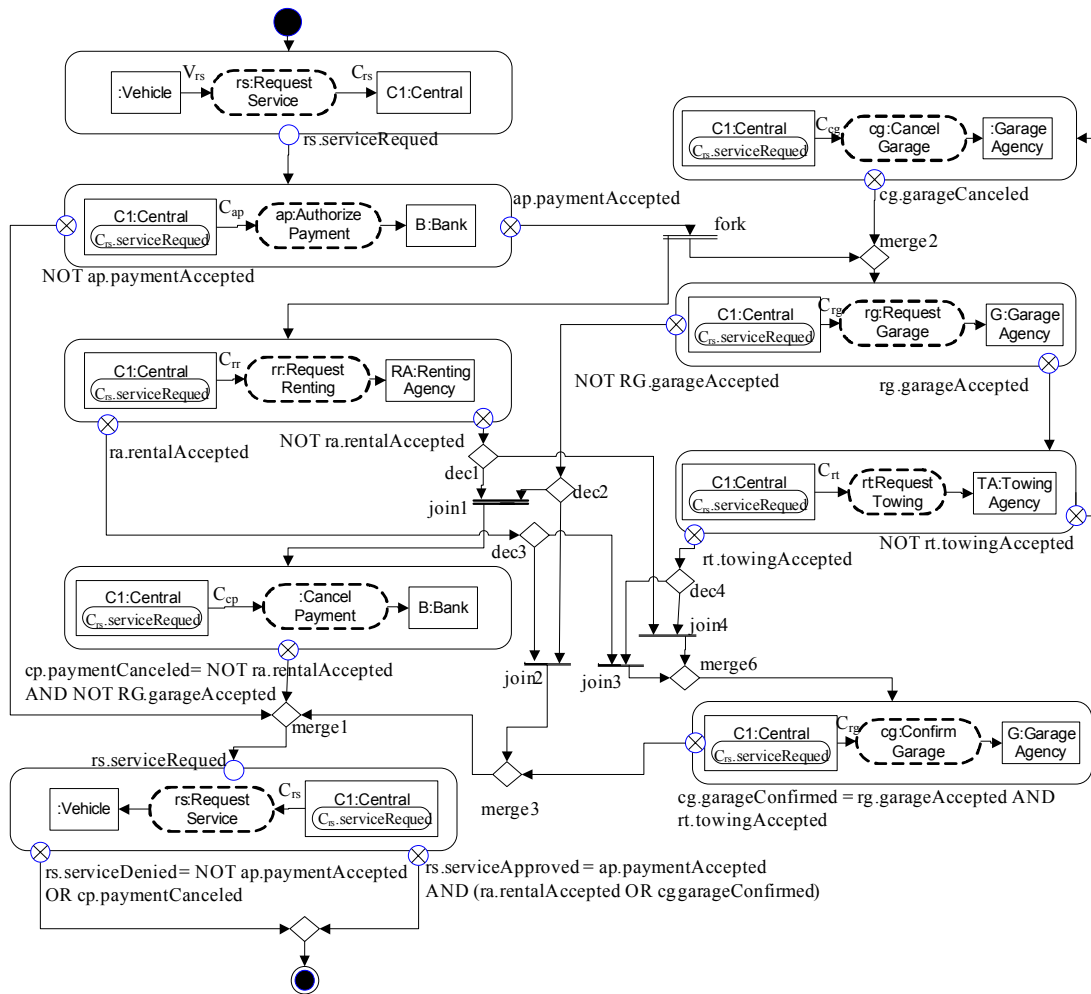


Figure 6-3 Goal sequence without a taxi service

Addition of a taxi service is not as straightforward as in models before and changes the diagram as shown in Figure 6-4. However, process logic is quite simple and it is believed that, in case of more complex connections, the ‘spaghetti’ of nodes and arcs connections could make a diagram unreadable. One possible solution on how to simplify the diagram would be to allow multiple arcs from activity’s exit point. For the sake of clarity elements like: fork, join, merge and dec (decision) have been enriched with names to allow a clear comparison with a Petri net.

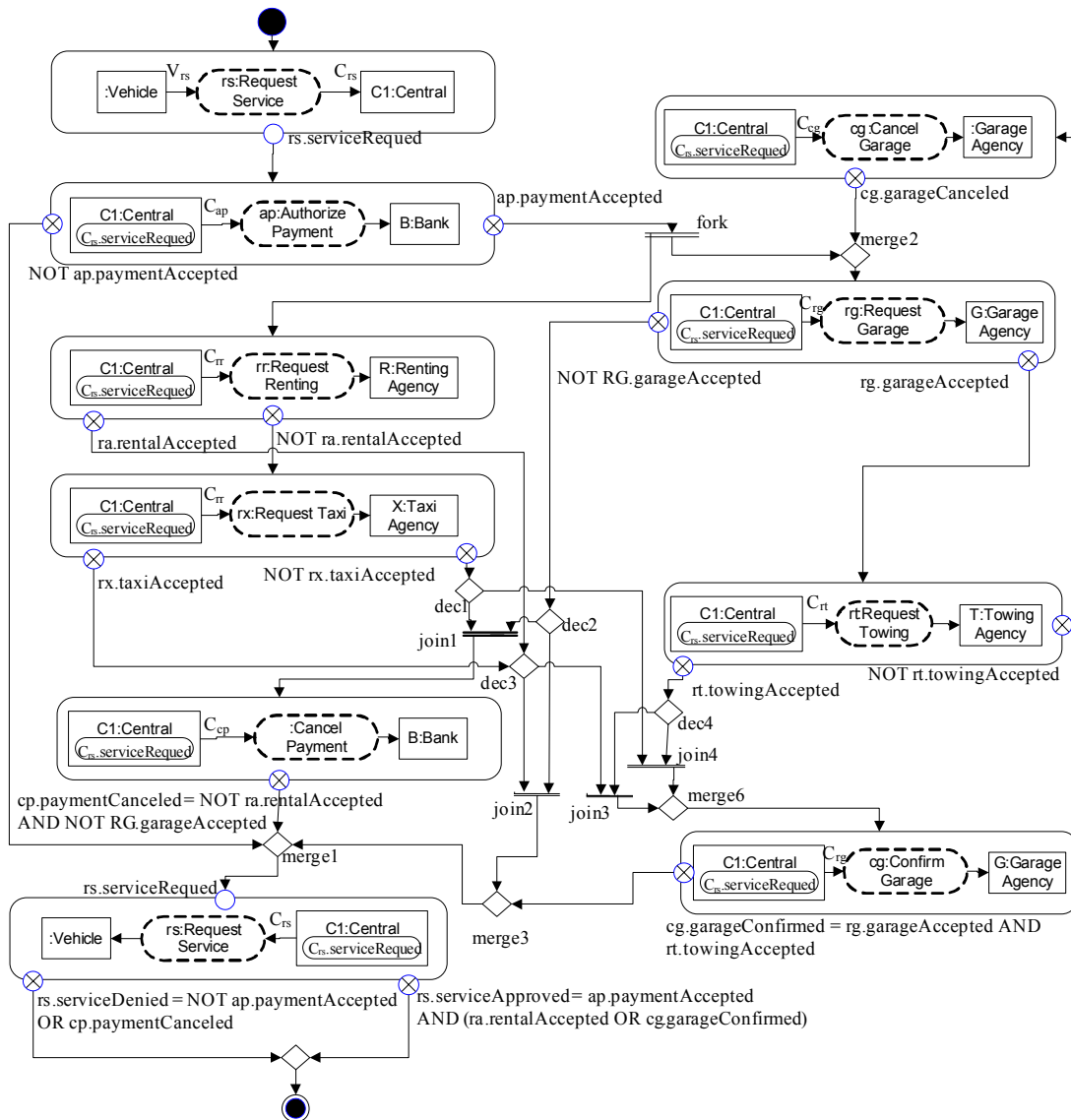


Figure 6-4 Goal sequence with a taxi service

Even during a simple diagram creation, many functional inconsistencies have been found and corrected. Consequent search for implied scenarios, which requires investigating roles that different components play, pointed out even more problems. Finding and correcting them even before a real design stage is obviously an advantage. They would probably be found anyway in later phases, but with significantly bigger cost, since design is usually more system and language specific and by requiring more details, it loses a big picture of a system. Similarly, cost of finding and removing a bug after implementation increases dramatically. It is believed that finding an error as early as possible is always worth the effort.

Step 5. Detailing the behavior of the collaborations

This step allows defining detailed communication between roles in a collaboration. Figure 6-5 shows eight different sub-collaboration interactions. For the sake of brevity, sub-collaborations that end in more than one way (Request Garage, Request Towing, Request Renting, Request Taxi, and Authorize Payment) are placed on the same sequence diagrams,

marking possible variations with [variation1/variation2]. Collaborations that require choosing a Web service from a UDDI registry (Request Garage, Request Towing, Request Renting, and Request Taxi) have a loop that is interrupted when no other services match the required constraints or when a chosen service approves a request.

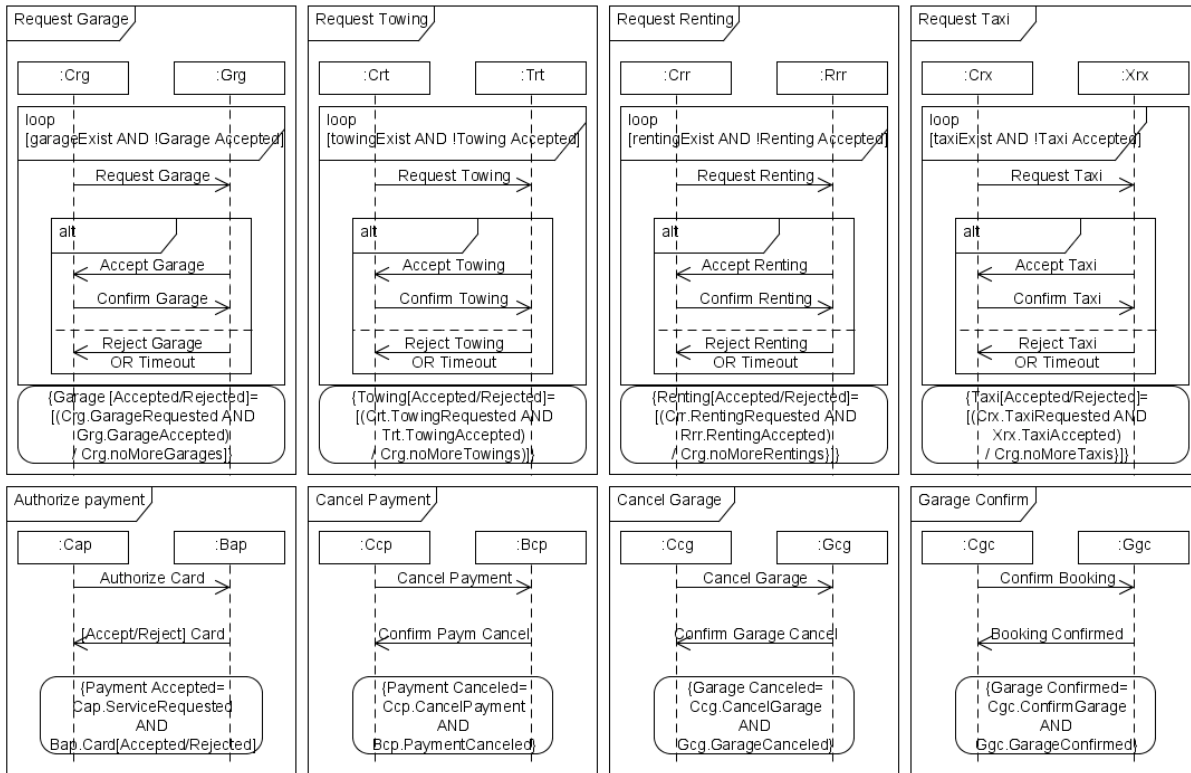


Figure 6-5 Detailed interactions for the sub-collaborations

At this point the design is finished allowing to check the model for implicit scenarios.

6.1.2. Implicit scenario detection

Implicit scenarios detection reveals system behavior that has not been expected, but may still emerge according to the process logic. Those undesired behaviors are related usually with concurrency between several parties that simply try to follow their own protocols. Implicit scenarios are detected by analyzing sub-role sequences that service role executes as a part of the service [CASS]. Sub-role sequence of a specific role can be extracted by “traversing each possible path of the goal sequence looking for occurrences of the same role” [CASS]. Next, the sub-role sequences are studied for possible inconsistencies and/or non-determinism.

Despite concurrent activities and the main node (Central), which plays several roles (Crs, Cap, Crr, Crg, Crt, Ccr, Ccg, Cgc, Ccp), the model is relatively simple. It is due to its client-server architecture where interactions are much more organized than in peer-to-peer networks. This is the main reason why both implicit scenario analyses [CASS]: isolated and interacting show no conflicts apart from a possible loop during the negotiation of a garage and towing services. Table 6-1 shows order of roles and concurrent activities that can happen

from the Central's point of view. Loops that are present in the third and fourth column can cause some unpredictable behavior when replies from out-dated and invalid interactions merge with current requests.

Every successful garage reservation has to be accompanied by a booked towing service. An incorrect behavior could appear when after a successful garage reservation the first (or more) towing service would not respond in an acceptable time. In this case a garage is cancelled to reserve a new one that hopefully matches some towing services. However, after a new towing service is requested, the old, outdated responses may arrive, causing a Central to incorrectly align a fresh request with timed-out reply. This problem should be resolved either by pre-reserving a service and then confirming (or canceling) the success using correlations (colors of tokens) or by making a permanent reservation that may be canceled. Since the possible running time of operation is not long, the former approach is preferred. In order to prevent different communications to interact, there also has to be a notion of session whenever a new service is requested from the Central. This is definitely a valuable note for using a correlation in later design phases.

Table 1. Sub-role sequences for the Central sub-role (without taxi)

| | | | | | |
|------------------------|-------------------------|---|---|--|---|
| Payment not authorized | Everything is available | Towing impossible for some garage(s), replacement car is available, | Towing impossible for some garage(s), replacement car is not available, | No garage nor replacement is available | Replacement car available, but no garage is available |
| Crs | Crs | Crs | Crs | Crs | Crs |
| Cap | Cap | Cap | Cap | Cap | Cap |
| Crs | Crg Crr | Crg Crr | Crg Crr | Crg Crr | Crg Crr |
| | Crt | Crt | Crt | Crt | Crt |
| | Ccg | loop:// Ccg | loop:// Ccg | Ccp | Crs |
| | Crs | // Crg | // Crg | Crs | |
| | | // Crt | // Crt | | |
| | | Ccg | Ccg | | |
| | | Crs | Crs | | |

6.1.3. Petri net realization

In order to automatically verify a model, it has been adapted to the CPN Tools, using rules from [FCGC] depicted in Figure 6-6.

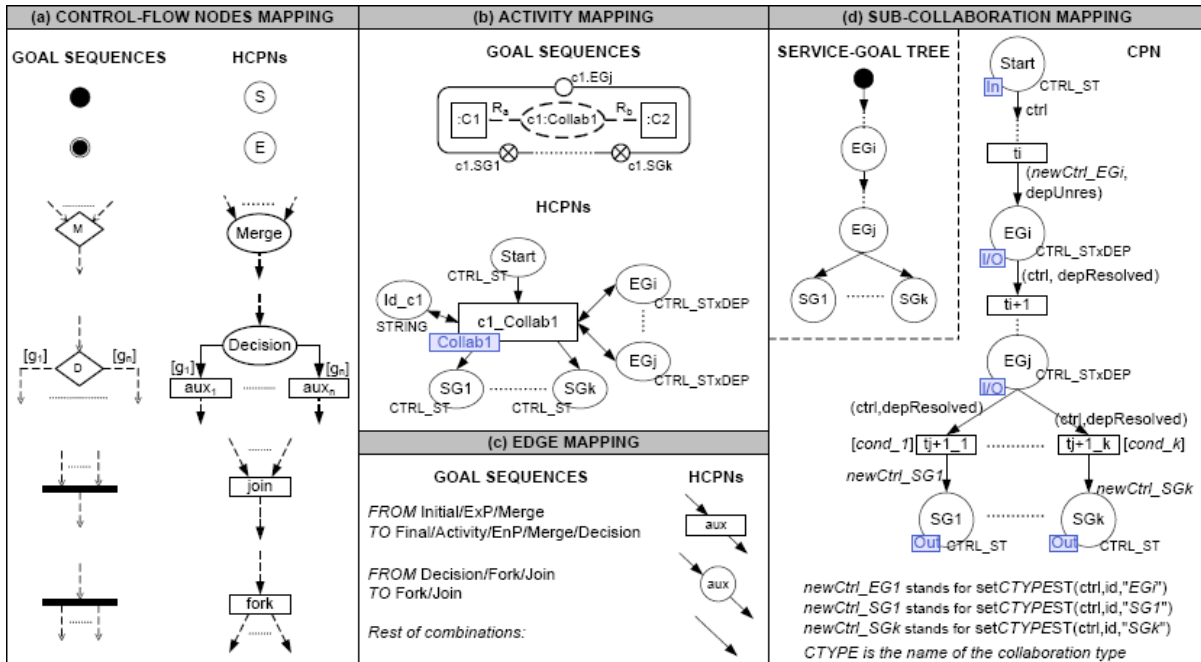


Figure 6-6 Mapping of goal sequence elements to HCPN [FCGC]

After following a transformation scheme, model's highest abstraction is depicted in Figure 6-8, while declarations together with page overview are depicted in Figure 6-7.

```

▼ Declarations
  ► Standard declarations
  ▼ val MAXGAR=3;
  ▼ colset CTRL_ST=product INT*INT*STRING;
  ▼ var ctrl:CTRL_ST;
  ▼ colset DEP =STRING;
  ▼ var d:DEP;
  ▼ colset CTRLxDEP=product CTRL_ST*DEP;
  ▼ var cXd:CTRLxDEP;
  ▼ var id:INT;
  ▼ var id1:INT;
  ▼ var id2:INT;
  ▼ var s:STRING;
  ► Monitors
  ▼ goalSequence
    reqServ
    authPaym
    reqRent
    reqGar
    reqTow
    reqTaxi
    cancGar
    cancPaym
    confBook
  Bank
  Garage
  Towing
  Taxi
  Renting

```

Figure 6-7 Declarations together with page overview

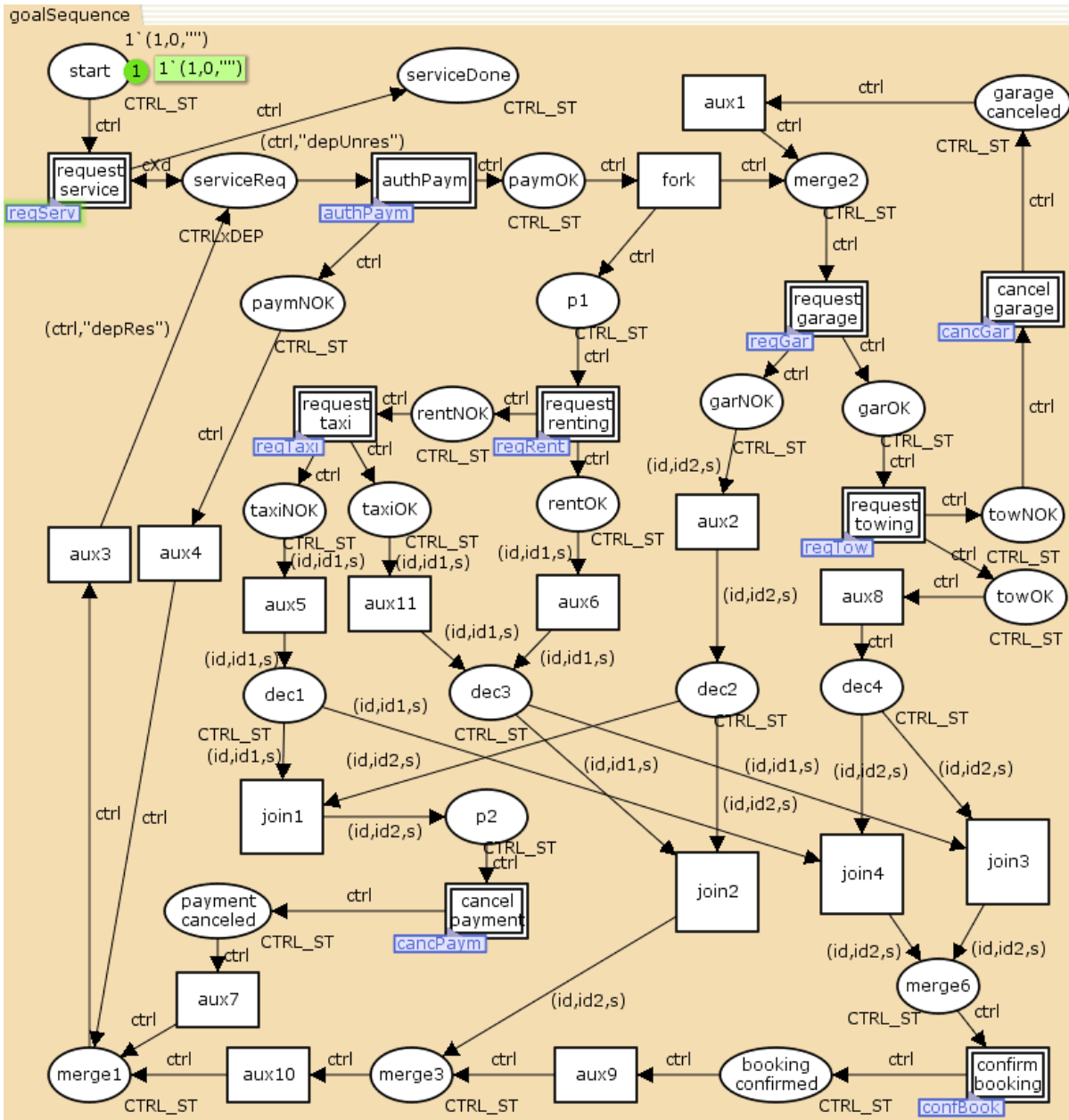


Figure 6-8 Collaboration goal sequence as Petri net

A token starts in the top left corner with a dependency mechanism explained in the next subsection. Token contains data of the request number (first place: 1), currently chosen garage number (second place: 0) and failure details (third place: 0). For the sake of brevity, failure details are not exposed and the choice of what data to send and to whom should be analyzed against some service policies in the implementation stage. One of the policies could be that, apart from the bank, no other services receive customer’s credit card data.

A token travels to sub-page “authPaym” and goes either to places “paymNOK” (payment Not OK) and merge1 (in case a credit card is rejected) or to “paymOK” (payment OK) and splits to two in a transition “fork”. From this moment, there are two parallel behaviors. The “left” token rents a car and if it is not possible – orders a taxi instead. It has to be noted, that more services may be requested in each sub-page, as explained further. The “right” token

reserves a garage and only if that succeeds – orders towing. In case a towing is rejected, previously chosen garage needs to be cancelled hoping that a new garage will match some towing.

The list of four join transitions ensures that the procedure ends with proper actions. In case neither taxi nor garages are available – transition “join1” is fired and the previously reserved deposit payment is canceled. In case towing (and so garage also) is available either join3 or join4 transitions are triggered leading to confirmation of the previously chosen garage. One can notice a pattern matching mechanism in those four join transitions that matches request identifier to distinguish between concurrently handled requests. In any case – process reaches place “merge1”, after which the status of the request is updated to “depRes” (dependency resolved) and finally, through sub-page “request service” ends in place “serviceDone”.

Comparing to the UML-like goal sequence depicted in Figure 6-4, Petri net depicted in Figure 6-8 seems more complicated having not only the nine activities (represented by sub-pages), but also many other nodes. The reason for that is that sub-collaborations need to have explicitly defined input and output points as places, as well as the additional transitions connecting decision and merge elements. Despite its complexity, simulation mechanism can quickly visually acquaint a designer with the behavior of the model. Another difference is that even though the first activity (“request service”) appears twice in UML-like diagram, it is represented by one sub-page in the Petri net. This modification is justified by the fact that the former two instances are functionally equivalent to the latter sub-page with an input/output place, as proposed in [FCGC]. A green color in a sub-page in the top left corner marks that there is an active transition inside.

Dependency mechanism

The whole procedure starts with a dependency mechanism in Figure 6-9 that routes resolved and unresolved requests. The technique is proposed by [FCGS] – a slightly more detailed elaboration of [CASS]. A vehicle sends a request which acquires a status “depUnres” and cannot proceed until the status is changed to “depRes”. The main logic on a page “goalSequence” is notified by an input/output place “serviceReq”. Status “depRes” is assigned after a token passes transition “aux3”. This mechanism ensures that a token does not circle unnecessarily around the net. Green color defines an active transition (“t1” in this case).

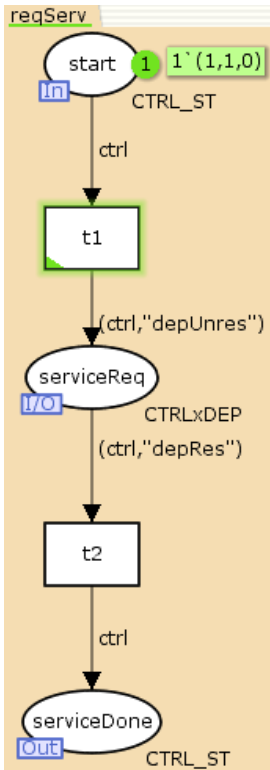


Figure 6-9 Dependency mechanism

Communication collaborations

Since this is the close-to-activity-diagram highest abstraction level it is necessary to specify lower level sub-pages. Neither [CASS] nor [FCGS] explain how to represent interactions described in step 5 of the methodology. Because the highest abstraction is the process itself, communication can be represented by fusion places, as shown in Figure 6-10.

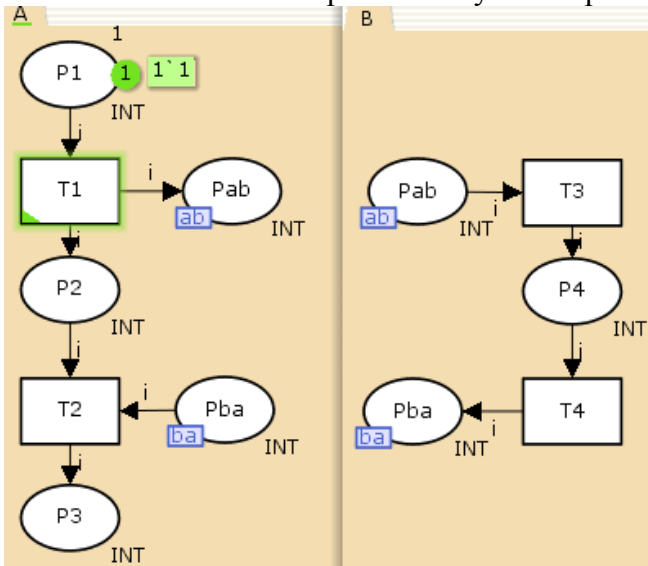


Figure 6-10 Communication through fusion places

After launching fork transition “T1”, process “A” creates a token in place “Pab”, which is fused with a place “Pab” in process “B”. After triggering transitions “T3” and “T4”, the

token returns to process “A” combined by a join transition with a token waiting in location “P2”. This technique allows modeling both synchronous and asynchronous communication in a one-to-one or one-to-many scheme. In addition, join transition “T2” allows matching arriving tokens for the expected data.

1. Bank interaction

Two of Central’s activities involve communication with bank:

- Authorize payment depicted in Figure 6-11
- Cancel payment depicted in Figure 6-12

In addition, Bank’s logic is shown in Figure 6-13.

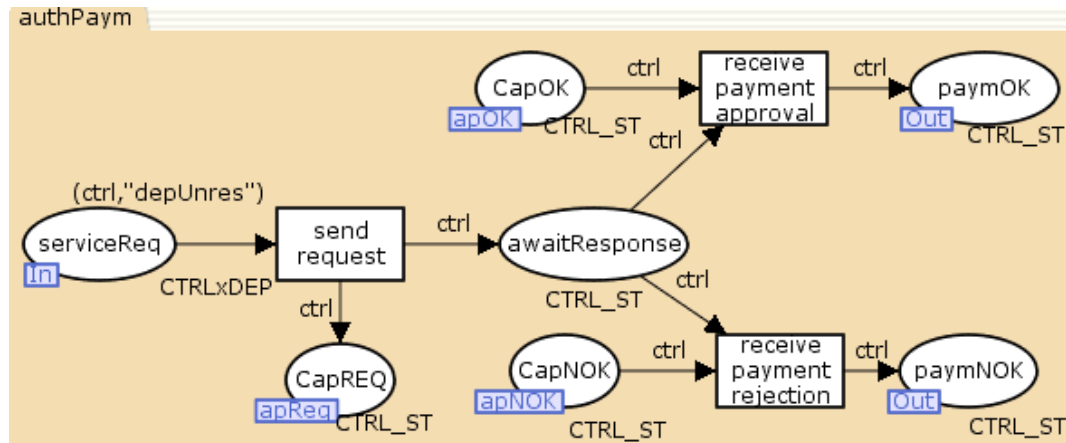


Figure 6-11 Authorize Payment interaction

For the sake of visual clarity acronyms have been used to describe interface names and communication channels. Interface name begins with a capital letter that identifies a party (C-Central, B-Bank, etc.), followed by two letters identifying interaction name (ap – Authorize Payment, cp – Cancel Payment, etc.) and capital letters identifying whether it is a request (REQ), positive/negative reply (OK/NOK) or reply confirmation (CONF). Additionally, fused places are tagged with an acronym of an interaction name (ap – Authorize Payment), followed by interaction nature (Req, OK, NOK, Conf). Needless to say, acronyms should be unique; otherwise CPN Tools will mark them as errors.

After an authorization request is sent by a fused place tagged with “apReq” (authorize payment request), Central’s process waits for a token either from “CapOK” tagged by apOK (authorize payment OK) or “CapNOK” tagged by “apOK” (authorize payment Not OK). After a response is received, the sub-page places a token in either “paymOK” or “paymNOK” outputs.

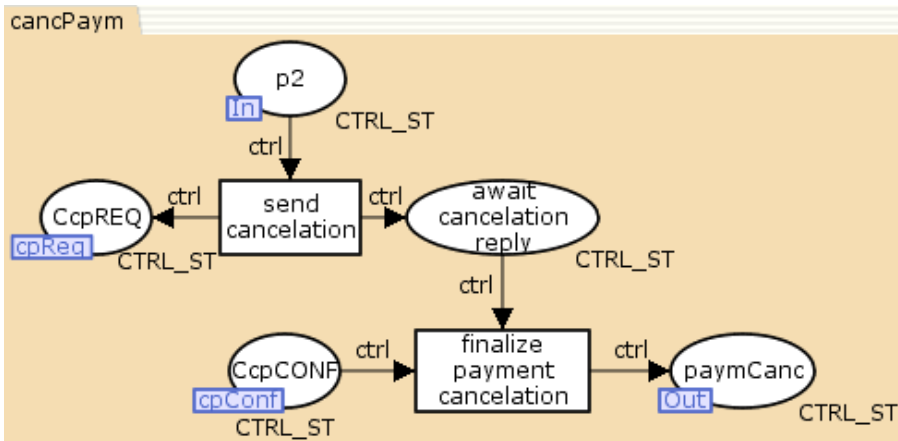


Figure 6-12 Cancel Payment interaction

Canceling payment is a simpler interaction than authorization in a way, that there can only be one reply from the bank – canceling confirmation that arrives to place “CcpCONF” tagged by “cpConf” (cancel payment Confirmation).

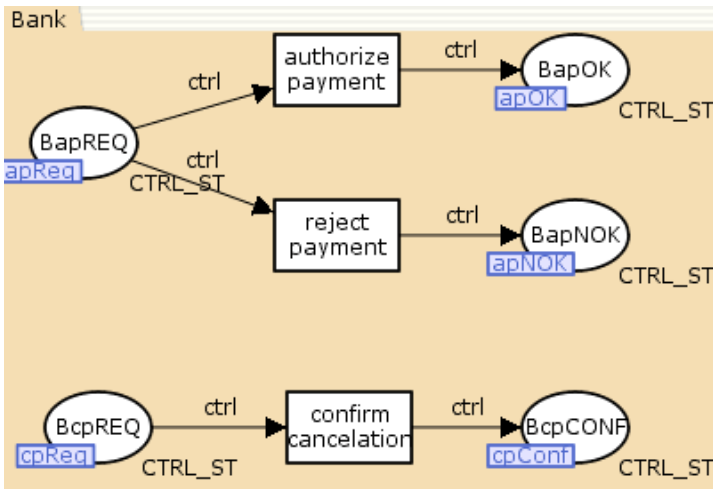


Figure 6-13 Bank's logic for authorize payment and cancel payment interactions

Because the model is analyzed from the perspective of Central's orchestration process, logic of a bank is reduced only to its provided methods, together with possible replies. Contrary to the reserve-and-confirm approach in the next paragraph, payment cancellation is simply a method that expects a confirmation. This is an example of a design by contract, where preconditions specify that the cancelled payment has been previously reserved and postconditions ensuring that it will be canceled. How that is performed, in bank's process (transaction atomicity, database operations, etc.) is irrelevant from Central's perspective.

2. Renting, Taxi and Towing Agencies

Collaborations: Request Renting, Request Taxi and Request Towing follow the same communication pattern. For the sake of brevity, only Request Towing interaction is depicted in Figure 6-14 and the towing agency logic is shown in Figure 6-15.

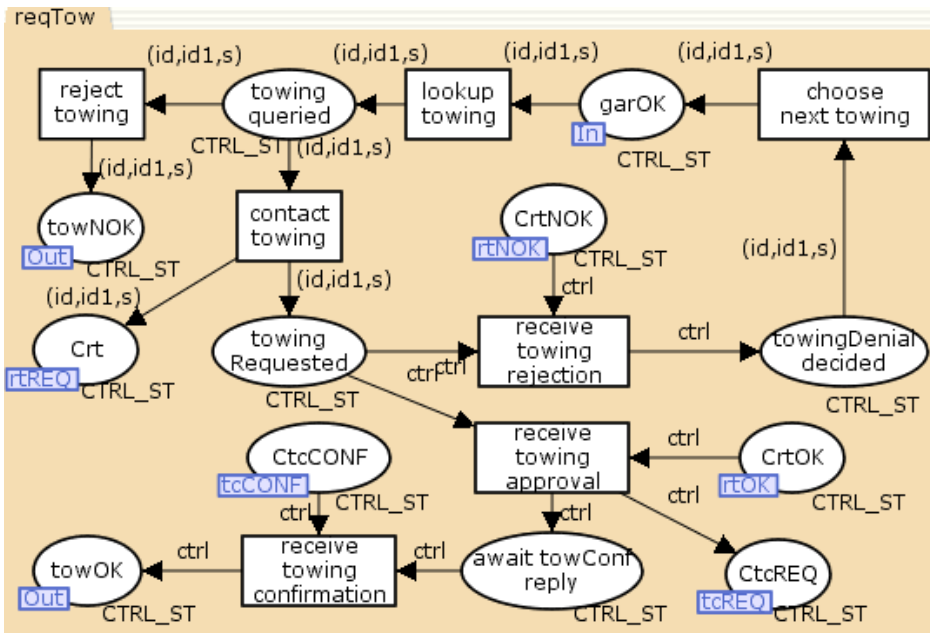


Figure 6-14 Request towing interaction

Towing may be reserved only after successful reservation of a garage. Thus, a token appears from place “garOK” and after querying UDDI registry for suitable services (transition “lookup towing”) it moves to a place ”towing queried”. Here, there is a non-deterministic decision whether there are suitable services at all. Value “s” is used to store specific request data (failure details, car position, etc.) that can be processed by external service.

In case the towing request is accepted, Central sends a request to the chosen service by placing token in place “Crt” tagged by “reREQ” (reserve towing Request) and waits in place “towing Requested” for either approval or denial. If the request is rejected, the loop goes back to UDDI registry lookup to find a new service. If towing is accepted, Central sends a confirmation and only after a confirmation reply – the reservation is approved.

The confirmation is a mandatory mechanism that ensures that only one service is reserved. Many services may be requested, but since neither servers nor communication are reliable, there has to be a mechanism that will reject the pending request, after a certain timeout, to ask another service. Since some services may try to respond after their timeout expires, a suitable correlation mechanism is necessary. Correlation needs to take under consideration:

- reference to Central (IP address, UDDI identifier, etc) to identify return point
- communication channel to current service with its identifier to precisely find end point services
- timestamp that justifies and proves scraping of outdated messages
- failure request details

Having a current status of procedure (time, current service) the system should discard all invalid messages and, if necessary, retransmit messages or generate timeouts on awaiting ports. The precise behavior depends on real circumstances and is a part of the implementation phase.

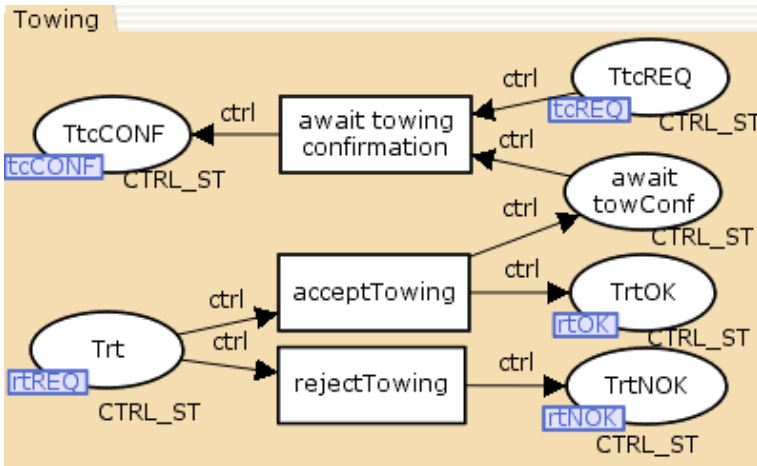


Figure 6-15 Towing agency logic

Complementing the other side of the protocol, a towing agency can either reject a request, or accept it, waiting for the confirmation to finally book the place. As an example of two-phase commit protocol, there is a risk of blocking a resource when Central does not reply, but similarly as before, possible solutions are to be decided in the implementation stage.

3. Garage interaction

Interaction “Request Garage” is depicted in Figure 6-16 and follows a similar pattern as “Request Towing” from Central’s perspective.

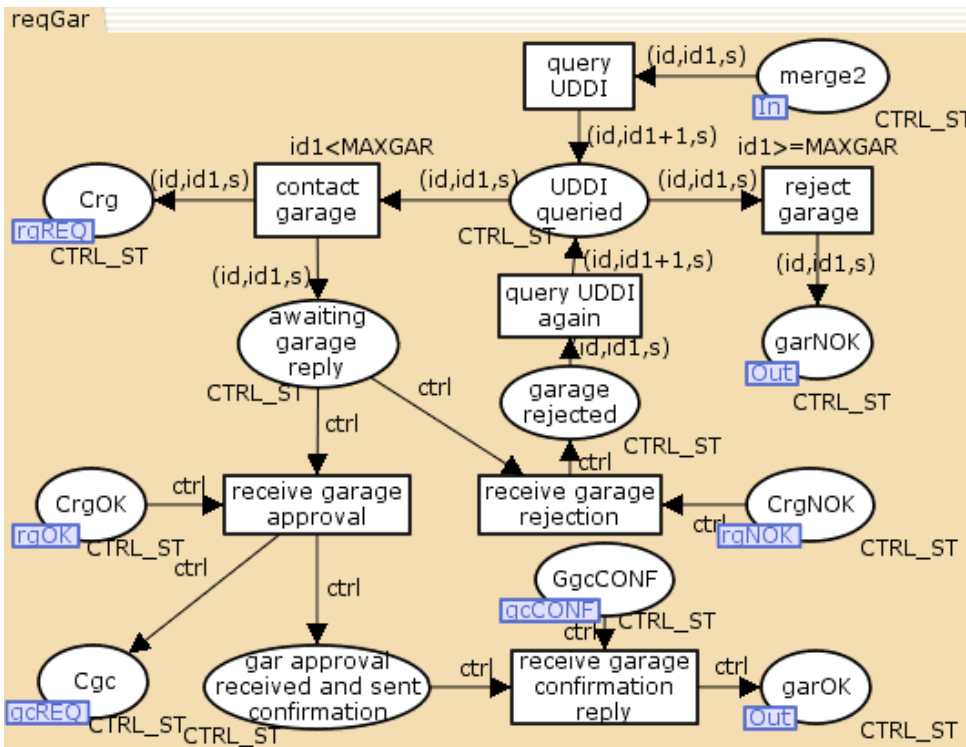


Figure 6-16 Request garage interaction

The additional guard on the “contact garage” transition ($id1 < MAXGAR$) limits the number of possible services in the UDDI registry and state space at the same time. Value 3 for the constant “MAXGAR” has been chosen to cover all possible system behaviors:

- both garage and towing are approved in the first iteration
- because towing is not available, an already approved garage is canceled and a new garage matches a towing
- no garages match any towing services

Every new garage rejection increases the counter in the token as shown in Figure 6-17.



Figure 6-17 Request garage interaction

Garage logic depicted in Figure 6-18 is significantly more complex because of all possible scenarios that a protocol has to manage. The protocol defines that a garage also needs to take under consideration collaborations: Cancel Garage (depicted in Figure 6-19) and Confirm Garage (depicted in Figure 6-20).

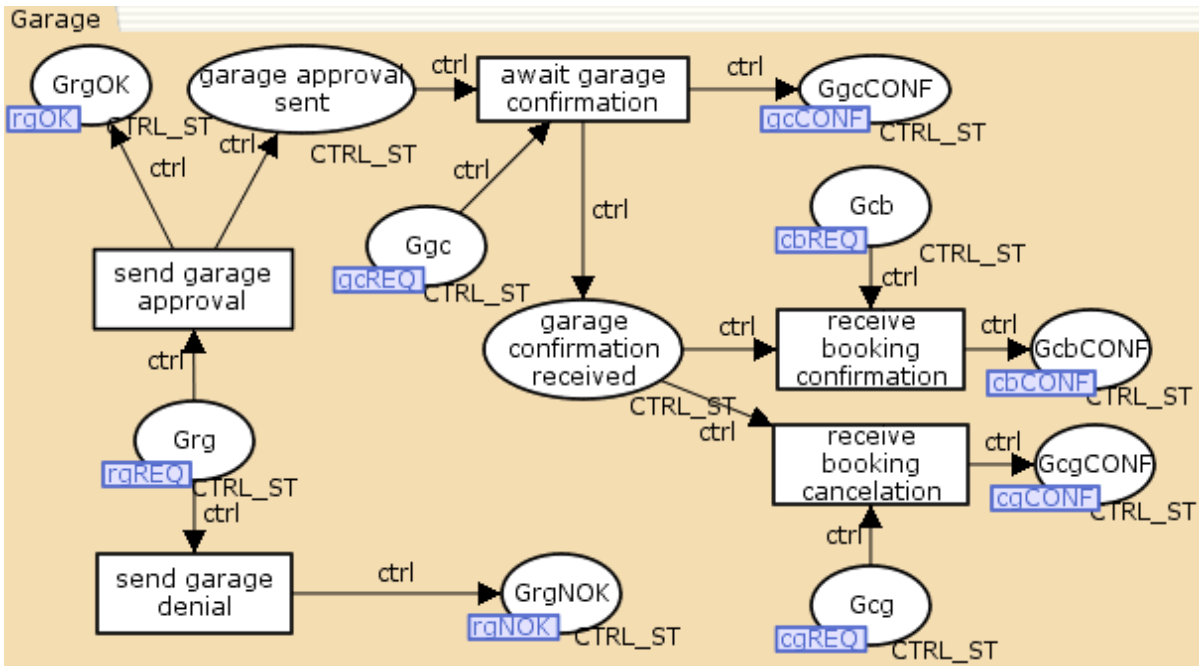


Figure 6-18 Garage logic

The protocol begins in place “Grg” tagged with id “rgREQ” (reserve garage request). According to some local logic, a decision is made whether to reject the request by putting a token in place “GrgNOK”, or to approve it waiting for confirmation. In this way only one garage is reserved, but since it may be canceled depending on the towing query, it has to wait with finalizing process until either booking confirmation or cancellation arrives.

Garage appointment is canceled if there are no towing agencies that can tow a vehicle there and the protocol is depicted in Figure 6-19.

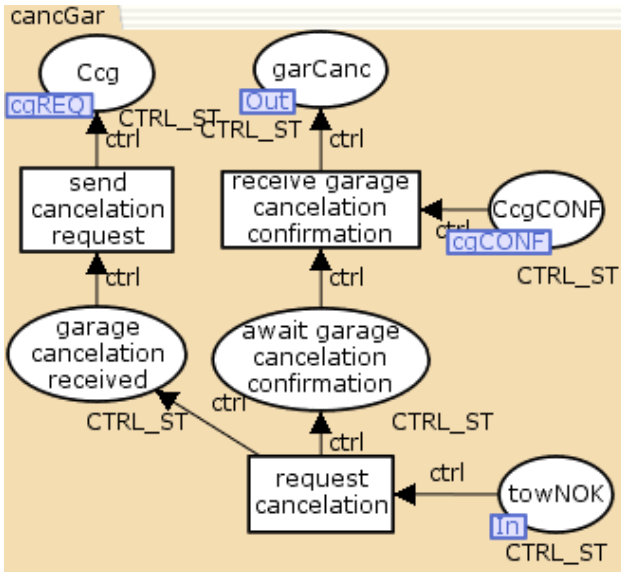


Figure 6-19 Cancel Garage interaction

After a successful reservation of towing, garage appointment can be finally booked. A pretty straightforward Petri net is depicted in Figure 6-20.

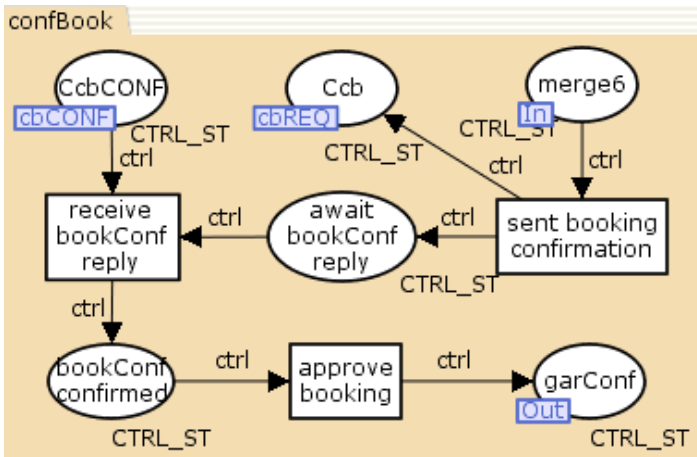


Figure 6-20 Confirm booking interaction

6.1.4. Verification

The most prominent feature of having all elements bound into one system is the possibility of verifying it as a whole. CPN Tools offers a wide range of tools for either general or detailed model verification. General reports consist of statistics regarding a state space (number of nodes, arcs, dead markings, token occurrence in places), whereas specific formulas can check detailed properties by predefined ML queries or ASK_CTL logic. The full state space report is in appendix Appendix F – State space reports.

State space analysis

State space analysis reveals a difference between the amount of nodes in plain state space and Scc (strongly connected components of the state space) graph.

```

State Space
Nodes: 1086
Arcs: 2302
Secs: 2
Status: Full

```

```

Scc Graph
Nodes: 582
Arcs: 1592
Secs: 0

```

This difference means that model's occurrence graph contains loops that can be optimized in a Scc. Those loops are located in sub-pages "reqTow", "reqRent" and "reqTaxi", (allowing to query more than one service) and also in the cancel garage loop in "goalSequence" page.

Boundedness properties do not show any problems such as never used places or unnecessarily multiplied tokens. All states are visited by at most one token (for one input token). Since the process is finite, there are no home markings that the token can reach from any place. Liveness properties reveal 4 dead markings (nodes 17, 1046, 1082, 1086) from where the token cannot proceed any further. Here are the possible final points where token in the last place, "serviceDone", equals:

- (1,0,"") – when the credit card is rejected
- (1,1,"") – after the first garage is accepted by the towing agency
- (1,2,"") – when the first garage is rejected, but the second is accepted
- (1,3,"") – when the second garage is rejected by the towing agency and thus, the garage is rejected too

There are also no other potentially problematic elements such as dead transitions (unused transitions that never trigger) or live transitions (transitions that can be reached from every place).

Further analysis of the behavior requires using ML expressions

Apart from the state space report, CPN Tools provides many useful queries [CTSSM] to inspect the state space. One of them (SearchNodes) allows traversing the state space, giving very flexible choice for searching conditions and return arguments. With a query in Figure 6-21, we can use it to find all dead markings:

```

val deadMarks = SearchNodes (
EntireGraph,
fn n => (length(OutArcs(n)) = 0),
NoLimit,
fn n => n,
op ::)
val deadMarks = [17,1086,1082,1046] : Node list

```

Figure 6-21 Function to search all states to find dead markings

Where the arguments denote:

- EntireGraph – search area,

- $fn\ n \Rightarrow (length(OutArcs(n)) = 0)$ – verification condition (only dead markings have no outgoing arcs in the state space),
- NoLimit – maximum number of nodes to find (search limit),
- $fn\ n \Rightarrow n$ – evaluation function executed on nodes fulfilling the verification condition,
- [],
- `op ::`

The same result can also be displayed using one of the automated predefined query shown in Figure 6-22

```
ListDeadMarkings (); val it = [17,1086,1082,1046] : Node list
```

Figure 6-22 Predefined query to find dead markings

One can also find a path between nodes with a query in Figure 6-23.

```
Reachable' (InitNode,17); A path from node 1 to node 17 is: [1, 2, 3, 5, 7, 9, 12, 17]
val it = true : bool
```

Figure 6-23 Query to find and display a path between nodes

Instead of creating an occurrence graph to make sure that all dead markings are only those with token(s) in the last state (“serviceDone”), one can display them (or only the first one for brevity) with a function print shown in Figure 6-24.

```
fun printAll (h::t) = print (NodeDescriptor h);
printAll(ListDeadMarkings ());
```

Figure 6-24 Function displaying a precise description of a first dead marking

The result depicted in Figure 6-25 contains a precise description of the marking with a state of every place:


```

val printAll = fn : Node list -> unit
998:
cancPaym'p2 1: empty
cancPaym'paymCanc 1: empty
cancPaym'CcpREQ 1: empty
cancPaym'CcpCONF 1: empty
cancPaym'awaitResponse2 1: empty
cancPaym'toSend2 1: empty
cancPaym'canceled 1: empty
cancGar'towNOK 1: empty
cancGar'garCanc 1: empty
cancGar'garCanc_requested 1: empty
cancGar'awaitConf 1: empty
cancGar'CcgCONF 1: empty
cancGar'Ccg 1: empty
Garage'GrgOK 1: empty
Garage'Grg 1: empty
Garage'awaitGarConf 1: empty
Garage'Ggc 1: empty
Garage'GrgNOK 1: empty
Garage'garConf_received 1: empty
Garage'GgcCONF 1: empty
Garage'GcbCONF 1: empty
Garage'GcgCONF 1: empty
Garage'Gcg 1: empty
Garage'Gcb 1: empty
Bank'BapREQ 1: empty
Bank'CapOK 1: empty
Bank'CapNOK 1: empty
Bank'BcpREQ 1: empty
Bank'BcpCONF 1: empty
confBook'merge6 1: empty
confBook'garConf 1: empty
confBook'CcbCONF 1: empty
confBook'Ccb 1: empty
confBook'bookConf_confirmed 1: empty
confBook'await_bookConf_reply 1: empty
goalSequence'start 1: empty
goalSequence'serviceReq 1: empty
goalSequence'serviceDone 1: 1` (1,3,0)
goalSequence'paymOK 1: empty
goalSequence'merge2 1: empty
goalSequence'garCanc 1: empty
goalSequence'garOK 1: empty
goalSequence'garNOK 1: empty
goalSequence'towNOK 1: empty
goalSequence'towOK 1: empty
goalSequence'garConf 1: empty
goalSequence'p2 1: empty
goalSequence'rentNOK 1: empty
goalSequence'rentOK 1: empty
goalSequence'paymCanc 1: empty
goalSequence'merge1 1: empty
goalSequence'merge3 1: empty
goalSequence'p1 1: empty
goalSequence'paymNOK 1: empty
goalSequence'merge6 1: empty
goalSequence'dec1 1: empty
goalSequence'dec4 1: empty
goalSequence'dec3 1: empty
goalSequence'dec2 1: empty
goalSequence'taxiNOK 1: empty
goalSequence'taxiOK 1: empty
reqServ'serviceDone 1: 1` (1,3,0)
reqServ'serviceReq 1: empty
reqServ'start 1: empty
Towing'Trt 1: empty
Towing'TrtNOK 1: empty
Towing'TrtOK 1: empty
Towing'TtcCONF 1: empty
Towing'await_towConf 1: empty

```

```

Towing'TtcREQ 1: empty
authPaym'serviceReq 1: empty
authPaym'paymOK 1: empty
authPaym'paymNOK 1: empty
authPaym'awaitResponse 1: empty
authPaym'CapREQ 1: empty
authPaym'CapOK 1: empty
authPaym'CapNOK 1: empty
authPaym'toSend 1: empty
authPaym'accepted 1: empty
authPaym'rejected 1: empty
reqGar'garNOK 1: empty
reqGar'garOK 1: empty
reqGar'merge2 1: empty
reqGar'Crg 1: empty
reqGar'CrgOK 1: empty
reqGar'Cgc 1: empty
reqGar'garConf_sent 1: empty
reqGar'await_garREQ 1: empty
reqGar'CrgNOK 1: empty
reqGar'GgcCONF 1: empty
reqGar'gar_not_approved 1: empty
reqTow'towNOK 1: empty
reqTow'garOK 1: empty
reqTow'towOK 1: empty
reqTow'towing_queried 1: empty
reqTow'Crt 1: empty
reqTow'CrtOK 1: empty
reqTow'towing_Requested 1: empty
reqTow'CrtNOK 1: empty
reqTow'towingDenial_decided 1: empty
reqTow'CtcREQ 1: empty
reqTow'await_towConf_reply 1: empty
reqTow'CtcCONF 1: empty
reqRent'rentNOK 1: empty
reqRent'p1 1: empty
reqRent'rentOK 1: empty
reqRent'Crr 1: empty
reqRent'renting_queried 1: empty
reqRent'renting_Requested 1: empty
reqRent'CrcCONF 1: empty
reqRent'await_rentConf_confirmation 1: empty
reqRent'CrcREQ 1: empty
reqRent'CrrOK 1: empty
reqRent'rentingDenial_decided 1: empty
reqRent'CrrNOK 1: empty
Renting'Rrr 1: empty
Renting'RrcCONF 1: empty
Renting'RrrNOK 1: empty
Renting'RrrOK 1: empty
Renting'renting_accepted 1: empty
Renting'RrcREQ 1: empty
Taxi'XrxNOK 1: empty
Taxi'XrxOK 1: empty
Taxi'taxi_accepted 1: empty
Taxi'XxcREQ 1: empty
Taxi'Xrx 1: empty
Taxi'XxcCONF 1: empty
reqTaxi'taxiNOK 1: empty
reqTaxi'taxiOK 1: empty
reqTaxi'rentNOK 1: empty
reqTaxi'Crx 1: empty
reqTaxi'taxi_queried 1: empty
reqTaxi'taxi_Requested 1: empty
reqTaxi'CxcCONF 1: empty
reqTaxi'await_taxiConf_reply 1: empty
reqTaxi'CxcREQ 1: empty
reqTaxi'CrxOK 1: empty
reqTaxi'taxiDenial_decided 1: empty
reqTaxi'CrxNOK 1: empty
val it = () : unit

```

Figure 6-25 Description of a marking node number 1086

Alternatively, it is also possible to check whether all dead markings have tokens only in the final “serviceDone” place. Since the formula contains all places from the model, only a part of the places is shown in Figure 6-26. All complete queries are available in page “queries” in the file “GSM-basic.cpn” on the attached CD.

```
( ( Mark.reqTaxi'await_taxiConf_reply 1 n) == empty ) andalso
( ( Mark.reqTaxi'taxiDenial_decided 1 n) == empty ) andalso
( ( Mark.reqTaxi'taxi_Requested 1 n) == empty ) andalso
( ( Mark.reqTaxi'taxi_queried 1 n) == empty ) andalso
( ( Mark.reqTow'CrT 1 n) == empty ) andalso
( ( Mark.reqTow'CrTNOK 1 n) == empty ) andalso
( ( Mark.reqTow'CrTOK 1 n) == empty ) andalso
( ( Mark.reqTow'CtcCONF 1 n) == empty ) andalso
( ( Mark.reqTow'CtcREQ 1 n) == empty ) andalso
( ( Mark.reqTow'await_towConf_reply 1 n) == empty ) andalso
( ( Mark.reqTow'towingDenial_decided 1 n) == empty ) andalso
( ( Mark.reqTow'towing_Requested 1 n) == empty ) andalso
( ( Mark.reqTow'towing_queried 1 n) == empty );
val deads = ListDeadMarkings ();
val paths = map verDead(deads);
```

```
val verDead = fn : Node -> bool
val deads = [17,1086,1082,1046] : Node list
val paths = [true,true,true,true] : bool list
```

Figure 6-26 Verification whether all dead states have tokens in one place

As seen in Figure 6-27 results can also be conveniently streamed to a file for further analysis with external tools.

```
let
  val fid = TextIO.openOut "deadMarkings.txt"
  val _ = TextIO.output(fid, "Details of dead markings: \n")
  val _ = EvalNodes(ListDeadMarkings (),
    fn n => STRING.output(fid, NodeDescriptor n) )
in
  TextIO.closeOut(fid)
end
```

```
val it = () : unit
```

Figure 6-27 Streaming query results to a file

Even though it should be possible to find and display a marking with certain constraints (like where tokens are in both states “rentNOK” and “garNOK”), most probably model’s complexity results in an error depicted in Figure 6-28.

```

(* display first finding *)
fun marking n = (Mark.goalSequence'rentNOK 1 n = [(1,1,"")]
  andalso Mark.goalSequence'garNOK 1 n = [(1,1,"")]);
val findFirst = List.nth(SearchNodes (
  EntireGraph,
  fn n => (marking n),
  NoLimit,
  fn n => n,
  [],
  op ::),0);
print(NodeDescriptor findFirst);

```

```

Error in ML expression:
val marking = fn : Node -> bool
Unhandled Exception: Subscript
  with message: subscript out of bounds
□build\evalloop.sml:311.19-311.22
□build\evalloop.sml:93.55
□util\stats.sml:164.40
□build\compile.sml:382.13-382.58
□build\evalloop.sml:93.55
□boot\list.sml:47.35-47.44

```

Figure 6-28 Error message while searching state space of a complex model

As depicted in Figure 6-29, the query behaves correctly in a similar model, but simplified by not having the garage, towing, and renting and taxi interactions.

```

(* display first finding *)
fun marking n = (Mark.goalSequence'rentNOK 1 n = [(1,1,"")]
  andalso Mark.goalSequence'garNOK 1 n = [(1,1,"")]);
val findFirst = List.nth(SearchNodes (
  EntireGraph,
  fn n => (marking n),
  NoLimit,
  fn n => n,
  [],
  op ::),0);
print(NodeDescriptor findFirst);

```

```

val marking = fn : Node -> bool
val findFirst = 24 : Node
24:
cancPaym'p2 1: empty
cancPaym'paymCanc 1: empty
cancPaym'CcpREQ 1: empty
cancPaym'CcpCONF 1: empty
cancPaym'awaitResponse2 1: empty
cancPaym'toSend2 1: empty
cancPaym'canceled 1: empty
cancGar'towNOK 1: empty
cancGar'garCanc 1: empty
Bank'BapREQ 1: empty
Bank'CapOK 1: empty
Bank'CapNOK 1: empty

```

Figure 6-29 Correct result for a query searching for a specific marking

Reachability properties

ASK-CTL logic allows formulating various queries to test the SS. One can check, for example, whether after reaching a certain state (for example when renting and towing has been rejected), some other state (like garage confirmation) can still be reached. The proof of a correct behavior of the system is presented in Figure 6-30.

```

fun cond n = (Mark.goalSequence'rentNOK 1 n <> empty
andalso Mark.goalSequence'towNOK 1 n <> empty);
val condStates = SearchNodes (
  EntireGraph,
  fn n => (cond n),
  NoLimit,
  fn n => n,
  [],
  op ::);
val amountOfConditions = length (condStates);
fun res n = (Mark.goalSequence'booking_confirmed 1 n <> empty);
val resState = NF("",res);
val check = POS(resState);
fun checkAll n = eval_node check n;
map checkAll(condStates);

```

```

val cond = fn : Node -> bool
val condStates = [515,222] : Node list
val amountOfConditions = 2 : int
val res = fn : Node -> bool
val resState = NF("",fn) : A
val check = EXIST_UNTIL (TT,NF("",fn)) : A
val checkAll = fn : Node -> bool
val it = [false,true] : bool list

```

Figure 6-30 Query to check whether garage can be confirmed even after renting and towing has been rejected in GSM

Two results come from the two possible tokens that may reside in place towNOK:

- rentNOK = (1,1,""), towNOK (1,2,"") for node 515
- rentNOK = (1,1,""), towNOK (1,1,"") for node 222

In the first case, the second garage will be canceled and since no other garages are available, there is no garage, nor towing booking is made.

It has to be noted that this test at the beginning found three nodes revealing a bug, particularly difficult to spot, that allowed token value (1,2,"") to appear in place "rentNOK". It was found that there was no guard "(ctrl,"depUnres")" on the arc between place "authPaym.serviceReq" and transition "authPaym.sendReq". In this way, a token could travel from place "goalSequence.merge1" again to sub-page "authPaym". In addition, correcting this bug was straightforward and not only resulted in reasonable query results, but also decreased the state space by 10%. This is an example that by analyzing automated verification results, one can efficiently check the behavior.

Safety properties

Besides checking that some states are reachable, one can also verify that some situations will never occur. In the example shown in Figure 6-31 it is verified whether after reaching a state "rentOK", it is not possible to cancel payment. In this case the deposit is required for at least a replacement car that will be delivered.

```

fun cond n = (Mark.goalSequence'rentOK 1 n <> empty);
val condStates = SearchNodes (
  EntireGraph,
  fn n => (cond n),
  NoLimit,
  fn n => n,
  [],
  op ::);
val amountOfConditions = length (condStates);
fun res n = (Mark.goalSequence'payment_canceled 1 n <> empty);
val resState = NF("",res);
val check = INV(NOT(resState));
fun checkAll n = eval_node check n;
map checkAll(condStates);

```

```

val cond = fn : Node -> bool
val condStates =
[984,983,982,97,933,932,931,930,929,867,866,865,864,790,789,788,787,706,705,
704,703,702,69,618,617,616,615,614,525,524,523,522,435,434,433,355,354,286,
285,227,226,177,176,133,1066,1065,1050,1049,1023,1022] : Node list
val amountOfConditions = 50 : int
val res = fn : Node -> bool
val resState = NF("",fn) : A
val check = NOT (EXIST_UNTIL (TT,NOT (NOT (NF("",fn)))))) : A
val checkAll = fn : Node -> bool
val it =
[true,true,true,true,true,true,true,true,true,true,true,true,true,true,
true,true,true,true,true,true,true,true,true,true,true,true,true,true,
true,true,true,true,true,true,true,true,true,true,true,true,true,true,
true,true,true,true,true] : bool list

```

Figure 6-31 Reachability verification

A list of true values proves the correctness of the system for every possible occurrence of state “rentOK”. The repetitive nature of the result values allowed many bugs to be quickly found during the design of the model.

Liveness properties

Testing for a state that will inevitably occur verifies that the system will fulfill its task and is one of the most important verification techniques. Two ASK-CTL formulas should be used for this purpose:

- EV (A) is true if the argument A becomes true eventually, starting from the current state.
- ALONG (A) is true if there exists a path for which the argument, A, holds for every state

However, both of them do not work as intended in CPN Tools. As a proof, a counter example is shown in a simple net in Figure 6-32.

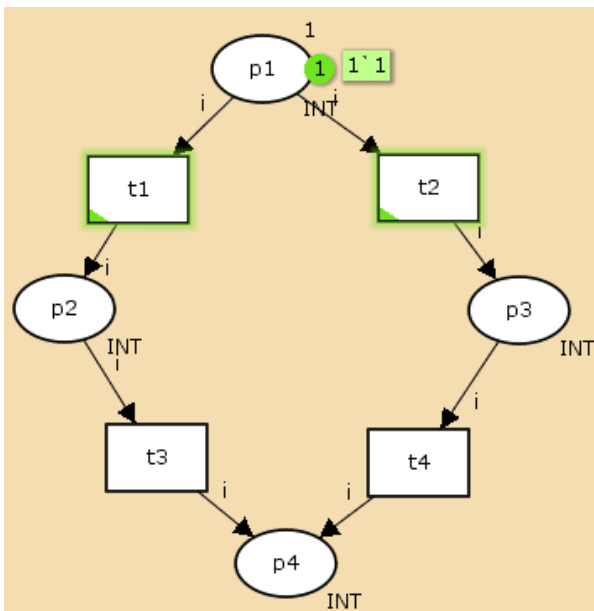


Figure 6-32 Simple Petri net to check liveness formulas

The query depicted in Figure 6-33 evaluates ALONG formula that verifies whether there exists a path from initial node where place “p2” is always empty. The result is false even though one can see that the path can be reached by triggering transitions “t2” and “t4”.

```

fun done n = (Mark.test'p2 1 n = empty );
val reach = ALONG(NF("",done));
eval_node reach InitNode;

```

```

val done = fn : Node -> bool
val reach = NOT (FORALL_UNTIL (TT,NOT (NF ("",fn)))) : A
val it = false : bool

```

Figure 6-33 Simple test of ALONG formula

The query depicted in Figure 6-34 evaluates EV formula that checks whether for all execution paths it is possible to reach a marking where place p2 contains a token. The result

“true” is counter intuitive since a token can travel through p3 instead without visiting p2 at all.

```
fun done n = (Mark.test'p2 1 n <> empty );
val reach = EV(NF("",done));
eval_node reach InitNode;

val done = fn : Node -> bool
val reach = FORALL_UNTIL (TT,NF ("",fn)) : A
val it = true : bool
```

Figure 6-34 Simple test of EV formula

Since ASK_CTL is not built in the CPN Tools, it is believed that the formulas are evaluated incorrectly by the model checker. That might be connected with the Scc optimizations that have been introduced into the tool. EV and ALONG are derived from a probably corrupted FORALL_UNTIL (A1,A2) formula, contrary to POS and INV, that are derived from EXIST_UNTIL. The bug has been reported to CPN Tools support (bug id: 2492) and awaits resolution.

6.1.5. GSM conclusions

Even though the methodology proposed leads to a specified model of the system, one can argue that the result of the whole methodology from [CASS] presents a model that lacks clarity. It is believed that the lack of abstraction does not allow understanding quickly how the parts interact, forcing a designer to study it carefully before changing anything. This disadvantage is important in the prospect of any future modifications and especially irritating when the change is small and refers to one specific part of the model. A proper usage of abstraction would help greatly by allowing it to find the specific part of the system with a top-down approach and after a shorter and simpler analysis, changing only that part.

Another drawbacks are the additional steps of creating an UML-like diagram before reaching a Petri net model. One can also notice that the Petri net has the interaction “confirm booking” (right after a place “merge 6”) which should be intuitively located where transition “aux8” is. This is a result of complex “spaghetti” wiring that was directly translated to a Petri net. Since the confirmation can be reached anyway and the overall functionality is not changed (apart from additional delay), it has been kept for the purpose of showing consequence of inefficient modeling.

Additionally, the lack of a clear division of logic between interacting parties does not allow modeling some crucial behaviors such as: communication errors and duplicated or invalid requests. Fusion places used as communication channels makes it both human and programmatically difficult to identify synchronized processes, and thus to find the root threads to generate WSDL description. It may also be problematic to design some structures to be used in described later (in chapter 8) model transformation.

In order to improve the above mentioned disadvantages, a top-down abstraction refining methodology is proposed.

6.2. Top-down abstraction refining

To face the clarity problem and possibly improve the scalability of a model, this section presents a purely Petri net based methodology. It is inspired by the first three steps of the [CASS] methodology, but then proceeds differently in specifying a model.

6.2.1. Methodology

The major difference from the previous approach is that the single composite collaboration specified in step 3 of [CASS] and shown in Figure 6-2 is the highest abstraction of a Petri net and an overview of the whole physical system. Different parties are modeled with single sub-pages references connected to all possible roles (represented by places) that they can play. Roles of different parties are connected by collaborations in a form of a transition or a sub-page.

The “top-down abstraction refining” name comes from a recommendation of adding interfaces to highest abstraction model on both interacting sides for each communication. Contrary to the previous methodology, the approach allows to perform communication without fused places, which are believed, through their loosely coupled approach, to cause confusion in a model. It is also worth to mention that separation of parties on the highest level is beneficial not only from the clear overview, but also opens new possibilities to model both orchestration and choreography.

It has to be noted that the sub-page logic no longer relates only to the specific service, such as the orchestration in [CASS], but can describe all collaborations that may appear. It is believed that this modification increases not only readability of the model, but also verification possibilities. Having all the services rather than those logically connected, coupled in one system allows verifying not only whether they are correct independently, but also whether they are not interfering with each other. One can also divide all the collaborations in several groups related to specific services in order to increase the readability. Grouping elements is intuitively supported by CPN Tools including common elements and collaborations sharing between groups.

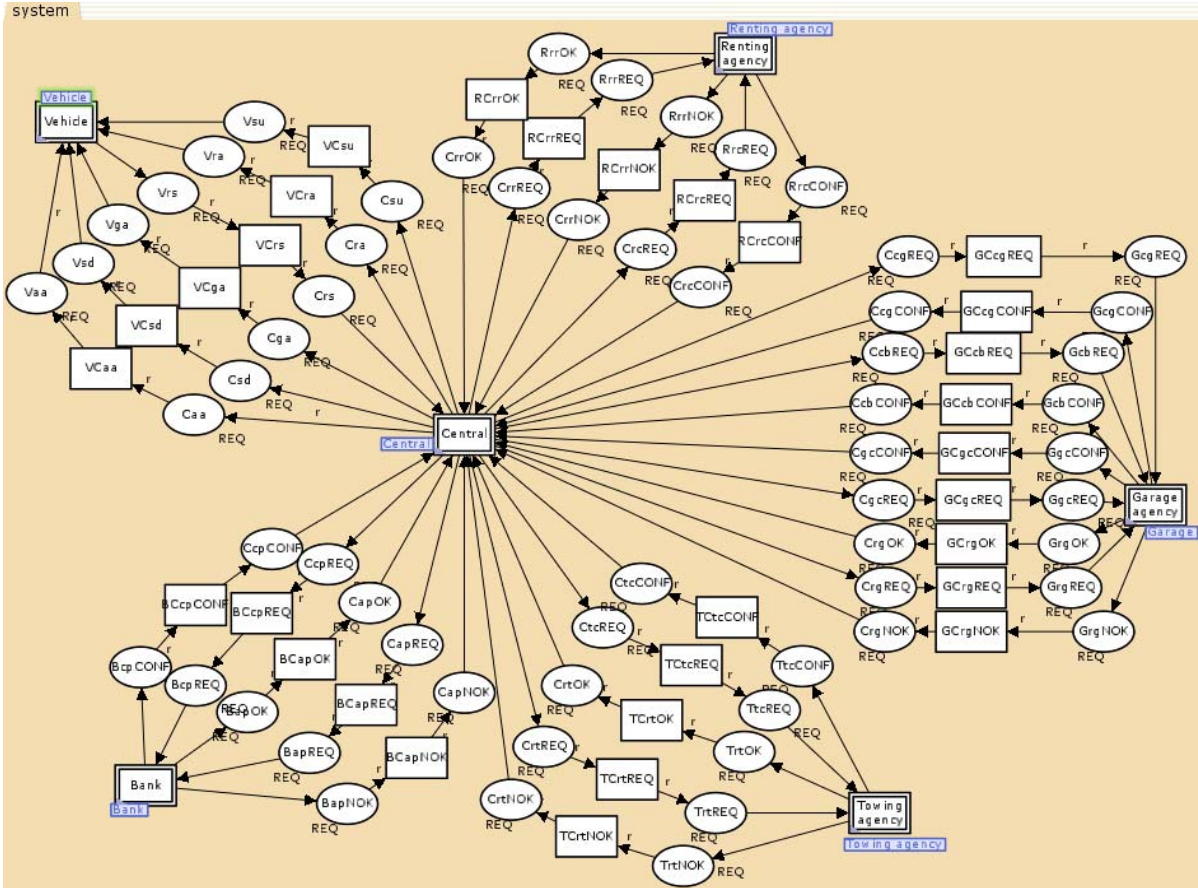


Figure 6-35 Highest abstraction overview of top-down abstraction refining methodology

As can be seen in Figure 6-35, overview abstraction shows clearly all parties with their both required and provided interfaces defined on component diagram in chapter 4.2.5. The system structure visually reveals client/server architecture, but peer-to-peer may be modeled with similar simplicity. That opens the methodology not only for orchestration scheme but also choreography. For the sake of visual clarity, acronyms have been used to describe interface names and communication channels. Interface name begins with a capital letter that identifies a party (C-Central, T-Towing Agency, etc.), followed by two letters identifying interaction name (rt – Request Towing, tc – Towing Confirm, etc.) and capital letters identifying whether it is a request (REQ), positive/negative reply (OK/NOK) or confirmation (CONF). Needless to say, acronyms should be unique or CPN Tools will mark them as errors. Almost all locations are of type INT*INT*INT that allows storing tokens with details (in original order):

- request id “i” distinguishes requests from each other (for simplicity, we analyze one request)
- counter “id1” (or sometimes “id2”) specifies number of currently chosen garage
- variable “s” represents specific failure details to be defined in implementation phase (and for simplicity equals 0)

Collaborations and their interfaces may be distinguished with the use of groups as shown in Figure 6-36.

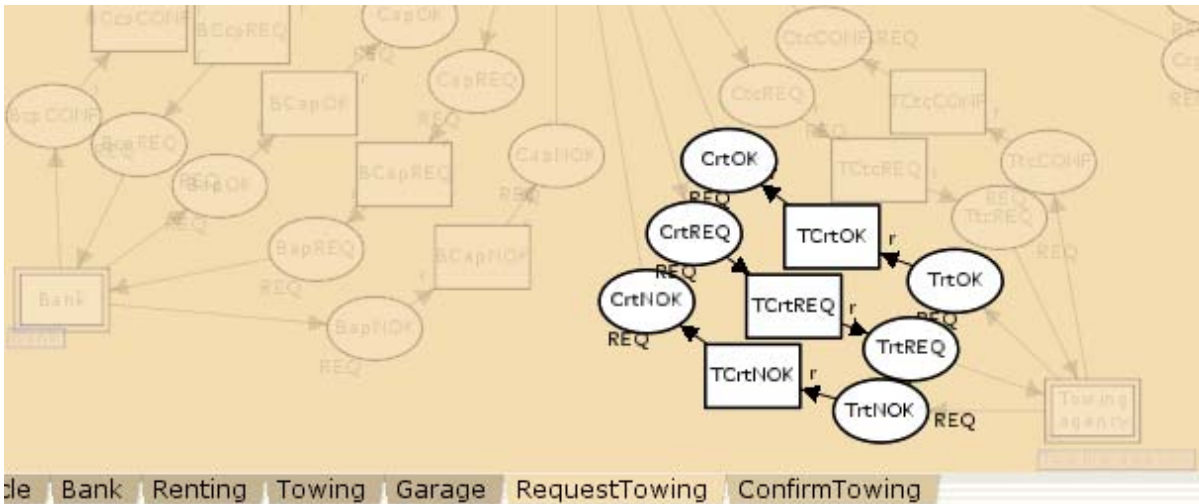


Figure 6-36 Request Towing collaboration with its interfaces

It has to be noted that interfaces (roles) of interacting parties may be connected with either a transition or a sub-page. Whereas transitions model simple communication, sub-pages can describe communication protocols, conveniently hiding process logic in a lower abstraction level. In addition, it is possible to reuse sub-nets for several interactions, making it easy and efficient to design communication templates and modify all of them at the same time. Since protocols are not the main focus of this thesis and increase state space, we continue using simple transitions.

Comparing to the previous methodology, one can also define a process for a vehicle with precisely specified all possible return messages, as shown in Figure 6-37.

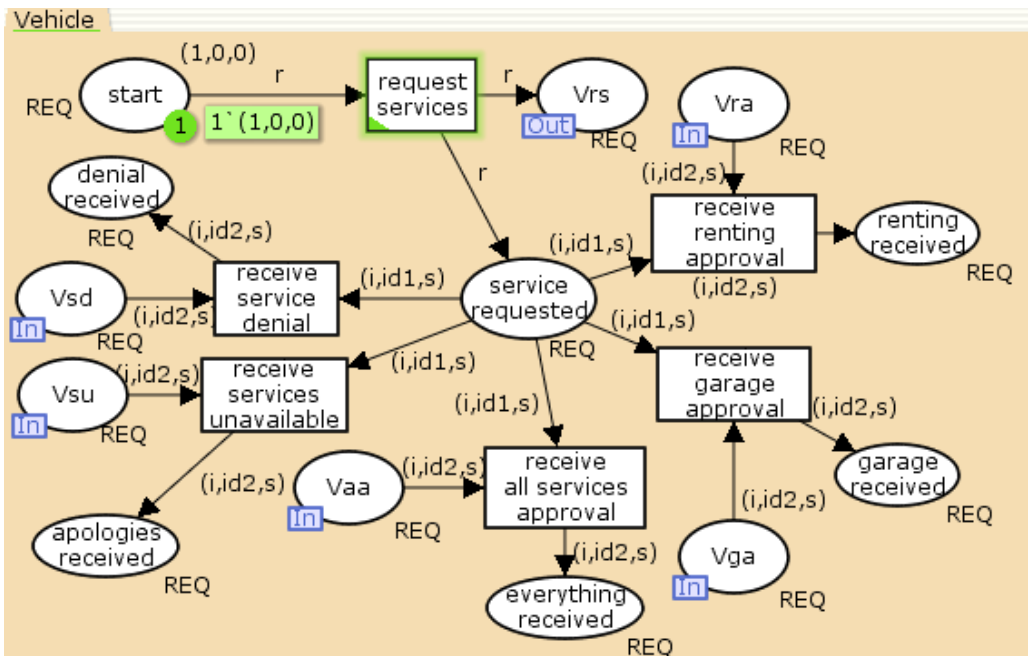


Figure 6-37 Vehicle logic

After requesting services by a required interface “Vrs”, a vehicle waits until any of provided interfaces: “Vsd” (service denial), “Vsu” (services unavailable), “Vga” (garage approved), “Vra” (renting approved) or “Vaa” (all approved) is invoked.

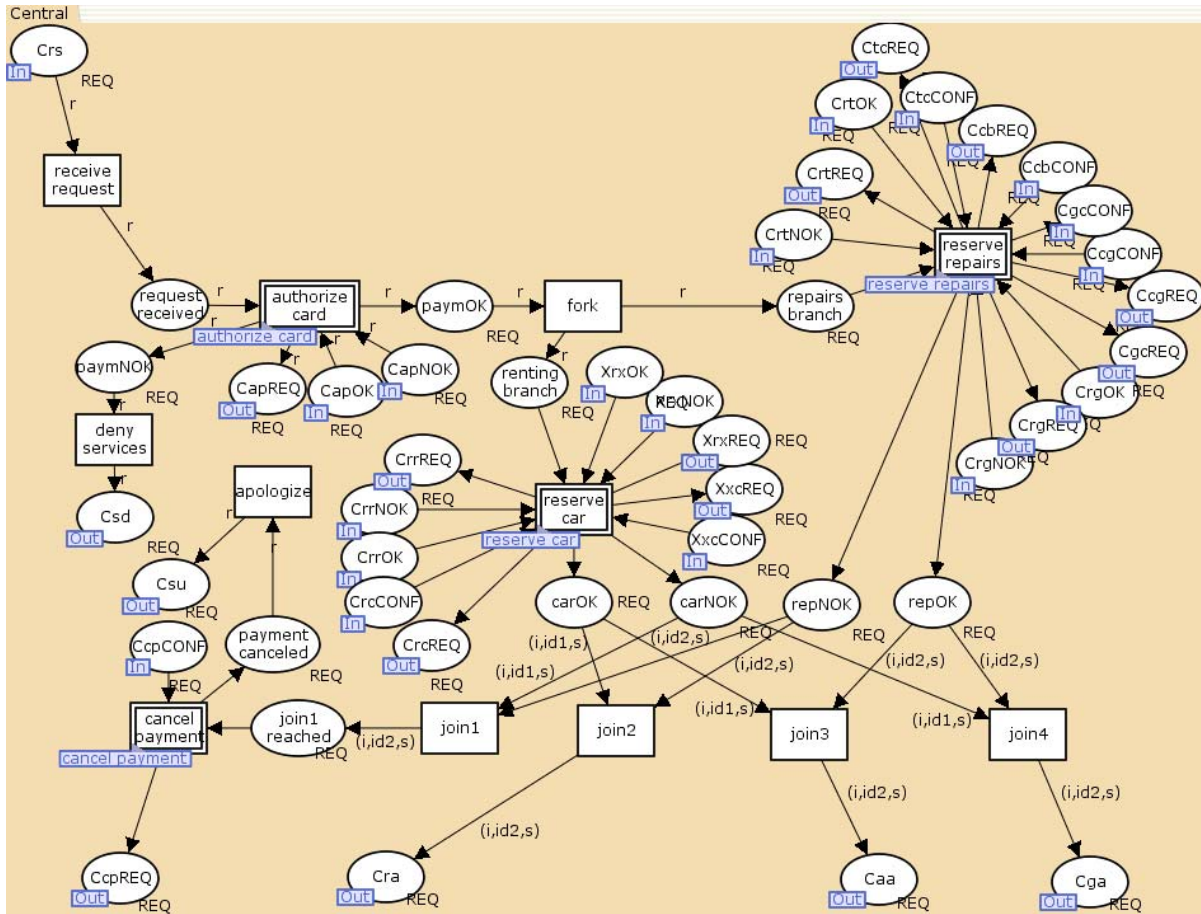


Figure 6-38 A view at Central's main logic

As can be seen in Figure 6-38, it is necessary that all roles (interfaces) that a party has are also included in party's sub-page and connected to references of even lower abstraction levels. Even though this approach requires more locations and seems to make model less readable, it allows clear binding of functionalities and their access points. To increase clarity, it is always possible to group interfaces related to certain parties or activities, as shown in Figure 6-39. Since our Central fulfills only one service to arrange services for a broken vehicle, its logic lies just below the overview abstraction. In case there would be another service, either its logic could be added to the same page or all services could be again pushed to lower abstraction levels, leaving only their sub-page references and related interfaces. All this flexibility in arranging systems not only makes it easy to implement designer's ideas, but also increases both clarity and scalability of a model.

The process logic takes a lot after the activity diagram depicted in Figure 4-2. The main difference, however, is the “reserve repairs” sub-page that handles all interactions with both garage and renting agencies. Similarly, sub-page “reserve car” handles car renting taxi agencies with the details hidden in lower abstraction level. Similarly to previous methodology, results of two concurrent activities (“reserve renting” and “reserve repairs”)

are matched against four join transitions (join 1 to 4) to respond to a Vehicle with a proper message.

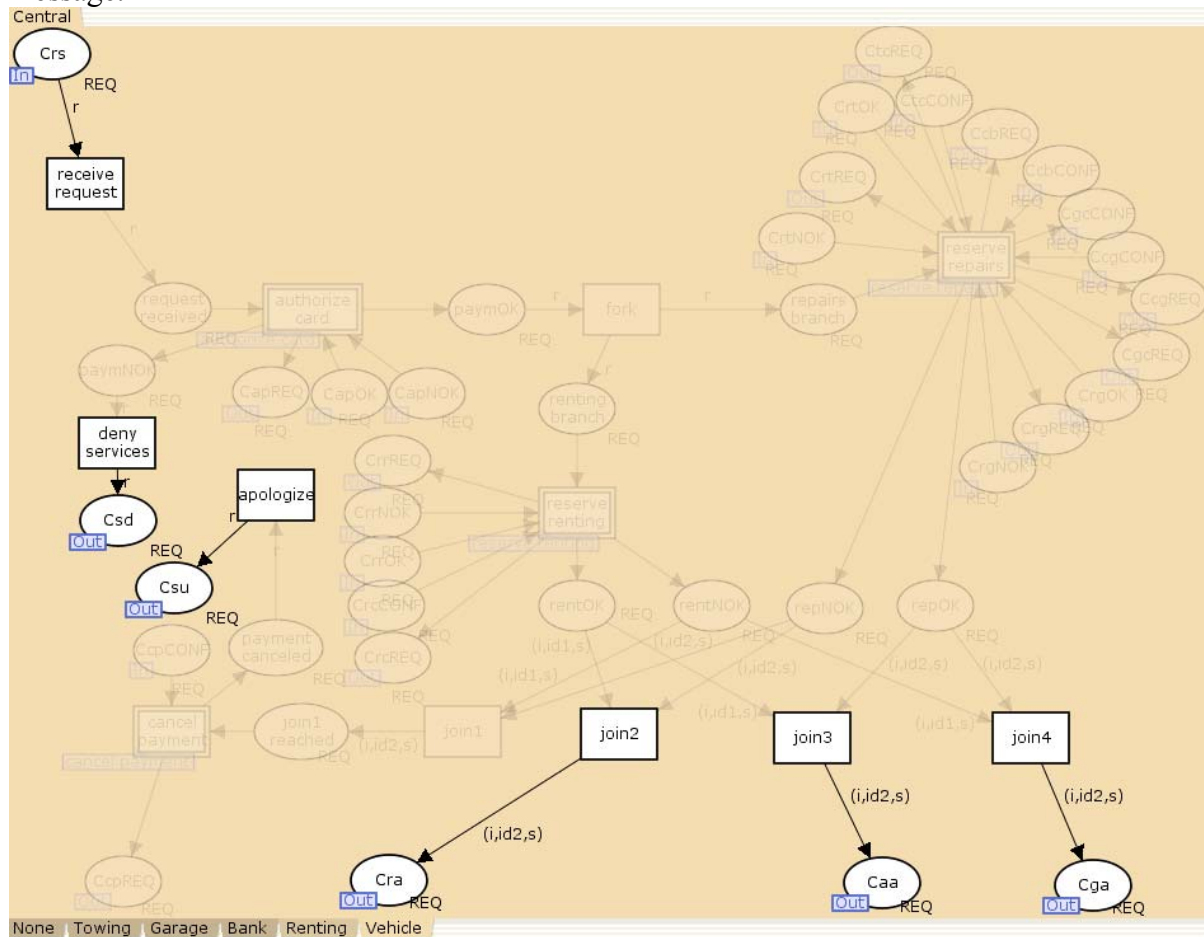


Figure 6-39 Interfaces and their transitions related to direct interaction with Vehicle

The textual representation of all the networks showing their hierarchy is presented in Figure 6-40.

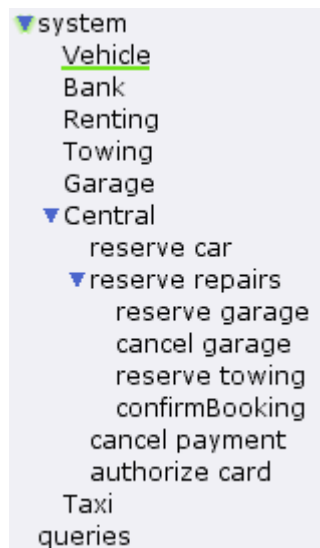


Figure 6-40 Networks hierarchy

Simple interactions like authorize payment, request towing, etc follow the identical protocol to those in the previous methodology with respect to fused places replaced by input and output ports. It is possible to analyze them in details by inspecting model “tar-basic.cpn“ on the attached CD. However, it is worthwhile to show a difference between modeling with abstractions and without. The topic is also discussed again in chapter 8.6.2, regarding verification issues. A good example for abstraction driven process modeling, is depicted in Figure 6-41, is a sub-net “reserve repairs” that groups garage and towing interfaces.

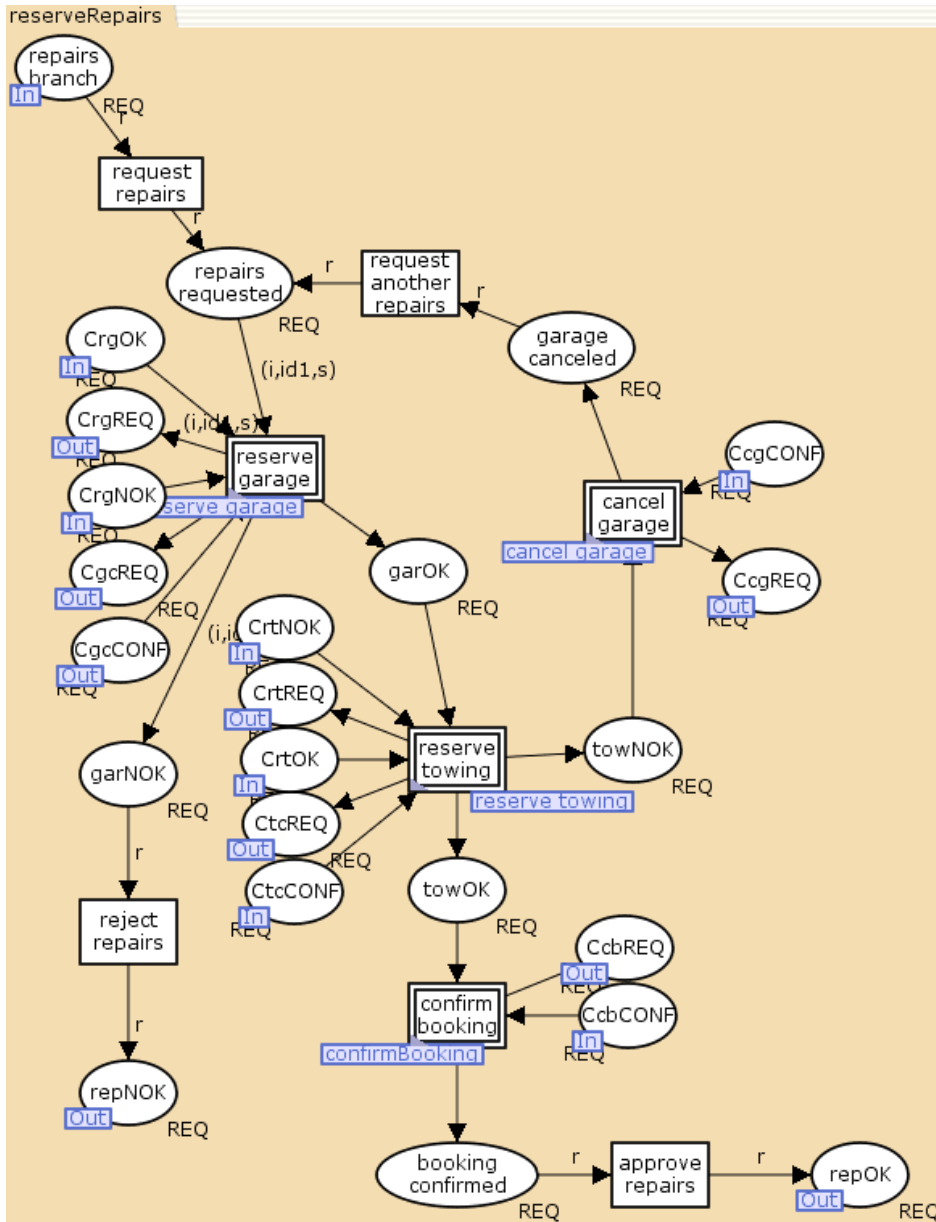


Figure 6-41 Process logic of “reserve repairs” sub-net

One can notice that all interfaces (ports places) that surrounded the sub-page in the higher abstraction level have now been distributed among even more specified sub-pages (“reserve

garage”, “reserve towing”, “confirm booking” and “cancel garage”). This allows to easily locating a process containing logic for a particular input or functionality. The process begins with a token appearing in a place “repairs branch” and following to sub-page “reserve garage”. A result of a sub-page “reserve garage” either exits the sub-page “reserve repairs” through place “repNOK” (if no garage is available to reserve), or involves sub-page “reserve towing” to arrange a towing car. The details of “reserve towing” are again conveniently hidden and the results are limited to token appearing in places “towOK” or “towNOK”. In case towing is registered successfully, the booking is confirmed in sub-page “confirm booking”. On the other hand, towing rejection results in garage cancellation and another garage request. It has to be noted that there may be more than one agencies contacted in one “reserve garage” or “reserve towing” sub-page. The details of “reserve garage” sub-page are shown in Figure 6-42.

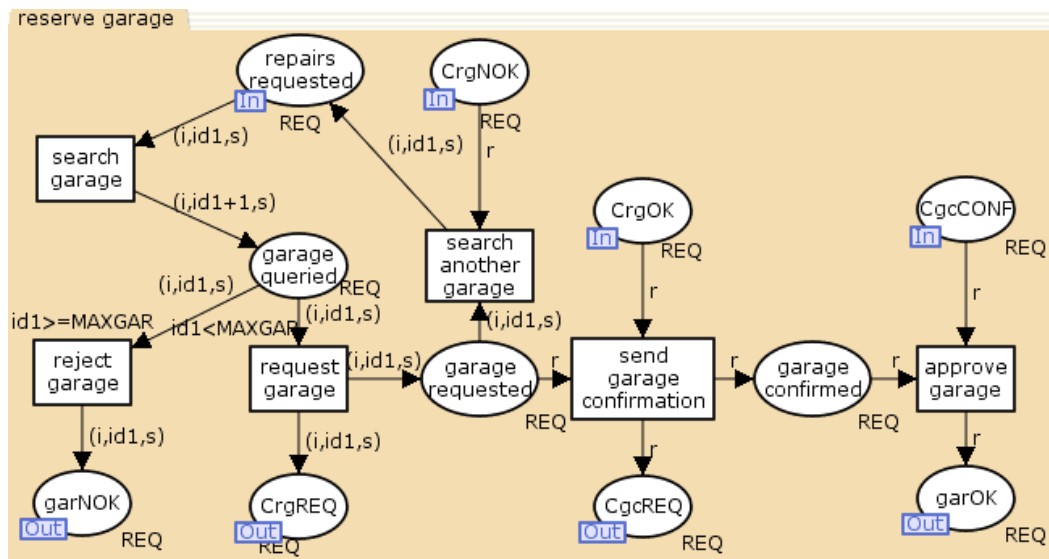


Figure 6-42 Details of “reserve garage“ sub-page

Figure 6-43 shows the lowest level of abstraction with a communication protocol while reserving garage. Process begins in place “repairs requested” (top) and proceeds to “garage queried” to decide whether there are still available garage services in UDDI registry (guard $id1 < MAXGAR$), or not (guard $id1 \geq MAXGAR$). In case there is, it is contacted via interface “CrgREQ” (from [C]entral [r]eserve [g]arage [REQ]UEST). A token travels all the way up to the highest abstraction level to be processed by Garage agency, as shown in Figure 6-43 and returns via either place “CrgOK” (approval) or “CrgNOK” (rejection). Request approval leads to another two-way confirmation communication that ends in place “garOK” (bottom-right corner). Request rejection loops into another garage request, while also increasing a counter $id1$ to limit possible number of garage services in UDDI. The counter is also increased with every garage cancellation, which prevents from requesting the same garage service again.

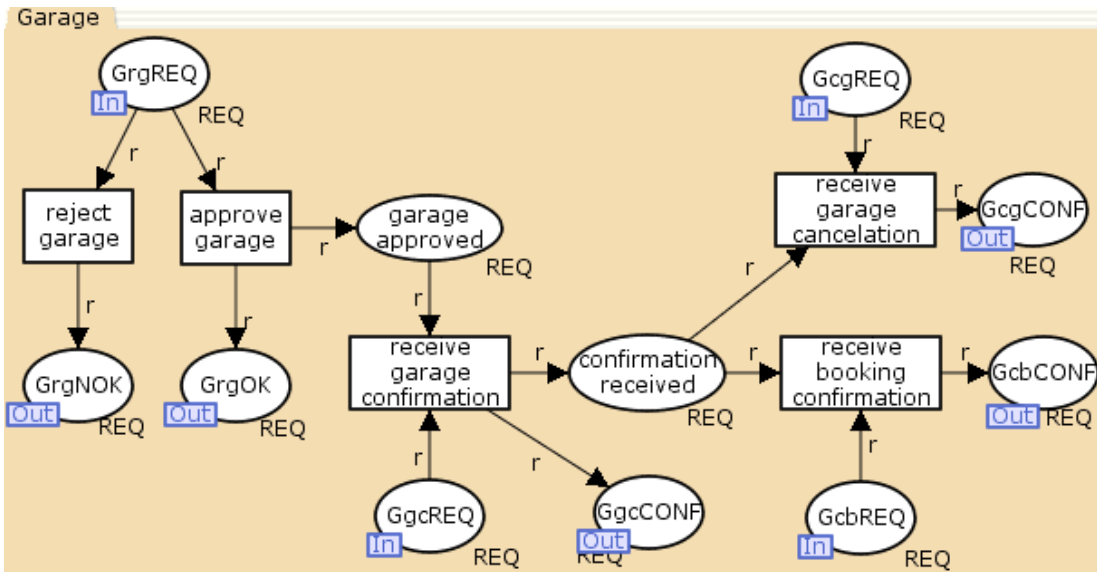


Figure 6-43 Details of garage service

As can be seen, garage service details are identical with the ones in previous methodology, but the communication always involves highest abstraction level. It makes it possible to simulate the model and follow its behavior relying only on the overview page.

6.2.2. Verification

The state space report reveals twice as big state space than in the previous methodology.

```
State Space
Nodes: 2093
Arcs: 4383
Secs: 4
Status: Full
```

```
Scc Graph
Nodes: 1317
Arcs: 3333
Secs: 1
```

The increase of state space is caused by all the additional elements related to forwarding tokens to the highest abstraction level and is believed to be a price for improved readability. Similarly as before, there are no problems with dead transitions (all places are reachable) and smaller Scc graph suggests loops in the model. Because of the different service return messages that reach a vehicle, there are more dead markings as shown in Figure 6-44.

```
ListDeadMarkings ();
val it = [22,2093,2071,2063,1909,1655,1611] : Node list
```

Figure 6-44 Dead markings in top-down abstraction refining methodology

CPN Tools allows to quickly verify whether a process terminates in expected places by checking is all final markings have a token in of vehicle's states as shown in Figure 6-45.

```

fun verDead n =
  ( ( Mark.Vehide'denial_received 1 n)<> empty ) orelse
  ( ( Mark.Vehide'apologies_received 1 n)<> empty ) orelse
  ( ( Mark.Vehide'everything_received 1 n)<> empty ) orelse
  ( ( Mark.Vehide'garage_received 1 n)<> empty ) orelse
  ( ( Mark.Vehide'renting_received 1 n)<> empty ) ;
val deads= ListDeadMarkings ();
val paths = map verDead(deads);

val verDead = fn : Node -> bool
val deads = [22,2093,2071,2063,1909,1655,1611] : Node list
val paths = [true,true,true,true,true,true,true] : bool list

```

Figure 6-45 Verification query whether final markings have tokens in expected final places

A higher number of final markings (7) than final places (5) is due to the fact that places `everything_received` and `garage_received` may contain more than one different final token. All possible tokens can be found in section `Best Upper Multi-set Bounds` of the state space report:

- `apologies_received (1,3,0)`
- `denial_received (1,0,0)`
- `everything_received (1,1,0) and (1,2,0)`
- `garage_received (1,1,0) and (1,2,0)`
- `renting_received (1,3,0)`

It can be concluded that “`renting_received`” confirmation and “`apologies_received`” are sent only if all garages have been tried. Both “`garage_received`” (and also towing), as well as “`everything_received`” confirmations, can be sent with either first or second garage being reserved.

As show in Figure 6-46 ASK_CTL extension allows verifying many custom reachability properties such as confirming a garage, even though renting and towing services have been rejected.

```

fun cond n = (Mark.reserve_car'carNOK 1 n <> empty
andalso Mark.reserve_repairs'towNOK 1 n <> empty);
val condStates = SearchNodes (
  EntireGraph,
  fn n => (cond n),
  NoLimit,
  fn n => n,
  [],
  op ::);
val amountOfConditions = length (condStates);
fun res n = (Mark.reserve_repairs'booking_confirmed 1 n <> empty);
val resState = NF("",res);
val check = POS(resState) ;
fun checkAll n = eval_node check n;
map checkAll(condStates);

val cond = fn : Node -> bool
val condStates = [532,1076] : Node list
val amountOfConditions = 2 : int
val res = fn : Node -> bool
val resState = NF("",fn) : A
val check = EXIST_UNTIL (TT,NF("",fn)) : A
val checkAll = fn : Node -> bool
val it = [true,false] : bool list

```

Figure 6-46 Reachability test whether booking can be confirmed after towing and both renting car and taxi have been rejected

Two markings fulfill the input conditions:

- token (1,0,0) in place “`carNOK`” and token (1,1,0) in place “`towNOK`”
- token (1,0,0) in place “`carNOK`” and token (1,2,0) in place “`towNOK`”

Since new garage can be requested only if second value is less than 2 (MAXGARAGE=2), the second case returns false as expected.

Similarly as in previous methodology, reachability formulae can check counter examples for safety formulae like the one shown in Figure 6-47, checking that payment will not be canceled once renting or taxi is reserved.

```

fun cond n = (Mark.reserve_car'carOK 1 n <> empty );
val condStates = SearchNodes (
  EntireGraph,
  fn n => (cond n),
  NoLimit,
  fn n => n,
  [],
  op ::);
val amountOfConditions = length (condStates);
fun res n = (Mark.Central'payment_canceled 1 n
  <> empty);
val resState = NF("",res);
val check = INV(NOT(resState) );
fun checkAll n = eval_node check n;
map checkAll(condStates);

val cond = fn : Node -> bool
val condStates =
[940,939,938,845,844,751,750,661,660,576,575,497,496,427,426,364,310,261,215,
2089,2085,2079,2070,2057,2041,2022,2021,1997,1996,1968,1967,1934,1933,1932,
191,1892,1891,1890,1844,1843,1842,1841,1788,1787,1786,1785,1725,1724,1723,
1658,1657,1656,1583,1582,1581,1502,1501,1500,1417,1416,1415,1414,1413,1324,
1323,1322,1321,1320,1228,1227,1226,1225,1133,1132,1131,1130,1036,1035,1034]
: Node list
val amountOfConditions = 79 : int
val res = fn : Node -> bool
val resState = NF("",fn) : A
val check = NOT (EXIST_UNTIL (TT,NOT (NOT (NF("",fn))))) : A
val checkAll = fn : Node -> bool
val it =
[true,true,true,true,true,true,true,true,true,true,true,true,true,true,
true,true,true,true,true,true,true,true,true,true,true,true,true,true,
true,true,true,true,true,true,true,true,true,true,true,true,true,true,
true,true,true,true,true,true,true,true,true,true,true,true,true,true,
true,true,true,true,true,true,true,true,true,true,true,true,true,true,
true,true,true,true,true,true,true,true,true,true,true,true,true,true,
true,true,true,true] : bool list

```

Figure 6-47 Verification query to check whether for all occurrences of both renting or taxi approval, deposit payment will not be canceled

A list of true results proves behavior correctness for all possible preconditions.

Similarly as in previous methodology, a lack of liveness verification does not allow complete verification. Therefore, another verification technique needs to be used to test the model.

6.2.3. TAR conclusions

The methodology addressed problems regarding clarity of the system by organizing all services hierarchically beginning from a physical overview abstraction level. This opens new modeling possibilities (like choreography), as well as enables further model transformations to generate WSDL descriptions of services, for example. Single interactions between services which are based on single transitions may be easily replaced by protocols, filters, monitors, etc.

However, because of the additional elements that had to be introduced to precisely define hierarchy, a state space has doubled. This is caused mostly by the communication through input and output ports which is more element extensive than fusion elements due to its hierarchical nature. However, it is believed that the additional redundancy is justified by the clarity and organization of interacting elements. Possible state space explosion problems in bigger systems should be solvable by component optimization described at the end of the chapter.

6.3. CPN Tools conclusions

CPN Tools has proven to be a very flexible tool for intuitive creation and simulation systems in both abstract and detailed way. The models explained in this chapter can be used for further analysis like multiple concurrent requests, message duplication problems, etc. CPN Tools allow many approaches to designing a system depending on the requirements and designer's personal preferences. One can create systems in a bottom-up or top-down model development [CPNT], matching conveniently (if possible) ports and sockets of different abstraction levels. The 5-step goal sequence methodology shows an orchestration approach, whereas the top-down abstraction refining allows to clearly organizing system's hierarchy at the cost of increased state space. One can also use a tradeoff between using hierarchical input/output ports and loosely organized fused places to find a best solution between readability and efficiency. It is worth mentioning that apart from the two presented methodologies, there are also other possibilities such as:

- use case driven development [UIPB] is a methodology that concentrates mainly on user needs and allows incrementally adding use case functionalities; each iteration brings a new version of the model or a new version of the software; before assigning use cases to iterations they should be prioritized by importance to the user and risk; use cases may be added according to methodology [PAVT], as shown in Figure 6-48
- BPEL transformation is a promising possibility because of the number of verification of many existing BPEL specifications; most of the BPEL techniques can be mapped directly to a Petri net

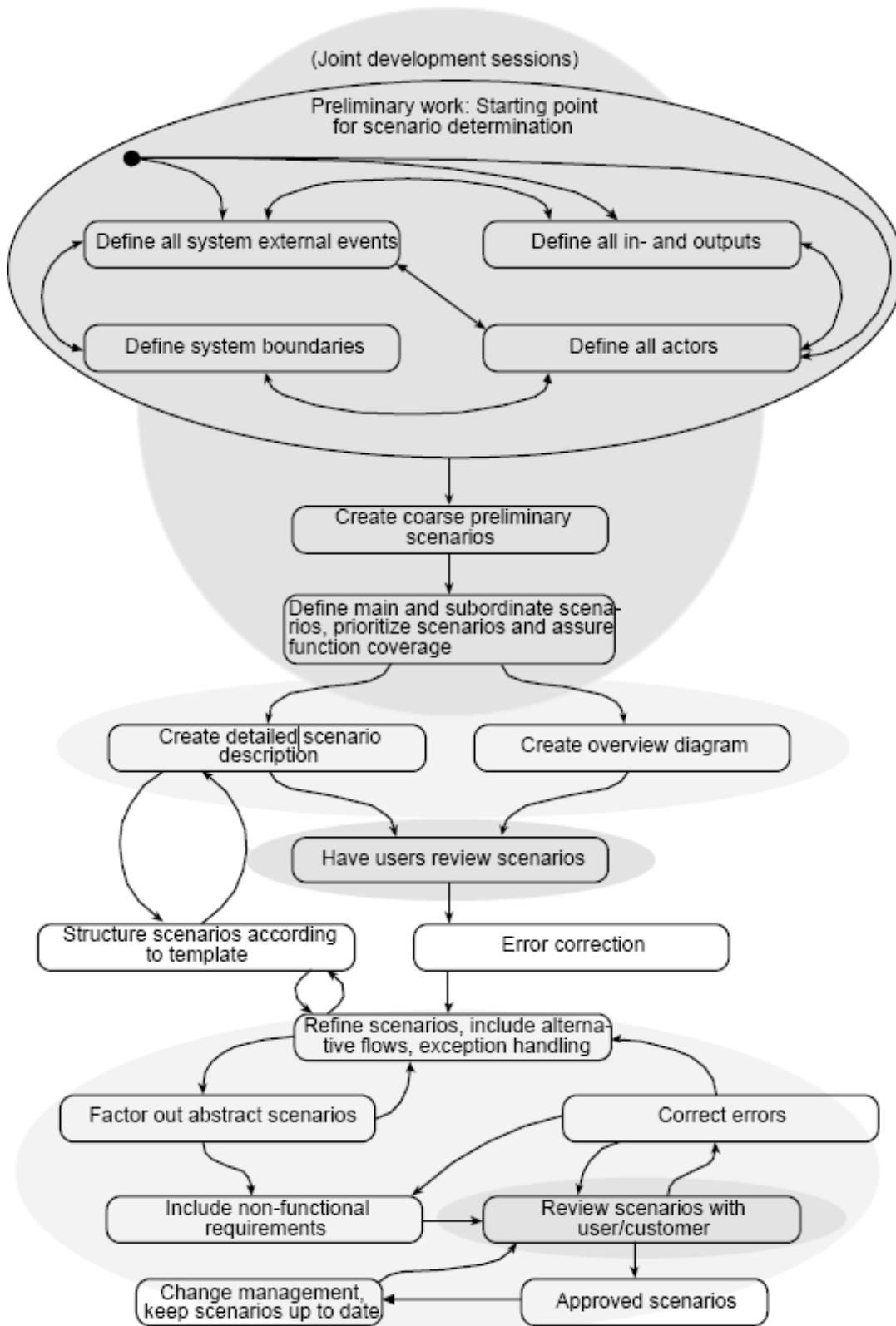


Figure 6-48 Scenario Elicitation, Scenario Creation and Structuring [PAVT]

There are some disadvantages that have been experienced in this thesis while using CPN Tools:

- significant amount of ambiguous bugs, shown in Figure 6-49, that appear at random times mostly during simulation of complex systems and freeze the application for several minutes

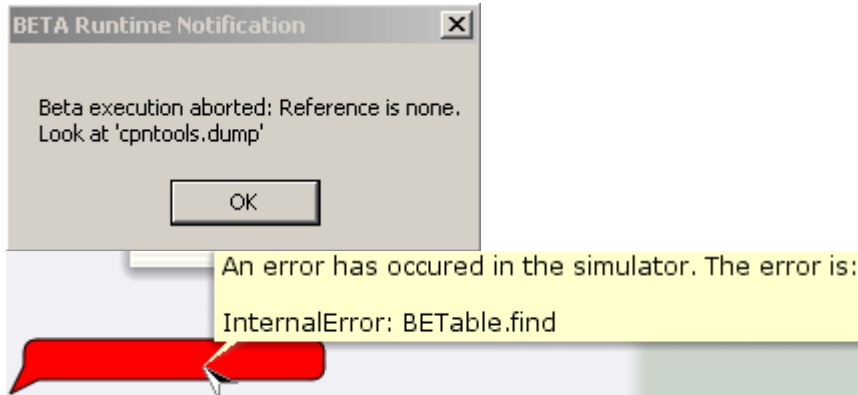


Figure 6-49 Ambiguous errors while running simulation

- slow and automatic syntax checking may irritate a designer by displaying errors during model creation
- state space is generated slowly and is mandatory before checking any formula (no on-the-fly model checking)
- ASK_CTL proof checker does not generate counter examples that could help a designer to locate a bug
- no default values for place's color set and arc identifier

It is believed that the complexity of a model causes most of the aforementioned disadvantages and thus some techniques are proposed here to decrease the problems:

- on-the-fly checker without a necessity to calculate whole state space
- counter example for CTL formulas
- possibility to interrupt the state space generation
- sealing transitions – to abstract away from the details and thereby not include a sealed component in formal model structure

Sealing transitions adheres to the MDD by hiding unnecessary details to focus on the important ones and by doing so, relieves a state space from “uninteresting” markings. All of the sub-transition's elements are not deleted but simply deactivated, waiting for unlocking. In this way a designer can incrementally extend the system without letting the complexity cause performance and reliability to drop. Sealing should not be programmatically challenging by simply replacing transitions, leading to sub-pages to normal transitions. One has to be cautious however, because transitions that accept or provide more arcs may change system's functionality. In this case a sub-page that was generating token on one of output arcs might generate tokens on both of them as a fork. Similarly, instead of accepting input from one of input arcs, an incorrectly transformed transition might require tokens on all of them to continue, like a join transition, and thus dead-locking a system. A simple, but not too flexible approach would be to allow sealing only transitions with one input and one output arc. One

has to note that sealing a part of a system may affect other parts of the system if they are related to it.

Although the test scenario was relatively simple, CPN Tools managed to simulate and generate complete state space on a typical home PC (P4 2,8GHz HT with 1GB RAM). Thus, there was no need for any extensive optimization techniques such as partial state space exploration. It is believed that much bigger systems could be investigated but would most probably need more powerful computational units.

Since liveness properties are the most important to check, a model needs to be verified by another tool. Instead of choosing another Petri net model checker that would address verification drawbacks of CPN Tools, a turn to Uppaal has been chosen. Firstly, Uppaal (analyzed in next chapter) is an efficient and reliable verifier and simulator. Secondly, transforming a Petri net to a compatible state machine would benefit the model understanding. Additionally, by model comparison, the biggest of Uppaal's modeling drawbacks (like abstraction) could be identified and possibly addressed. Lastly, having a model compatible with two specialized tools gives more verification options by having a possibility of combining both functionalities.

7. Uppaal

Uppaal has been created at Uppsala University in Sweden and Aalborg University in Denmark in 1997 and is meant for model responsive real time systems [TOU].

UPPAAL has an intuitive graphical editor that specifies a system by a number of interacting state machines. One can visualize model behavior by a simulator. In addition, a system can be then be tested by a model checker for logical formulas based on CTL logic, including time constraints.

Variables can be declared both globally and locally, but are limited to types:

- bool: boolean value can be true or false
- int: integer value can be both positive and negative numbers

Uppaal allows additionally declaring variable type as tables, constant or adding clock parameter to int variable to model timing properties of locations and edges.

Uppaal allows using procedures and a wide expression grammar depicted in Figure 7-1.

```

Expression → ID | NAT
              | Expression '[' Expression ']'
              | '(' Expression ')'
              | Expression '++' | '++' Expression
              | Expression '--' | '--' Expression
              | Expression AssignOp Expression
              | UnaryOp Expression
              | Expression BinaryOp Expression
              | Expression '?' Expression ':' Expression
              | Expression '.' ID

UnaryOp → '-' | '!' | 'not'
BinaryOp → '<' | '<=' | '==' | '!=' | '>=' | '>'
            | '+' | '-' | '*' | '/' | '%' | '&'
            | '|' | '^' | '<<' | '>>' | '&&' | '||'
            | '<?' | '>?' | 'and' | 'or' | 'imply'
AssignOp → ':=' | '+=' | '-=' | '*=' | '/=' | '%='
            | '|=' | '&=' | '^=' | '<<=' | '>>='
  
```

Figure 7-1 Syntax of expressions in BNF [TOU]

Locations may also be prioritized as urgent not allowing time counters increase before executing, or committed, which are executed even before urgent and model atomic actions.

Verification formulas can be written in a specification that is a subset of TCTL and has two types of path and state quantifiers:

- E – exists a path (**E** in UPPAAL)
- A – for all paths (**A** in UPPAAL)
- G – all states in a path (**G** in UPPAAL)

- F – some state in a path ($\langle \rangle$ in UPPAAL)

Uppaal allows following combinations of operators:

- $A[] p$: For all paths, p holds in all states
- $E\langle \rangle p$: For some path, p holds in some state
- $A\langle \rangle p$: For all paths, p holds in some state
- $E[] p$: For some path, p holds in all states
- $p \rightarrow q$: For all paths, if p holds, then for all paths q will hold in some state

Where $p ::= a.loc \mid gd \mid gc \mid p \text{ and } p \mid p \text{ or } p \mid \text{not } p \mid p \text{ imply } p$

Uppaal is a very efficient model checker. Even though it supports a subset of TCTL, offered formulae are enough to check safety and liveness properties. Additionally, some expressiveness of TCTL can be achieved by introducing ‘predicate’ variables updated when a system reaches a certain state. To improve efficiency (by using predicates), it supports partial verification when a part of a specification can be tested against a subset of all requirements.

Next, two typical kinds of usage are shown: model of a component and a protocol. Both models have been partially presented in chapter 4.2.7 but in an informal way, thus no automatic verification was performed.

7.1. Component analysis

Uppaal’s representation of a system specified above, from a point of view of vehicle’s communication component inside the car, is depicted on diagram in Figure 7-2. Failure investigation begins with an abstract process “Sensors” which models problem detection. Vehicle’s logic classifies the problem either as serious, asking GPS for current position, or a small bug, logging it for next service control. Location “analysis”, as well as other interacting locations, has timeout channel simulating the software system malfunction and asking the driver to contact the Central instead. GPS component either provides a current position of a car or asks a driver to call the Central. Bank reader component asks for a card and provides card data to the system or displays card error message if the data is corrupt. If both card and GPS data are correct, the service request is sent to the Central.

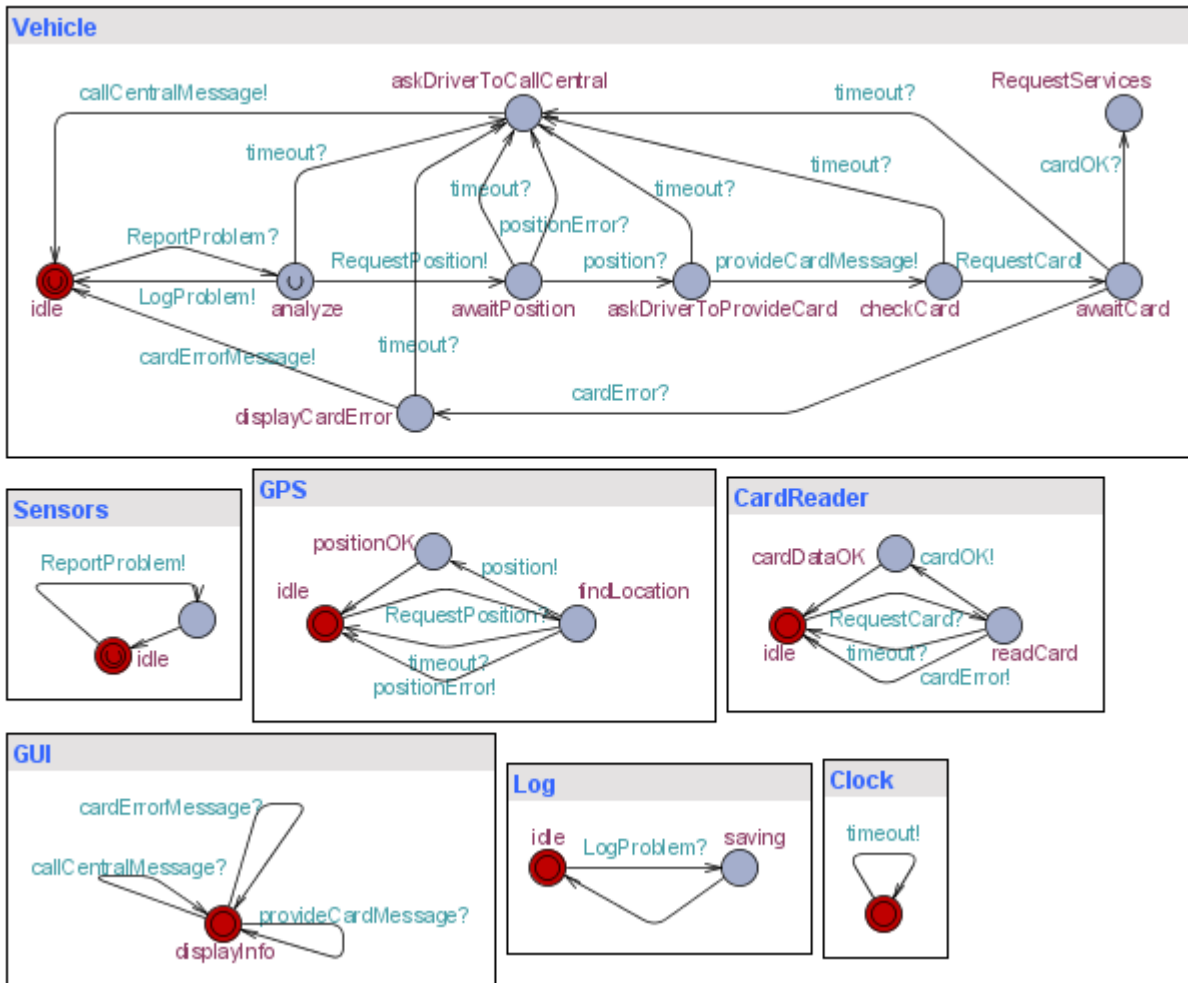


Figure 7-2 State machine of Vehicle component behavior

Component correctness can be tested by formulas checking:

- reachability – it is possible to achieve a goal of a protocol

```
E<< Vehicle.RequestServices
Property is satisfied.
```

- safety – there is no deadlock (apart from the last state)

```
A[] !deadlock || Vehicle.RequestServices
Property is satisfied.
```

- liveness – after both garage and towing approve services, it will eventually reach the last ‘success’ state

```
(GPS.positionOK && CardReader.cardDataOK) --> Vehicle.RequestServices
Property is satisfied.
```

It is also possible to measure the efficiency and qualitative timing parameters of the system. In order to do it, states and transitions in the system above need to be enriched with clocks and time constraints. Model as well as verification conditions are available on CD in file “Vehicle-component.xml”.

7.2. Protocol analysis

Apart from modeling detailed behavior of a component, Uppaal can also analyze protocol already described in brief in chapter 4.2.7. However, protocol completeness requires adding towing service process as depicted in Figure 7-3.

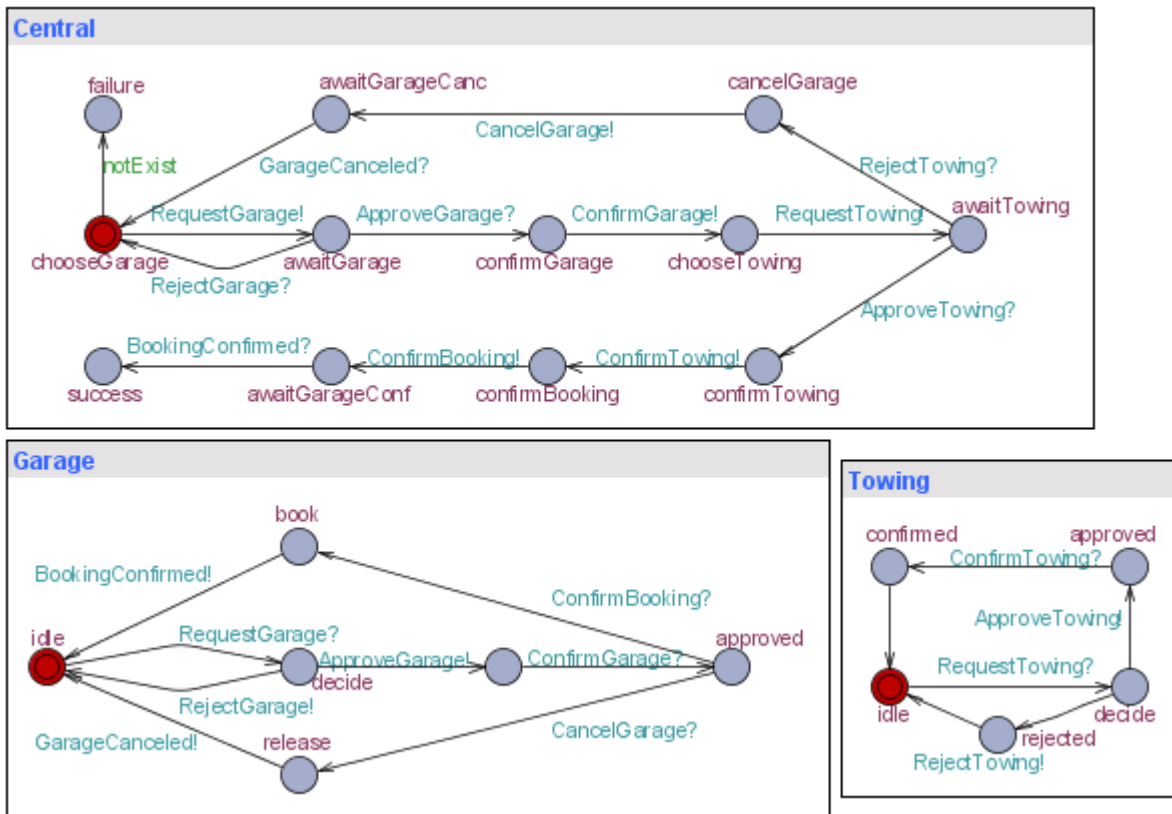


Figure 7-3. State machine of an interaction protocol of reserving garage and towing

From the Central’s perspective, protocol begins in location “chooseGarage” and continues to “awaitGarage” until a garage decides whether to accept or reject request. In case it is accepted, the garage is confirmed, according to a two-phase-commit protocol. Next, a towing service is requested and, depending on a response, the previously reserved booking is either canceled or permanently confirmed. Garage service’s role in the protocol is simpler than Central’s but more complicated than Towing service. It has to be noted that all interactions use a two-way hand shake that ensures that both parties are involved.

Protocol correctness can be tested by formulas:

- reachability – it is possible to achieve the goal of protocol

```
E<> Central.success
Property is satisfied.
```

- safety – there is no deadlock (apart from the last state)

```
A[] !deadlock || Central.success
Property is satisfied.
```


- liveness – after both garage and towing approve services, it will eventually reach the last ‘success’ state

```
(Garage.approved && Towing.approved) --> Central.success
Property is satisfied.
```

Model as well as verification conditions are available on CD in file “protocolForPortG2C.xml”.

7.3. Uppaal conclusions

The two above examples show the suitability of Uppaal to verify properties of interacting parts of a system. Despite its verification efficiency, it seems that UPPAAL has been constructed to handle rather small real-time systems. The lack of abstraction mechanism makes it difficult to design systems in a compositional, rather than synchronizing approach.

Introducing time while modeling big systems may lead Uppaal to exceed memory of the application, when a state space becomes too big. The state explosion problem can be, however, limited with the help of abstraction mechanism [TOU].

Because of its great efficiency, it would be beneficial to adopt Uppaal to SOA standards. However, following improvements would be convenient:

- abstraction mechanism allowing to group and nest processes together in a tree-like structure as shown in Figure 7-4

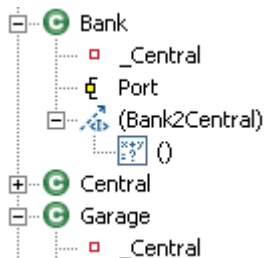


Figure 7-4 Tree-like structure of related elements

- less complicated data passing that could be generated automatically by pointing source and destination locations together with a variable to communicate
- full TCTL logic supporting nesting path quantifiers that could help to construct more complicated verification conditions
- on-the-fly simulator while editing a model would greatly improve model creation if a designer could see immediately how a model reacts
- performance analysis support with a monitor that could gather specified data (like variables) during an automated simulation; random values would be helpful in simulating failures
- interface improvements:
 - display of more than one template in editor allows to have an overview of a model
 - input suggestions for synchronization channels or variable names

- highlighting closely related elements and templates (those that synchronize together)

Without the aforementioned improvements, designing a SOA system in Uppaal is troublesome. In particular due to the lack of concurrency and abstraction in state machines, creating models is not as intuitive as in CPN Tools. Thus, to overcome design problems, a model transformation from CPN Tools to Uppaal, presented in chapter 8, has been pursued. The Petri net compatible with CPN Tools is mapped to a functionally equivalent state machine available to load by Uppaal. As a result it is possible to verify missing liveness properties and completely simulate the model. Having the same system in a closer-to-hardware abstraction is believed to increase the understanding of the model as well as open new possibilities of further analysis or transformations. One can only mention various code generators that are based on the state machine models. Finally, having a model compatible with two specialized tools allows using their either complementing or overlapping mechanisms to verify the system.

8. Model transformation

When talking about transformation of a model, usually three categories need to be taken under consideration:

- syntax – every model has to adhere to its specific environment
- semantics – models should correspond to each other with respect to the behavior
- visualization – models should correspond to each other with respect to their graphical representation

Transformations can also be categorized with respect to start and end products [BSSU]:

- model to model – to create or update a model starting from a higher abstraction
- model to code – to generate machine (platform) specific implementation code
- refactoring – to perform local changes on one model like changing class names, moving a package and so forth

In terms of abstraction level, a transformation can be classified as [BSSU]:

- forward – from higher abstraction to the lower
- reverse – from lower abstraction to higher

Motivated by the verification problems in CPN Tools and modeling issues in Uppaal, a unifying approach has been chosen to combine the best of both tools. In order to have the transformation process verified, it needs not only to be described theoretically, but also implemented. Working algorithm proves the correctness of the algorithm and allows quick testing of different cases. It also allows having a consistent and documented model driven methodology that starts from requirements represented as Petri net and ends in executable code.

Together, CPN Tools and Uppaal seem to cover most of the aspects of a SOA system starting from an abstract model of the orchestration and ending with low-level element details. Both tools are compared in Table 2.

Table 2 Comparison between CPN Tools and Uppaal

| Feature | CPN Tools | UPPAAL |
|----------------|--|---|
| Compatibility | + resemblance to UML activity diagrams, vast theoretical background and research + XML data storage + connection | + resemblance to hardware, threads, code generators + XML data storage + java remote connection |
| Modeling | + full concurrency (forks, joins) - no net initialization + ML data types (tuples), programming support | +/- concurrency only between different templates (processes) + template (process) initialization + arrays, some programming |

| | | |
|----------------------|---|---|
| | <ul style="list-style-type: none"> - only global declarations token based data passing | <ul style="list-style-type: none"> support + broadcast channels + local and global declarations synchronous data passing |
| Timing (performance) | <ul style="list-style-type: none"> - only delays + exporting performance data + performance analysis, monitors, statistic functions | <ul style="list-style-type: none"> + minimal and maximal delay |
| Interface | <ul style="list-style-type: none"> + innovative and adjustable control (context sensitive marking menus, gestures, colors) + possible multiple input devices (mouse, trackball) can be controlled by many users + efficient text filling when creating + auto filling of color type + intuitive hierarchy system (creating, viewing) + groups | <ul style="list-style-type: none"> - traditional, slightly configurable interface (colors) + easy choice of many elements |
| Extensions | <ul style="list-style-type: none"> + SS visualization: Graphviz[GRAP] + TCTL logic: ASK_CTL [DAM] + animation and visualization: BRINTNeY [BRIT] | <ul style="list-style-type: none"> - no extensions but few Uppaal variations: + Cost-Uppaal supports cost annotations of the model and can do minimal cost reachability analysis. This version also has features for guiding the search. [ACAP] + Distributed-Uppaal runs on multi-processors and clusters using the combined memory and CPU capacity of the system [DRAT, DTMC] + T-Uppaal test case generator for black-box conformance testing [TUPP] + Times is a tool set for modeling, schedulability analysis and synthesis of (optimal) schedules and executable code. The verification uses Uppaal [TTMI] |

| | | |
|--------------------------|--|---|
| Model checking | + automatic on-the-fly syntax checking | manual |
| Verification language | + ML, predefined SS, functions, CTL predicates | a variant of TCTL logic |
| State space verification | queries saved in the model - whole SS needs to be generated before checking any expressions one SS optimization - cannot stop SS creation - no counter example generation | queries saved in external file + SS created on the fly when checking expressions + various SS optimizations (over and under approximations, reduction, breadth or depth first) + can stop verification + counter example trace (some, shortest, fastest) |
| Simulation | - only visual - only forward simulation (cannot step backward) - no variable monitor | + both textual and visual + can change previously taken path, updating also progress + variable monitor |
| Bugs | - significant | + almost none |
| License | + free | + free for non-commercial applications in academia, and for private persons. For commercial applications a commercial license is required. |

It is worth to mention that this kind of transformation is experimental and no other similar research is known. The reason for that is that a state machine is actually a subset of Petri net restricted to one input and output from every transition, thus having a sequential rather than concurrent execution scheme. Because of problematic concurrent actions, even a simple Petri net may result in a complex state machine, losing clarity by that process.

8.1. Element transformation

The most problematic in mapping a CPN Tools' Petri net to Uppaal's state machine is the lack of few Petri-net mechanisms like:

- a single state machine does not support concurrency (forks, joins) resulting in much more complicated structures
- no passing tokens
- no hierarchy
- no equivalent for flexible ML color sets (tuples)

Next, a transformation of distinct Petri net mechanisms is presented. All models are signed with their file names that are included on the attached CD.

Despite being a subset of Petri-net, state machines in Uppaal have their differences not only in structure, but also in terminology as presented in Table 3.

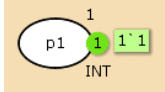

Table 3 Terminology differences between CPN Tools’ Petri net and Uppaal’s net

| CPN Tools | Uppaal |
|---|---|
| place | location |
| transition with arc(s) | edge |
| transition’s guard | edge’s guard |
| transitions time delay (@+) | invariant in location and guard in edge |
| transition’s operations : input (); output (); action(); | edge’s update |
| page | template (process) |

8.1.1. Places/Locations

As can be seen in Table 4, mapping of a place to a location is straightforward with respect to the type of data a place can contain (color of a token). A single state machine in Uppaal cannot have more than one location active at a time. In order to have a consistent behavior of a state machine, multiple tokens on one Petri net are not recommended. Initial node in Uppaal corresponds to a place with a token in a Petri net.

Table 4 Transformation of a single place

| Place in CPN Tools | Location in Uppaal |
|---|---|
|  |  |
| transfl.cpn | transfl.xml |

By default (see user guide in next section), all locations are created as urgent. This allows conveniently sketching a working model without any time flow. No passing time allows using “leads to” verification formula without having to specify time constraints. After the sketch of the system is completed, a designer may generate the state machine again without urgent tags and provide real time constraints to model time related aspects such as performance, timeouts, etc.

Since state machines communicate through channel synchronization that does not contain any data, the token value is kept in each template’s local array “int t[N]” where N is a maximum token size extracted from the model. The token is “passed” between templates with a data passing technique described later. Initial value of a template is extracted from the token and added to local declarations:

```
int t[1]={1};
```

8.1.2. Transitions

In general, there are four types of transitions available in Petri nets:

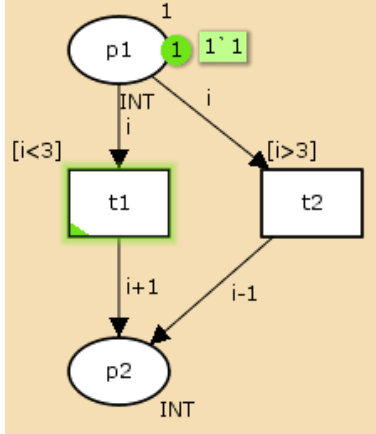
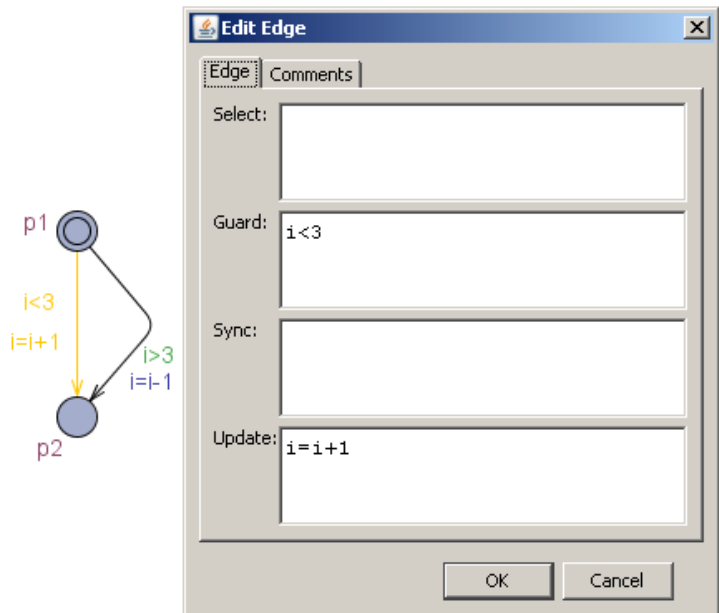
- one-to-one
- one-to-many
- many-to-one
- many-to-many

The proposed transformation supports the first three with a workaround on how to process the last one as well.

a) Simple transitions (1 to 1)

Since transitions in state machine can only have one input and one output edges, for the sake of clarity, they have been completely removed. It can be seen that the token of value 1 has been mapped as a global variable “i” in Uppaal. Similarly, a guard on a transition and the “i+1” function are mapped into a guard and update function. Similarly to places, transitions can have more input and output edges. Also, in case more than one edge is free to be taken, the choice is nondeterministic.

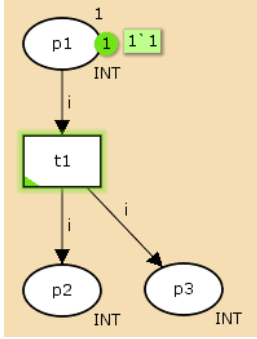
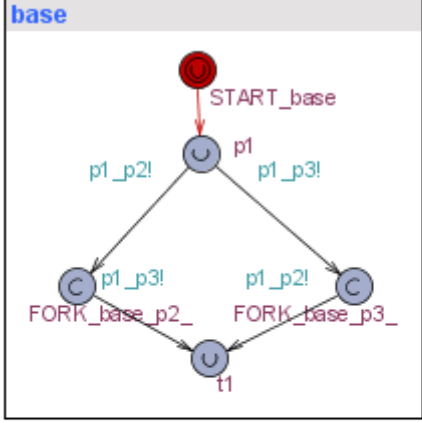
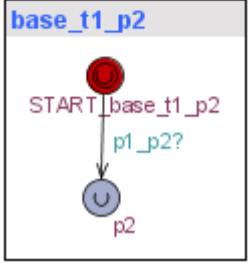
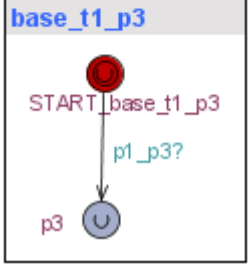
Table 5 Transformation of 1-to-1 transition

| CPN Tools | UPPAAL |
|---|--|
|  <p>The diagram shows a Petri net with two places, p1 and p2, and two transitions, t1 and t2. Place p1 contains one token. Transition t1 has an incoming edge from p1 with weight 1 and an outgoing edge to p2 with weight i+1. Transition t2 has an incoming edge from p1 with weight i and an outgoing edge to p2 with weight i-1. Guards [i<3] and [i>3] are associated with the edges from p1 to t1 and t1 to p2, respectively. The transition t1 is highlighted with a green border.</p> | <p>Declarations: int token[2]={1,0};</p>  <p>The UPPAAL diagram shows two places, p1 and p2, with a transition between them. The edge from p1 to p2 has a guard i<3 and an update i=i+1. The edge from p2 to p1 has a guard i>3 and an update i=i-1. The Edit Edge dialog box shows the following configuration:</p> <pre> Edge Comments ----- ----- Select: Guard: i<3 Sync: Update: i=i+1 </pre> <p>Buttons: OK, Cancel</p> |
| transf2a.cpn | transf2a.xml |

b) Fork (1 - 2..*)

Since it is forbidden to have more than one output edges from a transition in Uppaal, the concurrent states need to be separated into different processes as shown in Table 6. Only then do their local behaviors provide the necessary parallel course of actions.

Table 6 Transformation of a form transition

| CPN Tools | UPPAAL |
|---|---|
|  |    |
| transf3.cpn | transf3.xml |

A Petri net shows a typical concurrency technique where token is multiplied in a fork transition “t1”, so that afterwards both places (“p2” and “p3”) contain one token each. Mapping that to Uppaal requires creating two additional processes (“base_t1_p2” and “base_t1_p3”) that can independently generate a necessary interleaving. One can also notice that the names of both processes and added locations contain name of a parent, involved transition and destination place. This hierarchical notation is believed to ease managing complex models. Since two transitions can fire together, two situations are modeled: one, where place “p2” is reached first, then “p3”, and then an alternative path where “p3” is reached first followed by “p2”. Both paths lead to place “p23” where both places “p2” and “p3” are active. Committed attributes enforce firing of two edges in the same time unit which may make a difference in complex systems.

One can argue that committed location enforces incoming and outgoing event to happen together so there is no purpose of modeling multiple paths. That would be true in case of message based communication, when events correspond to sending messages. However, SOA supports also other interacting techniques like RPC that require direct and lasting communication between peers. It may be possible that one concurrently started process actually initiates another one (also supporting RPC) that is supposed to work in parallel. In this case, it is useful to test all contact schemes to find potential invocation order and also possible deadlocks.

The algorithm allows the concurrency to be much more complicated, as shown in Figure 8-1, leading to multiple concurrent behaviors.

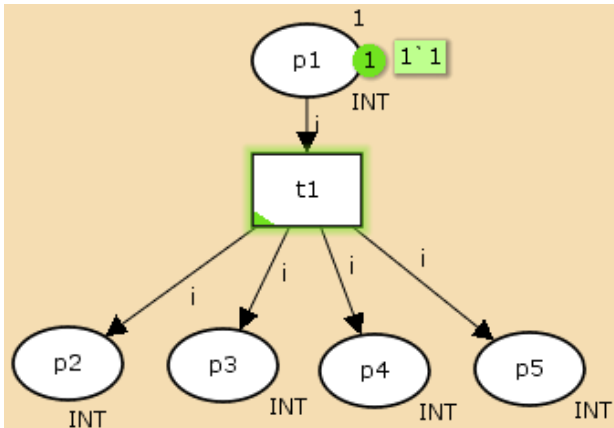


Figure 8-1 Petri net of a beginning of 4 concurrent behaviors

In such a case, all possible invocations might make a state machine model depicted in Figure 8-2 hard to follow. In order to correct clarity, by default all interleaving locations are placed on top of each other. This solution saves precious workspace while preserving complete behavior.

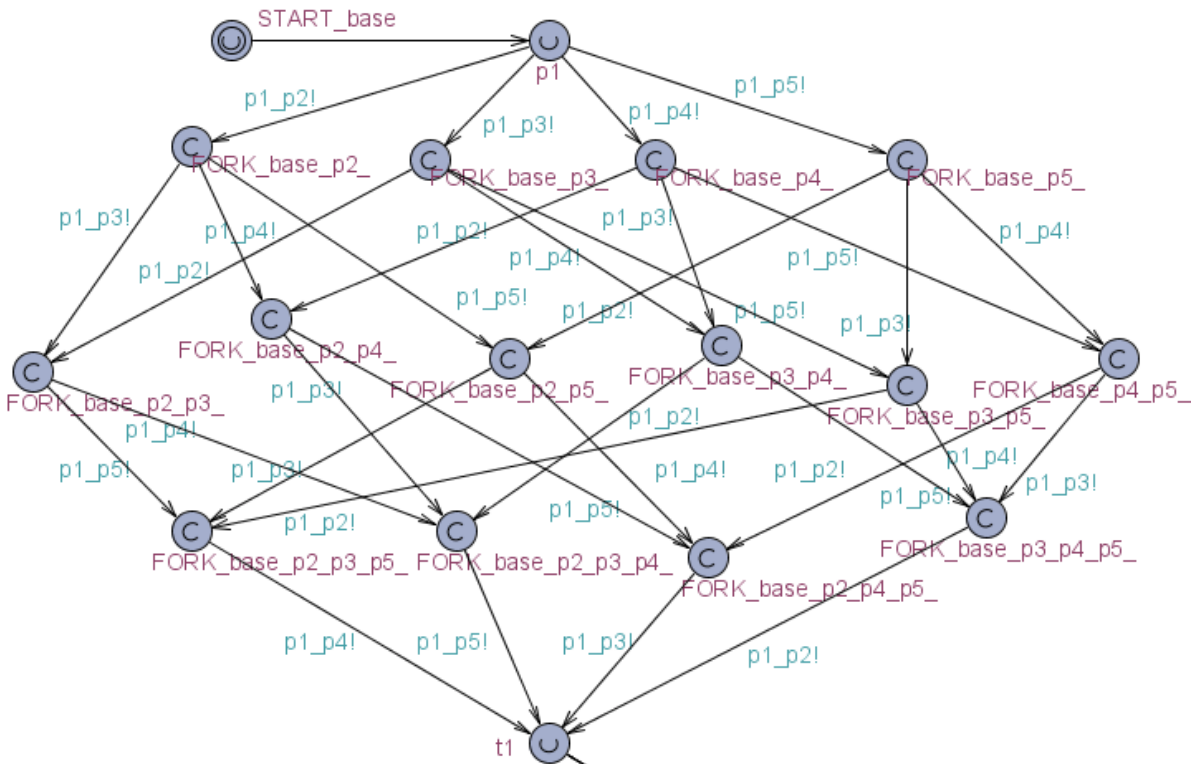
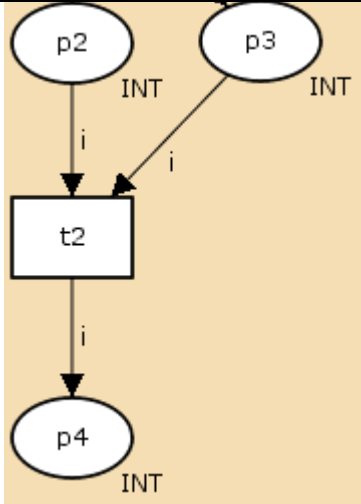
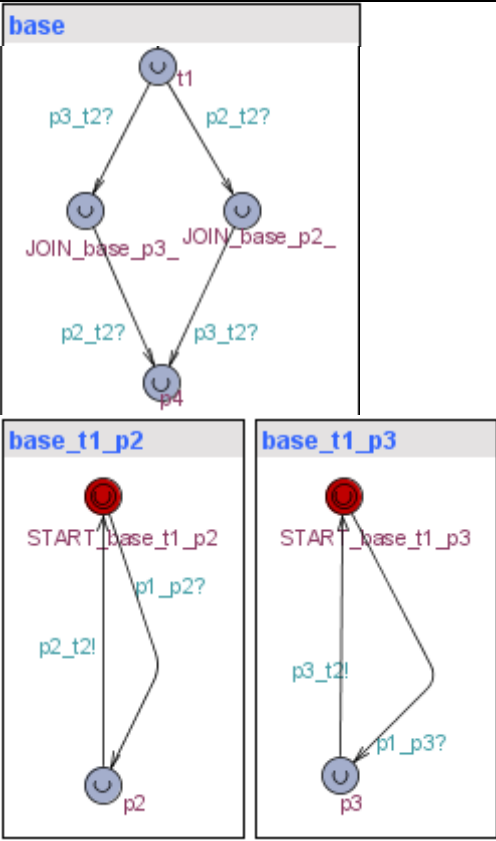


Figure 8-2 State machine of a beginning of 4 concurrent behaviors

c) Join (2..* - 1)

Usually concurrent behaviors unite after fulfilling their tasks as shown in Table 7. Similarly as before, separated processes may be combined in any order.

Table 7 Transformation of join transition

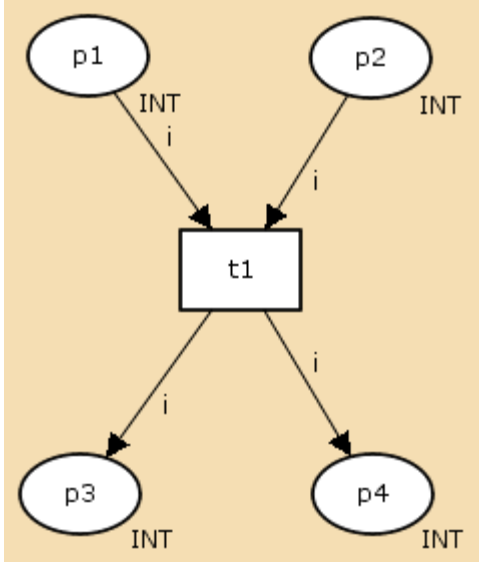
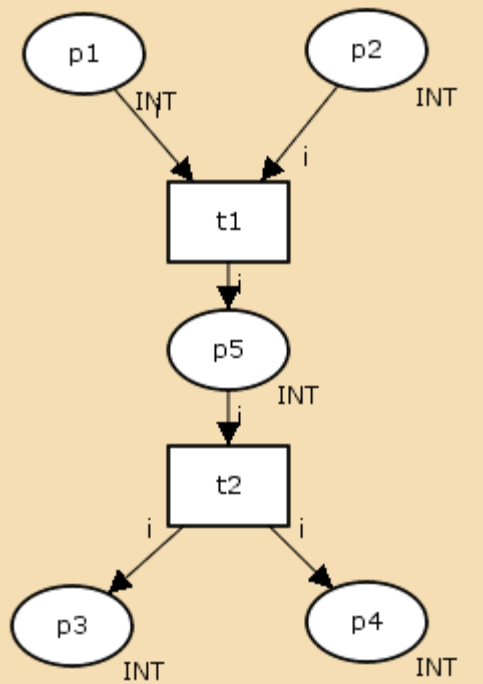
| CPN Tools | UPPAAL |
|--|--|
|  <p>The diagram shows a Petri net with three places: p2, p3, and p4, each containing one token. There are two transitions: t1 and t2. Transition t1 is located between p2 and p3. Transition t2 is located between p2, p3, and p4. Edges connect p2 to t1, p3 to t1, t1 to t2, and t2 to p4. All edges are labeled with the letter 'i'.</p> |  <p>The diagram shows a state machine with three states: 'base', 'base_t1_p2', and 'base_t1_p3'. State 'base' is the initial state and contains a token in place 't1'. Transitions from 'base' lead to 'base_t1_p2' (labeled 'p3_t2?') and 'base_t1_p3' (labeled 'p2_t2?'). Transitions from 'base_t1_p2' and 'base_t1_p3' lead back to 'base' (labeled 'p2_t2?' and 'p3_t2?' respectively). State 'base_t1_p2' contains a token in place 'p2' and is labeled 'START_base_t1_p2'. State 'base_t1_p3' contains a token in place 'p3' and is labeled 'STAR_base_t1_p3'.</p> |
| transf4.cpn | transf4.xml |

A Petri net shows two tokens that merged in transition “t2” (join) to generate one token in place “p4”. As in previous example, state machines need to simulate a possible interleaving of concurrent processes. One has to notice that this is not a precise transformation of the concurrent join. The Petri net join is activated only when both inputs contain tokens. The ‘mapped join’, however, is activated from both inputs sequentially. In the analyzed scenario, this does not matter much because: (1) tokens symbolize a persistent indication of a noticed event; (2) state machines have an always-ending procedure. However, this difference becomes important when timeouts and retransmissions are considered and the half-point needs to be changed before completing join transaction. Those more complicated examples are not supported by automatic transformation and have to be manually adjusted. After uniting, both processes (“base_t1_p2” and “base_t1_p3”) return to their initial points.

d) Advanced transitions (1..* - 1..*)

Because of the computational complexity only many-to-one (join) and one-to-many (fork) transitions are transformed. However, Table 8 shows a simple workaround on how to split one complex transition into two supported ones.

Table 8 Transformation of many-to-many transitions

| Complex transition | Supported transitions |
|--|---|
|  |  |
| transf4c.cpn | transf4c.cpn |

8.1.3. Sub pages

Allowing abstraction refining (possibility to modularize a system) is one of the most important features of CPN Tools and sub-pages are inevitable elements of a bigger system. However, there is no abstraction mechanism in Uppaal, so the mechanism needs to be mapped by synchronizing processes. To increase intuitiveness, the child process has parent's process name right before its own as shown in Table 9. It is believed to help designer managing complex models.

Table 9 Transformation of a sub-page

| CPN Tools | UPPAAL |
|-------------|-------------|
| | |
| transf5.cpn | transf5.xml |

Since Uppaal requires that there can be only one channel that synchronizes at a time⁵, instead of one Petri net action, state machines need two steps (synchronizations) to reach place “p2” on the parent net:

- synchronize and pass token to place “p1” on net “parent_child”
- synchronize and pass token to place “p2” on net “parent”

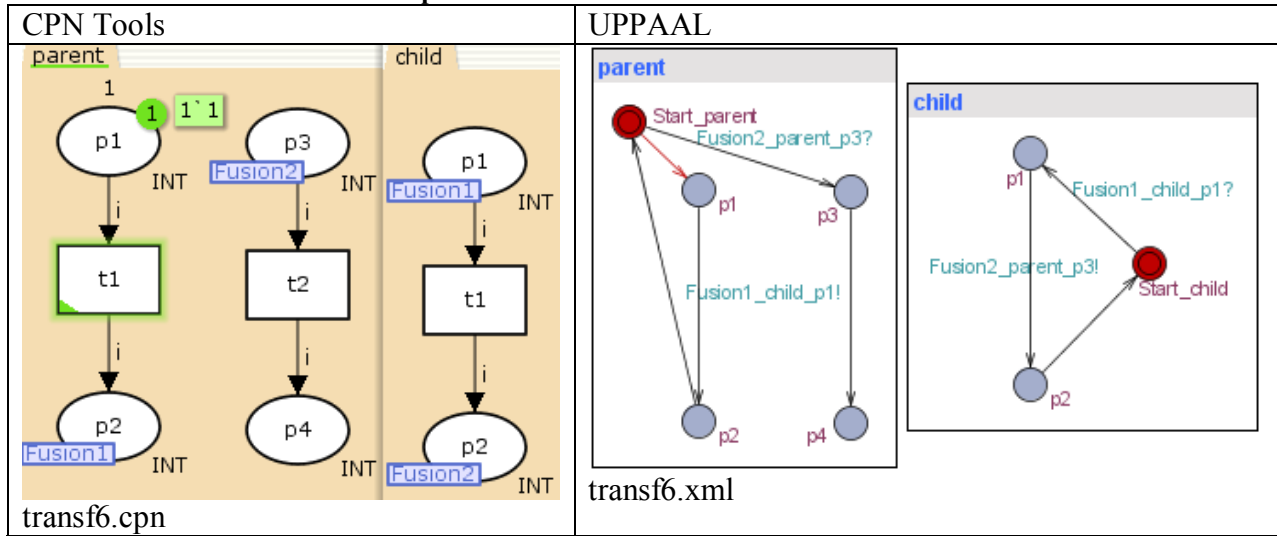
Because of the coding complexity, a state machine is activated not when a token approaches place “p1” on page “parent”, but after it executes transition “t1”. Similarly, the child state machine activates back the parent state machine and afterwards goes back to initial state. Those small discrepancies do not change the functionality of a system, but might change the result of verification formulae (between CPN Tools and Uppaal) which take first states of sub-nets under consideration.

8.1.4. Fusion places

Fusion places allow communicating not only with places in a local page, but also with other networks as shown in Table 10. Because of the algorithmic complexity, fusions of maximum two places are allowed. Fusion places must also belong to different nets.

⁵ apart from, not discussed here, broadcast channels that are used for other purposes

Table 10 Transformation of fusion places



Additionally, contrary to Petri net, Uppaal’s fusion allows a token to activate a state machine only when it is in the proper state to do that (usually initial). A Petri net may accept more than one token, while a state machine cannot.

8.1.5. Data passing

Colored Petri nets have a natural way of passing data in a token. State machines, however, communicate with each other by channel synchronization that does not carry any information. This problem to pass data between processes can be resolved by using a shared, global variable as shown in Table 11.

Table 11 Transformation of data passing

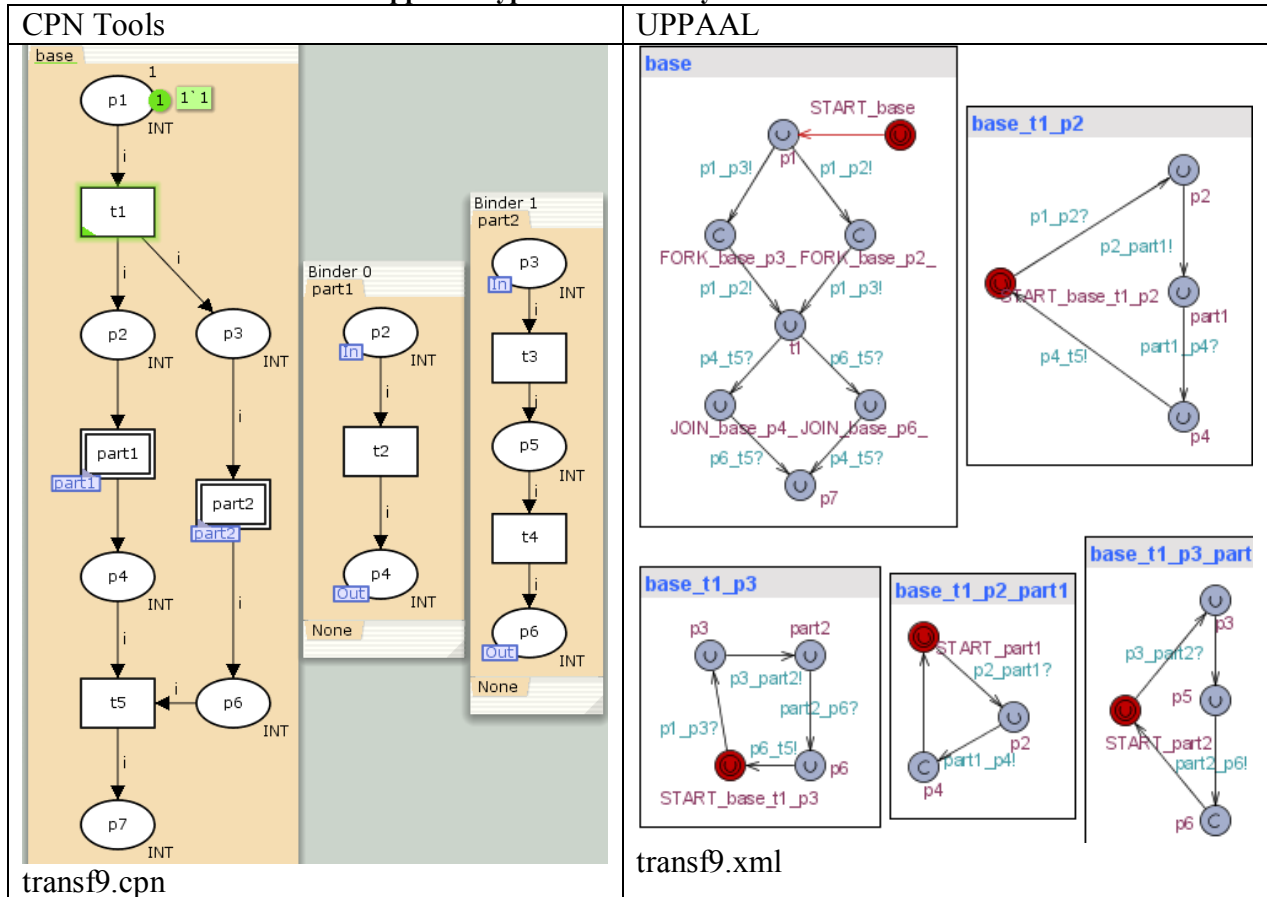
| CPN Tools | UPPAAL |
|---|---|
| <p>The diagram shows two Petri nets, net1 and net2. Net1 (left) has places p1, p2, and p3, and transitions t1 and t2. Place p1 contains 2 tokens, p2 contains 1 token, and p3 contains 1 token. Transition t1 has an outgoing edge to p2 labeled 'i'. Transition t2 has an outgoing edge to p3 labeled 'i'. Net2 (right) has places p2 and p3, and transition t3. Place p2 contains 1 token and has an incoming edge from net1 labeled 'In'. Transition t3 has an outgoing edge to p3 labeled 'i'. Place p3 has an outgoing edge to net1 labeled 'Out'.</p> | <p>Global declarations: int shared[2]; urgent chan p2_t2; urgent chan t2_p3;</p> <p>net1 declarations: int token[2]={2,0};</p> <p>net2 declarations int token[2];</p> <p>The UPPAAL diagram shows the transformation of the CPN Tools Petri net. It consists of two sub-nets: net1 and net1_net2. Net1 (left) starts with a red START_net1 token in a place. It has places p1, p2, t2, and p3. Transitions are labeled with conditions and actions: p1 to p2 is 'token[0]>1', p2 to t2 is 'p2_t2! shared[0]=token[0]', and t2 to p3 is 't2_p3? token[0]=shared[0], shared[0]=0, shared[1]=0'. Net1_net2 (right) starts with a red START_t2 token in a place. It has places p2 and p3, and transition t2_p3!. The transition is labeled 't2_p3! shared[0]=token[0]'. The final place is labeled 'C'.</p> |
| transf7.cpn | transf7.xml |

As a result, a local variable token[0] in “net2” changes its value from 1 to 2. Changing the value of global variable shared to 0 when it is not used anymore reduces the state space. The type that the token may store is an array of integers int token[x] with x extracted by processing the Petri net. Therefore, string and boolean data need to be adapted to int in a Petri net. Since Petri net passes data in a token and state machine in a local array “int token[]”, the condition on transition t1 needs to compare token[0] with 1. For simplicity, only integer values are automatically passed, stored or compared.

8.1.6. Complete concurrency

Because of the complex nature of concurrency it is recommended that parallel processes do not interact with each other. In those cases, a Petri net can be safely divided into separated templates (processes), as shown in Table 12.

Table 12 Transformation of a supported type of concurrency



As can be seen Table 12 there are two concurrent sub-pages (part1 and part2) in a Petri net. After transformation, process “base” is significantly simpler, delegating the activities to other processes. And so, two additional processes (“base_t1_p2” and “base_t1_p3”) are created to model concurrent behavior after transition “t1” and before transition “t5”. Each of them executes sub-processes: “base_t1_p2_part1” and “base_t1_p3_part2” before merging again in transition “t5”. Committed locations (“FORK_base_p2” and “FORK_base_p3”) ensure that both locations are activated in the same time unit. This technique is not necessary in a simple example like this one, but without that, transformation of complex system may create deadlocks. As can be noticed, processes “base_t1_p2_part1” and “base_t1_p3_part2” have also committed locations (“p4” and “p6”) that decrease the state space of the whole model.

8.1.7. Declarations

CPN Tools supports not only basic types like string, integer or boolean, but is based on a flexible ML system that allows combining many different types into tuples. On the other hand, the only basic types Uppaal provides are: int, boolean and clock. An extensive comparison is presented in Table 13.

Table 13 Comparison between syntax

| Meaning | CPN Tools (ASK-CTL) | UPPAAL |
|---------------------------------|---|--|
| constant | val MAXGARAGE=3; | const int MAXGARAGE=3; |
| integer variable | colset INT = int; var i:INT; | int i; |
| boolean variable | colset BOOL = bool; var b:BOOL; | bool b; |
| other type variables | colset STRING = string; var s:STRING; | - (string is not supported) (workaround) int s[length(STRING)]; |
| type declaration | colset ID=INT; | typedef int ID; |
| timed variables | colset c:INT timed; | clock c; |
| tuples | colset DEP =INT; colset CTRL_ST=product INT*INT*DEP; | - not supported |
| | var ctrl: CTRL_ST; | - not supported (workaround) int ctrl[3]; |
| synchronization between nets | ports, sockets, fusion places | channels |
| comment | (* <text> *) | // <text> |
| procedures | fun <name> (<parameters>):TYPE = < statements> ; | < TYPE > <name>(<parameters>) { < statements> [return <expression>;] } |
| logical AND | andalso | &&, and |
| logical OR | orelse | , or |
| logical NOT | not | ! |
| equality | <expression1>=<expression2> | <expression1>===<expression2> |
| condition | if <expression> then <statement1> [else < statement2>] | if (<expression>) <statement1>; [else < statement2>;] |
| | | |

Automated transformation supports only constant values. All other declarations are copied as comments into Uppaal declarations waiting for manual adjustment.

8.1.8. Verification expressions

Despite using similar logic originating from TCTL, there are significant differences. CPN Tools does not support it directly, but only after activating the additional library ASK_CTL. On the other hand, Uppaal prohibits nesting of the modal operators. Because of the different structure of models and usage of formulas, they are not transformed automatically. The verification formulas' grammar is presented in Table 14.

Table 14 Comparison between supported logical formulas

| Meaning | CPN Tools (ASK-CTL) | UPPAAL |
|--|---------------------------|-----------------------------|
| both A and B are true | AND (A,B) | A && B |
| either A or B are true | OR (A,B) | A B |
| for all paths in all reachable states, A is true | INV (A) | A[] A |
| for all paths A becomes eventually true | EV (A) | A◇ A |
| A holds for all states on at least one path | ALONG (A) | E[] A |
| it is possible to reach a state where A holds | POS(A) | E◇ A |
| if A holds, then eventually (for all paths) B will also hold | --- | A --> B, A[] (A imply A◇ B) |
| there exist a deadlock | ListDeadMarkings ()◇empty | A[] deadlock |

8.2. Algorithm

The general idea is to explore a Petri net in a breadth-first fashion, extracting parts that can be mapped into state machine elements and binding them together with synchronization. A token can be referred to as an active location in a state machine with a token value stored in a local variable in an active process. Token's parent page is also treated as a root reference point for all created processes, so it should usually be placed in the main page. Due to the erroneous liveness verification in CPN Tools, the transformation algorithm has been tested by comparing simulations of original and result models.

Data structures

- TODO – stores arcs to inspect
- INITIALS – stores references to all templates with their initial locations' ids
- GATEWAYS – stores references to all templates with the transitions that lead to them
- LOCATIONS– stores all already created locations
- TEMPLATES– stores all already created templates
- EDGES – stores currently saved edges between locations

Used variables:

- Arc P2T – stores a temporary arc from a place to a transition
- Arc T2P – stores a temporary arc from a transition to a place
- Transition T – stores a temporary transition
- Place P – stores a temporary place
- LOCATION – stores a reference to temporary template

For the sake of algorithm clarity, types have also convenient methods:

- Arc
 - placeEnd – returns a place that is connected by the arc
 - transitionEnd – returns a transition that is connected by the arc
- Transition
 - otherEnd (P) – contains a place connected to P by either in, out or in/out relation
 - IsSubpage () – return boolean value whether a transition is a sub-page
- Place
 - IsOutputPort – return boolean value whether place is an outgoing port
 - IsIOPort – return boolean value whether place is an I/O port
 - IsFusedPort – return boolean value whether place is a fused port

Input: lists of Petri net(s) elements (PLACES, TRANSITIONS, ARCS, PAGES)

Output: list of state machine(s) elements

FIND_INITIAL_TOKENS()

```
if INITIALS.size > 1 then EXIT ("Model cannot contain more than one initial token!")
```

```
while (TODO.size > 0)
```

```
    P2T = TODO.getFirstElement
```

```
    TODO.removeFirstElement
```

```
    T = FIND_TRANSITION (P2T)
```

```
    P2T.guard = T.guard // transfer constraints from transition to arc
```

```
    if ( T.IsSubpage )
```

```
        then CREATE_SUBPAGE (P2T, T, TODO)
```

```
    else if (T.inputArcs > 1) AND (LOCATIONS contains all places of T.inputArcs)
```

```
        then CREATE_JOIN (P2T, T, T.inputArcs, TODO)
```

```
    else if (T.outputArcs > 1)
```

```
        then CREATE_FORK (P2T, T, T.outputArcs, TODO)
```

```
    else if (T.hasOutputArc) then
```

```
        P = FIND_PLACE (T.outputArc)
```

```
        if ( P.isOutputPort )
```

```
            then CREATE_OUTPUT_PORT (P, P2T, T.getOutputArc, T, TODO)
```

```
        else if ( P.isIn-OutPort )
```

```
            then CREATE_IN-OUT_PORT (P, P2T, T.getOutputArc, T, TODO)
```

```
        else if ( P.isFusedPort )
```

```
            then CREATE_FUSED_PORT (P, P2T, T.getOutputArc, T, TODO)
```

```
        else CREATE_COMMON_PORT (P, P2T, T.getOutputArc, T, TODO)
```

```
    end if
```

```
// create a connection to a place on the same template
```

```
CREATE_COMMON_PORT (P, P2t, T.getOutputArc, T, TODO)
```

```
FIND_OR_CREATE_LOCATION (LOCATIONS.get(P), P)
```

```
SAVE_ARC (P2T.placeEnd, T2P.placeEnd)
```

```
ADD_OUTGOING_ARCS (P, TODO)
```

```

// create a connection to an input/output place in another template
CREATE_IN-OUT_PORT (P, P2T, T.getOutputArc, T, TODO)
    FIND_OR_CREATE_LOCATION (LOCATIONS.get(P), P)
    // add arcs on parent template
    ADD_OUTGOING_ARCS (P, TODO)
    SAVE_ARC (P2T.placeEnd, T.getOutputArc.placeEnd)
    T = transition on the other side of I/O port
    // create location on the other side of I/O port
    FIND_OR_CREATE_LOCATION (T.parent, T.otherEnd(P))
    // connect new location
    SAVE_ARC(T, T.otherEnd(P))
    // connect back to initial place
    SAVE_ARC (T.getOutputArc.placeEnd, initial place for T.getOutputArc.placeEnd)
    ADD_OUTGOING_ARCS (T.otherEnd(P), TODO)

```

```

// create a connection to an output place in parent template
CREATE_OUTPUT_PORT (P, P2T, T.getOutputArc, T, TODO)
    TEMPP=ORIGP;
    do
        // find a transition that binds place P
        for each ITEM in TRANSITIONS
            if (ITEM.binds (TEMPP)
                // proceed to bound place
                TEMPP=GETPLACE (T.getBoundPlace)
                BREAK
            end if
        while (TEMPP.isOutputPort)
        if (T.leadsToAnotherTemplate )
            then LOCATION = (find another template)
            else LOCATION=FIND_OR_CREATE_TEMPLATE(child, parent)
        // add arc to last place on child page
        SAVE_ARC (P2T.placeEnd,P)
        // add arc to place on parent page
        SAVE_ARC (T, TEMPP)
        // connect the last place on child page to initial place
        SAVE_ARC (P, initial place of P)
        ADD_OUTGOING_ARCS (P, TODO)

```

```

// returns a place connected to an arc
FIND_PLACE (ARC)
    for each ITEM in PLACES do
        if ARC.IsConnected (ITEM) then
            return item

```

```

// recursively creates connections to all outgoing places
RECURRING_FORK()

```

```

// connects a transition to places from all output arcs (from different templates)
CREATE_FORK (P2T, T, T.outputArcs, TODO)
  RECURRING_FORK()
  for each ITEM in T.outputArcs do
    // create a new template
    LOCATION = FIND_OR_CREATE_TEMPLATE (child, parent)
    GATEWAYS.add (LOCATION, T)
    // add initial place
    FIND_OR_CREATE_LOCATION (LOCATION, "initial")
    // add first place on template
    FIND_OR_CREATE_LOCATION (LOCATION, ITEM.placeEnd)
    SAVE_ARC (local initialPlace, ITEM.placeEnd)
    ADD_OUTGOING_ARCS (P, TODO)
  end

// recursively creates connections for all incoming places
RECURRING_JOIN()

// connects places from all input arcs (and different templates) into one place
CREATE_JOIN (P2T, T, T.inputArcs, TODO)
  for each ITEM in ARCS do
    if (ITEM.transitionEnd == T)
      then P = getPlace (ITEM)
    RECURRING_JOIN (parameters)
  for each ITEM in T.inputArcs
    P = GET_PLACE (ITEM.placeEnd)
    SAVE_ARC (P, initial place of P's template)
  end
  // add joined places arcs to processing
  ADD_OUTGOING_ARCS (P, TODO)

// finds or creates a location (2nd argument) in a template (1st argument)
FIND_OR_CREATE_LOCATION (TEMPLATE, LOCATION)

// creates a new template with a name and combined by both arguments
FIND_OR_CREATE_TEMPLATE ( NAME, PREFIX)

// adds all outgoing arcs from place P to list TODO
ADD_OUTGOING_ARCS (P, TODO)
  for each ITEM in ARCS do
    if ARC.IsConnected (ITEM) AND NOT ARC.wasAdded
      then TODO.add (ITEM)

// adds an initial place in a template in 1st argument with a name of 2nd argument
ADD_INITIAL_PLACE (LOCATION, PLACE)

```

```

// saves an arc between 1st argument and 2nd argument to EDGES
SAVE_ARC (P1, P2)

// create a connection to a place in child template
CREATE_SUBPAGE (P2T, T, TODO)
    ORIGINAL = T
    FIND_OR_CREATE_LOCATION (
        LOCATIONS.getTemplate(P2T.getConnectedPlace),
        T
    )
    SAVE_ARC (P2T.getConnectedPlace, T) // connect to a place
    // loop to get to a lowest abstraction level in a cascade of input/output ports
    do
        P = T.getOtherEnd (P)
        for each ITEM in ARCS do
            if (ITEM.beginsWith(P) )
                then T = FIND_TRANSITION (ITEM)
        while (T.isSubPage)
        if (T.leadsToAnotherTemplate )
            then LOCATION = (find another template)
            else LOCATION=FIND_OR_CREATE_TEMPLATE(child,parent)
        ADD_INITIAL_PLACE (LOCATION, T)
        FIND_OR_CREATE_LOCATION (LOCATION, P)
        SAVE_ARC (new initial place, P)
        ADD_OUTGOING_ARCS (P, TODO)

// returns a transition connected to an arc
FIND_TRANSITION (ARC)
    for each ITEM in TRANSITIONS do
        if ARC.IsConnected (ITEM) then
            return item

// adds to the INITIALS list all places that contain tokens
FIND_INITIAL_TOKENS ( ):
    for each item in the list PLACES do
        if the item contains token then
            INITIALS.add (the item)

// exits application with a reason displayed
EXIT (REASON)

```

Apart from the algorithms, there is some detailed supplementary information:

1. Initial location

In current version of the algorithm, multiple initial tokens are forbidden. However, some tokens in Petri nets are used as iteration counters, semaphores or have other local tasks. One can replace those mechanisms with variables and focus on the main logic. The token value needs to be replaced by the corresponding local variable(s) that will be updated whenever the

token value changes. The only type supported is 'int', so one can represent strings as a list of chars (int[]) or code the values as integers:

Table 15 Possible conversion between string and int data type

| int | string |
|------|--------------|
| 0 | "" |
| 1 | "serviceReq" |
| 2 | "depUnres" |
| 3 | "depRes" |
| etc. | |

2. Template and location creation

Whenever a template or location is created, there is a check to find whether it has already been created. That prevents unnecessary creation of elements in case of loops. Names of templates and locations are copied from Petri net places, if possible. For some additional elements that have to be added (forks, sub-nets, and initial locations), a name related to the element is chosen. To resemble the original structure of the model, locations are placed in the position extracted and transformed from the Petri net (for direct mappings) or with the position of the nearest element (for added locations).

3. 'todo' list

The list contains all the arcs that need to be investigated when transforming the model. As locations are added, new arcs are added for every location and once they are, they are marked as 'added' to prevent traversing the same arc again. Hence the algorithm works in a breadth first fashion starting from the initial node with regards to the join node that requires all incoming locations to be processed before proceeding.

4. Edges

Only unique edges are added to the model so as to avoid duplication. Name of the channel in a synchronized edge consists of a source and target elements' names.

5. Connection graphs

Join and fork nodes generate connections graphs that represent all possible occurrences of communication. Those graphs become quite big in terms of space and in order not to clutter the model, all intermediate nodes are stored in the very same place. If necessary, a designer might relocate them afterwards.

6. Channels

Synchronization channels are created and added to declarations automatically with a name of the elements they bind. Since a name of a channel is created separately for both communicating templates, it is required that the binding locations (in, out and IO) have the same names on both templates.

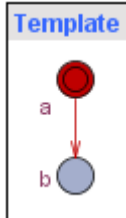
7. Guards and assignments

Guards and assignments on edges are constructed from the transitions that the originating arcs connect to. However, because of the complexity, there is only a mechanism transforming

simple expressions ($=, <, >, ==, <=, >= 0$). However, by default, even unrecognized inscriptions are attached to edges that allow Uppaal to find and highlight them.

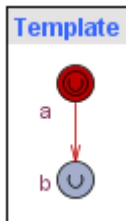
8. Urgency

The algorithm has a default parameter “urgency” set to true, that creates all the locations urgent. Detailed parameter description is in the chapter “CPN2Uppaal manual”. This feature is convenient when verifying a draft model. Checking liveness properties requires specification of time constraints and without that, even a simple model verification query result is not useful, since there is also a possible execution, where model remains in for an indefinitely long time:



```
Template.a --> Template.b  
Property is not satisfied.
```

Before all the time constraints are added, setting all locations to urgent can reveal estimated results:



```
Template.a --> Template.b  
Property is satisfied.
```

9. Labels

Contrary to Uppaal, CPN Tools allows a name of a label to contain multiple lines and spaces. Thus, the transformation changes all whitespaces to characters:”_”.

8.3. User guide

Usage:

```
Java -jar CPN2Uppaal.jar <[path]inputFile> <[path]outputFile> [option1  
option2 ...]
```

Where `inputFile` is a CPN Tools compatible Petri net and `outputFile` specifies a Uppaal compatible data.

Options:

Following options allow adjusting some of the transformation behavior of creating an Uppaal’s compatible state machine:

- `urgency=<true/false>` – enable urgency in all location apart from some that are committed; allows to quickly sketch a model and verify liveness properties (default value is true)
- `guards=<true/false>` – enables or disables guards on generated arcs (default value is true)
- `synch=<true/false>`– enables or disables synchronizations on arcs (default value is true)
- `assign=<true/false>`– enables or disables all assignments (including data passing) on arcs (default value is true)
- `zoom=N` – magnifies the input model by a natural number N (default value is 1); negative values will reverse a model in both X and Y axis

8.4. Limitations

Due to a restrictive character of state machines, transformation algorithm required constant tradeoffs between expressiveness to transform all possible patterns and effort to program them. In order to simplify algorithmic complexity, following limitations to input Petri net have been introduced together with possible workarounds:

- there is no more than 1 initial token on one page
- fusion places and treated as communication channels and must be unidirectional (bidirectional transformation may work depending on neighboring connections), binding only 2 places that belong to different nets. In addition, sink fusion place must not have any outgoing arcs.
- join and fork transitions cannot lead directly to ports (both in and out) or fusion places; outgoing arcs should not have incremental assignments;

This limitation can be omitted by additional place between transitions and forbidden places.

- sub-page cannot lead directly to fused places

This limitation can be omitted by additional place between transitions and forbidden places.

- only one-to-many and many-to-one transitions are acceptable
- fused place's requirement that incoming fusion must be connected to join transition merging also with non-fusion arc from other part of the system
- pattern matching is also not supported; one can manually add guards if necessary
- complex arc inscriptions might generate problems since the algorithm flattens formulas to generate updates and guards
- there is a variable inscription on every arc, even those connected to sub-pages

8.5. Application details

Most of the difficulties of algorithm implementation were connected with the huge amount of input elements that had to be consistently transformed, regardless their possible relationships. Additionally, the experimental nature of the transformation of specific structures did not allow planning the implementation into easily manageable modules. There were also some issues connected with a lack of documentation for data format in Uppaal together with its unusual reactions to incorrect data, as explained later in section.

8.5.1. Code structure

The main class that initiates all other objects, as well as contains the transformation algorithm is `Transformer`. After processing user input parameters, it activates class `XMLParser` to parse the input XML data from CPN Tools. While the elements are parsed using `org.w3c.dom` library key, elements represented by objects (`Place`, `Arc`, `Transition`, `Fusion`, `Page`) of a Petri net are added to `Transformer`'s vectors: `places`, `arcs`, `transitions`, `fusions`, `pages`. Afterwards an overview of the gathered data is displayed and processed by the algorithm described in the next section. The algorithm adds consequent children to the XML data compatible with Uppaal. After all reachable elements are processed, the final XML data is displayed and written into a file, if it was specified, or into a file with the name of the input file, but with extension `.xml`. For the sake of brevity, full code listing, together with a class diagram, are on attached CD.

8.5.2. Uppaal's data format

Uppaal stores its model in XML with the root component (`<nta>`) that contains global declarations (`<declaration>`), a list of templates (`<template>`) and system declarations (`<system>`). Every template contains children:

- name of the template (`<name>`)
- local declarations (`<declaration>`)
- list of locations (`<location>`)
- initial location (`<init>`)
- list of transitions (`<transition>`)

Each location has attributes storing a unique reference number (`id`) and coordinates (`x,y`). It also contains children:

- visible name of the location (`<name>`) together with its coordinates
- optional invariants (`<invariant>`)
- optional urgency tag (`<urgent>`)
- optional committed tag (`<committed>`)

Each transition contains children:

- reference to source location (`<source>`)
- reference to target location (`<target>`)
- optional synchronization label (`<label kind="synchronisation">`)
- optional assignment label (`<label kind="assignment">`)
- optional guard label (`<label kind="guard">`)

It is worthwhile to mention to anyone that would like to work with Uppaal data that the order of children makes a big difference in how the data is interpreted. For example, even though Uppaal does not display warnings, when a transition contains its initial location tag after one (or more) transitions, it simply does not display any transitions at all. Similar undefined results appear when a location has urgency or committed tags placed before invariants. Transitions that point to a location outside a local template may be displayed, pointing to nothing, or not displayed. Additionally, the maximum length of a template name is limited to

64 characters and despite being a long string for a name, it was sometimes not sufficient to represent attached hierarchy information.

8.6. Methodologies

Following sections present analysis of models created before in CPN Tools and transformed to Uppaal. For the sake of brevity, not all of the processes are shown, but all of them are on the attached CD.

8.6.1. 5-step collaboration goal sequence

In order to verify liveness properties and prepare the model for code generation, the model has been transformed to Uppaal. Because of some algorithm limitations, the model received a few modifications with respect to replacing string type data of failure details to int type (list of modifications is in Table 16) and adding additional intermediate places wherever join or fork transitions lead directly to output or bidirectional ports or fusion places.

Table 16 List of inscription modifications in GSM transformation

| location | before modification | after modification |
|---|---|--|
| initial token | 1`(1,1,"") | 1`(1,1,0) |
| declarations | colset CTRL_ST=product INT*INT*STRING; | colset CTRL_ST=product INT*INT*INT; |
| declarations | var s:STRING; | var s:INT; |
| declarations | colset DEP =STRING; | colset DEP =INT; |
| arc inscription in page reqServ | (ctrl,"depUnres") | (ctrl,1) |
| arc inscription in page reqServ | (ctrl,"depRes") | (ctrl,2) |
| arc inscription in page goalSequence | (ctrl,"depRes") | (ctrl,2) |

Adding additional elements has not changed page number (16) but increased other elements number: places (from 135 to 167), arcs (from 217 to 281), and transitions (from 83 to 115). As a consequence, the state space has been doubled:

State Space

Nodes: 2590
Arcs: 5396
Secs: 3
Status: Full

Scc Graph

Nodes: 707
Arcs: 2721
Secs: 0

Apart from the additional elements, the modified model follows the same idea and for the sake of brevity it is not described in the report. Conformance of the two models has been

checked by simulation and verification queries that provided the same results despite increased state space.

The result Uppaal state machine reveals a number of 51 templates, 327 locations and 442 transitions. For the sake of brevity, only few most interesting parts are described. Complete models are on the attached CD: “GSM-extended.cpn” (input Petri nets) and “GSM.xml” (result state machines).

Despite the increased clarity achieved by not displaying some information, the models may be still too complex to analyze on paper. However, the figures should give the impression on how the logic in Petri net corresponds to the one of state machine. It is recommended that careful analysis of complete models on the attached CD should be done with help of Uppaal or CPN Tools, which greatly support readability by zooming, relocation or highlighting related elements.

After some visual modifications to increase clarity, the first and main page of the model is shown in Figure 8-3.

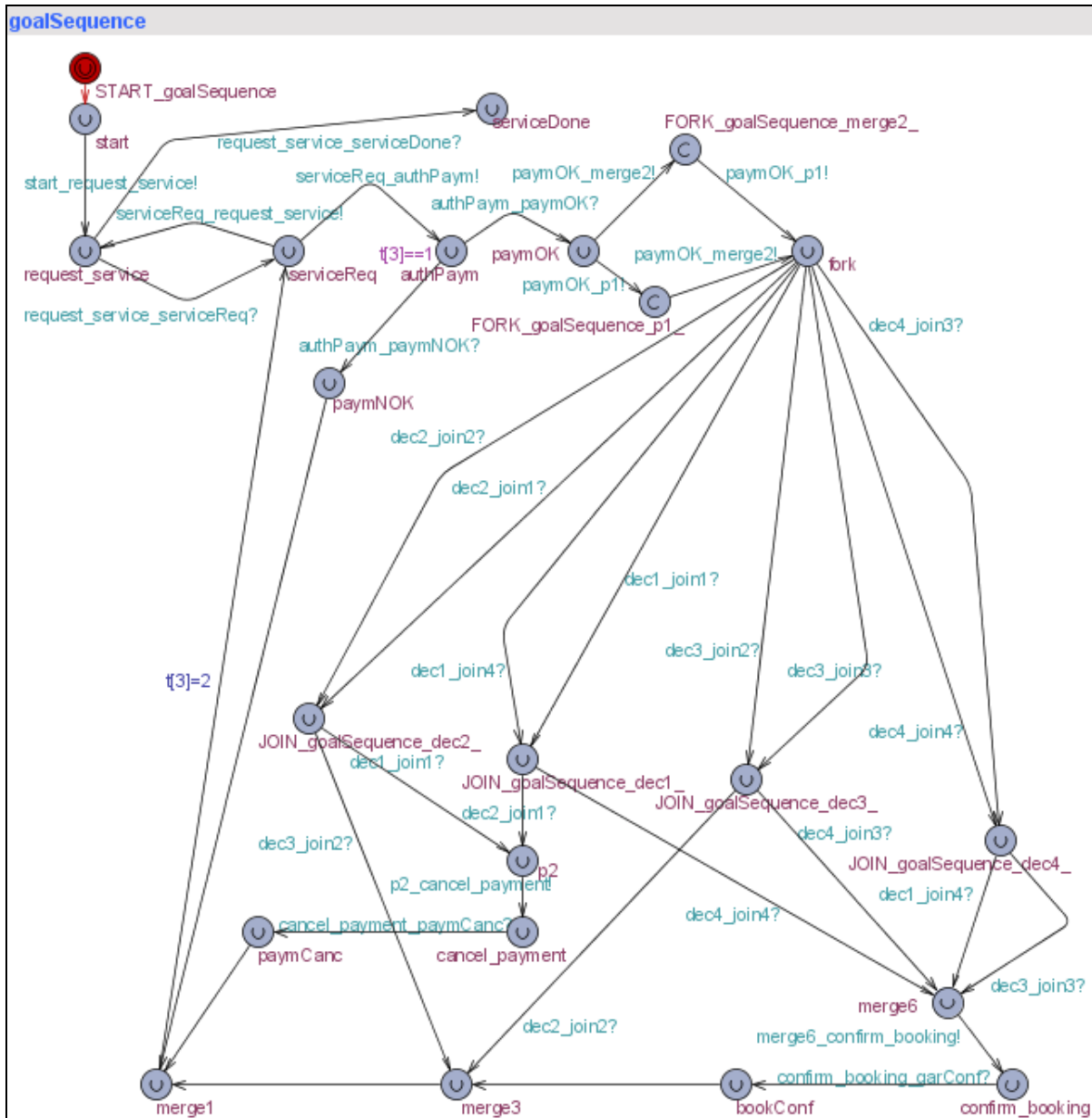


Figure 8-3 First and main template of a transformed model

One can see a visual resemblance between the Petri net and state machine in locations at the top and the bottom of Figure 6-8. However, the similarity of main logic ends at the location “fork”, where two concurrent processes are initiated. Similarity starts again from locations “p2”, “merge3” and “merge6” after all concurrent processes are merged.

For the sake of clarity, edges do not have any update assignments. Assignments are mainly used to have a flow of data between synchronizing elements, but tend to clutter a model with textual representation of transferred data. The only assignment that is visible is between location merge1 and serviceReq is “t[3]=2” with a purpose of forwarding the control through “requestService” to final “serviceDone”. Complementing the control flow mechanism, location “authPaym” has an invariant “t[3]==1” that is set when a request initially comes from process “requestService”. Because of the lack of string type, number 1 is equivalent to

original “depUnres” whereas number 2 means “depRes”. One can also notice that those values are not compared with a variable in a token, as in Petri net, but with a local variable $t[3]$ that represents the token data. Identifier 3 is deduced algorithmically from the original assignment “(ctrl,2)”. Since variable ctrl is of a tuple type of three integers (INT*INT*INT), the additional number 2 at the end needs to stand on the fourth position of the array t (short for token).

Since the main logic has been separated at the fork location, it is also interesting to show how those two branches look. Both branches are depicted together in Figure 8-4.

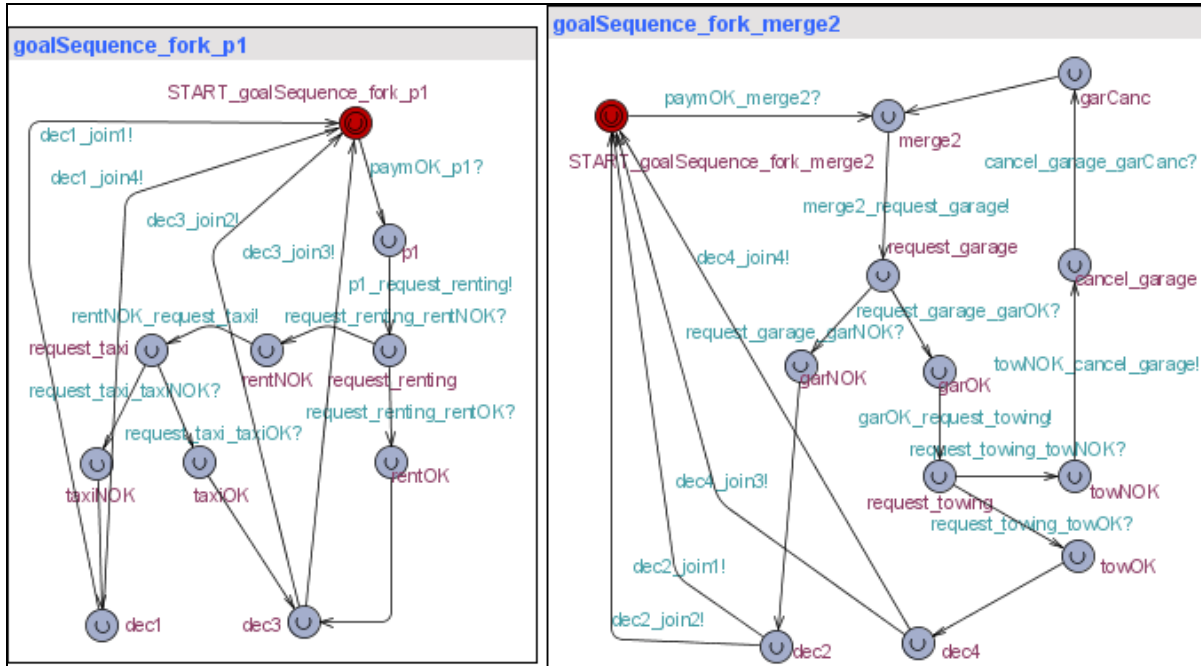


Figure 8-4 Concurrent processes that begin at transition “fork”

Both templates in Figure 8-4 show most of the missing logic from Figure 8-3. One can notice templates’ names that suggest the hierarchy and structure of processes. Name “goalSequence_fork_p1” should be interpreted as process “p1” that begins after triggering transition “fork” in process “goalSequence”. This artificially introduced hierarchy allows not only managing large models, but also uniquely identifies processes. One should know that contrary to IDs in CPN Tools, a name is the only information to differentiate a template in Uppaal and names cannot repeat. For the sake of clarity, channel names are created from the nearby location’s and transition’s name. This tradeoff can however be problematic in models where two different templates have identical places and transition names, as they might interfere with each other through the same channel name. It is therefore required to have unique transition names in the model.

Template “goalSequence_fork_p1” arranges activities connected with reserving renting and taxi. Two of their final locations: “dec1” and “dec3” have two arcs, each back to initial place. Those arcs correspond to a choice of join transitions for both of them in the input Petri net. Template “goalSequence_fork_merge2” arranges garage and towing together with the loop to cancel a garage if necessary.

“d” is compared with a constant value MAXTOW and transitions: “reject towing” and “contact towing” to distinguish whether UDDI registry contains suitable towing services.

Verification has found another mean infinite loop in the characteristic dependency mechanism. It revealed a problem with transforming input/output places. Contrary to fused places that are activated together in Petri nets, state machine needs to have two communication channels to provide bidirectional communication and to introduce a loop, where control flow can be infinitely passed between both processes. The possible resolutions of the problem are:

- modification of the algorithm to support adding necessary guards
- manual adjustment required from designer
- change of input Petri net to an equivalent mechanism

Modification of the algorithm turns out to be difficult, because guards on edges cannot check incoming data and invariants placed on destination locations block the process. Manual adjustment requires only two guards to be added (“t[3]==2” on edge leading from “serviceReq” to request_service in template “goalSequence_reqServ“ and “t[3]==1” on edge leading from “serviceReq” to “START_request_service” in template “goalSequence”). One that does not wish to modify a state machine needs to replace input/output location in dependency mechanism with either input and output or fusion ports.

Apart from the verification conditions inspired by use cases, Table 17 shows some model specific verification that bases on the experience of designer. Some tests may be redundant by being already checked in case-based tests, but with test automation, unnecessary tests are outweighed by following a systematic methodology of finding verification properties. Some tests from Table 17 cannot be represented by single formulas; others need to be modified to match a transformed model. The conditions can be conveniently extracted from the possible results of main activities from a business process logic point of view. One example is to check whether a proper join transition (last column) is reached after some conditions occur. This expectation-based reasoning relies on precondition that a request is processed only once:

```
E<> (IsPaymentRejected || IsPaymentAuthorized) && goalSequence.authPaym
```

All tests are included in the appendix D.

To sum up, the model passed successfully all tests apart from the one that checks whether request is “fresh”, since time analysis has not been introduced yet.

8.6.2. Top-down abstraction refining

After adjusting the model to limitations, the element number increased the number of locations to 296 (from 252), transitions to 161 (from 118) and arcs to 480 (from 393). As a consequence, the state space became twice as big:

```
State Space
Nodes: 4381
Arcs: 9055
Secs: 8
Status: Full
Scc Graph
Nodes: 2941
Arcs: 7207
Secs: 1
```

Apart from adding new elements, a few other minor element locations and name modifications had to be introduced to the Petri net. Name changes are connected with Uppaal’s limitation for a maximum length of template name which is 64 characters. Since a state machine element is created hierarchically by adding consequent children element names to already existing parent’s name, some descriptive names had to be abbreviated (for example S stands for System). Since the input Petri net was not using any string type variables, no changes had to be made in this area.

The result Uppaal state machine reveals a number of 56 templates, 436 locations and 596 transitions. For the sake of brevity, only few most interesting parts are described below. Complete models are on the attached CD in files: “TARM-basic.cpn” (input Petri nets) and “tarm.xml” (result state machines).

A first template depicted in Figure 8-6 shows a close resemblance to page “System” in input Petri net. For the sake of clarity edge assignments are not shown but the full (and visually improved) model is available in file “tar-extended.cpn” on the attached CD.

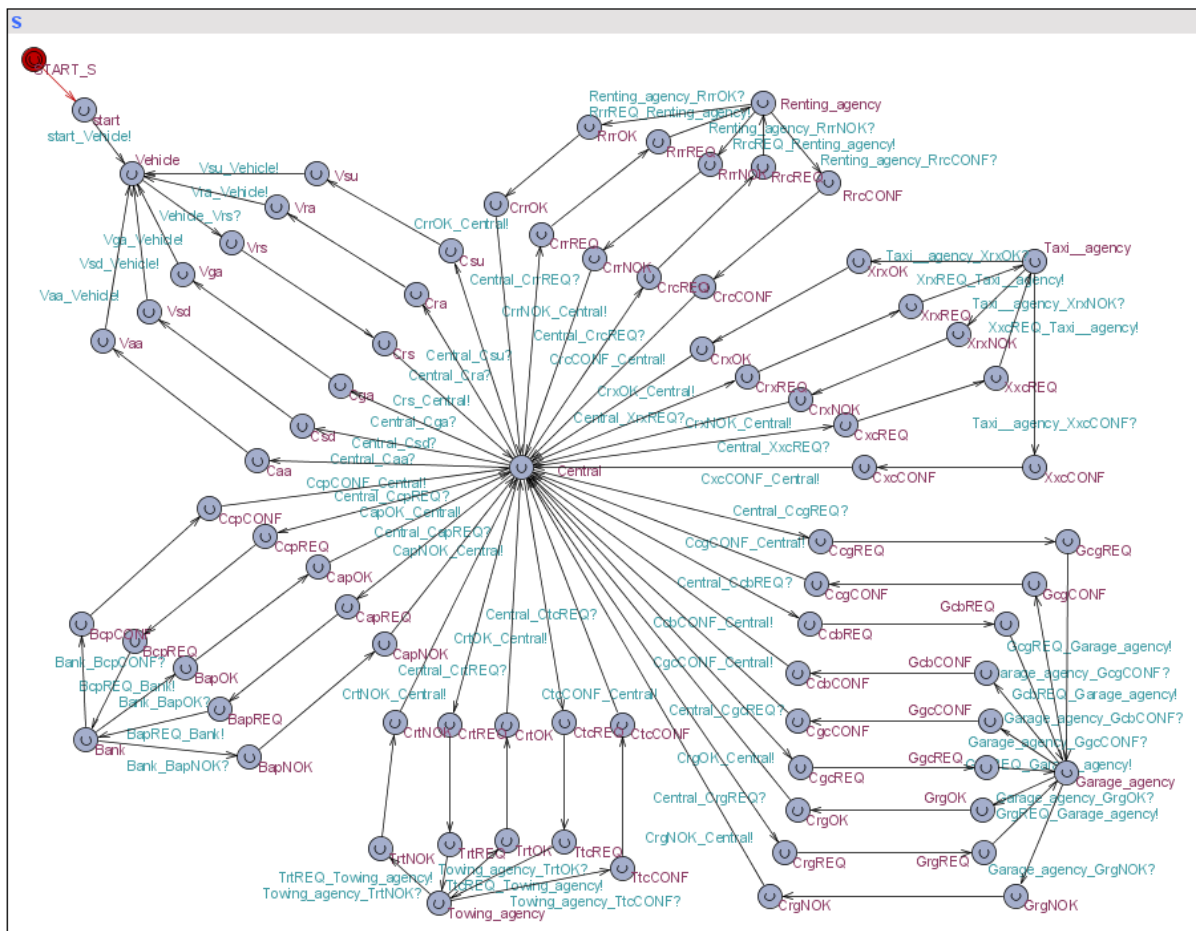


Figure 8-6 Highest overview abstraction level after transformation of TAR methodology

Similarly to the Petri net, the highest abstraction shows clearly what service interfaces are provided and required. One can also visually check that proper interfaces are connected with each other with only the first capital letter abbreviating service's name.

From the overview level, the process begins in place “START_S” (in top left corner) and follows interfaces “Vrs” and “Crs” to the Central. According to Central’s logic shown later in Figure 8-7, a Bank service is contacted by interfaces “CapREQ” (authorization payment request) and “BapREQ”, resulting in a response either through “BapOK” ([B]ank [a]uthorization [p]ayment [OK]) or BapNOK ([B]ank [a]uthorization [p]ayment [N]ot [OK]). In case the reply is negative (BapNOK), the Central replies back to the vehicle through locations “Csd” (service denial) and “Vsd”. In other case, process logic orchestrates possibly repetitive communications with other parties to reach a customer with an appropriate service request result.

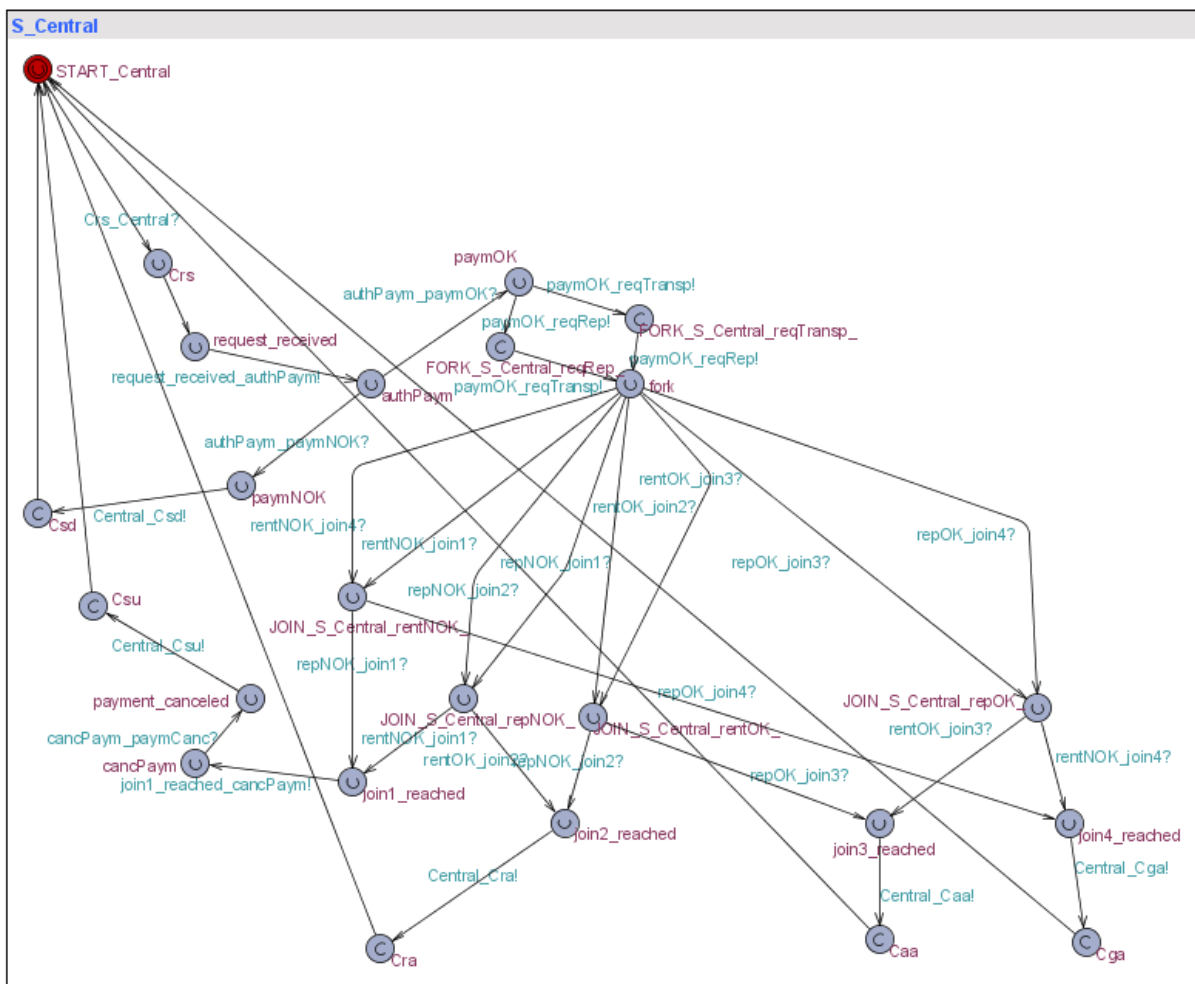


Figure 8-7 Central’s main logic

The top and bottom part of the logic of a Central, shown in Figure 8-7, corresponds to the Petri net shown in Figure 6-8. The main part is described in two concurrent templates activated at transition “fork”. Both of them are worth showing because of the comparison between abstraction driven and no abstraction experiment initiated in chapter 6.2.1. The first template “S_Central_fork_reqRep_resRep” is depicted in Figure 8-8 and shows logic related

to reserving garage and towing together with garage confirmation and cancellation. The figure is clear, since the input Petri net used an abstraction of processes to model main logic. On the other hand, model depicted in Figure 8-9 shows a transformation of a simpler process logic connected with reserving replacement car or, if that is impossible, a taxi instead. Despite being fully functional, the transformed model is hardly readable, similarly to its original Petri net. That would not be a big issue if those models were only for machine's interpretation, but algorithm debugging, testing or simulation become problematic. This experiment proves that using abstraction improves not only readability of one model, but may also greatly affect other parts of the design process. A cost for using abstraction in this case is slight with one additional template for every abstracted sub-page.

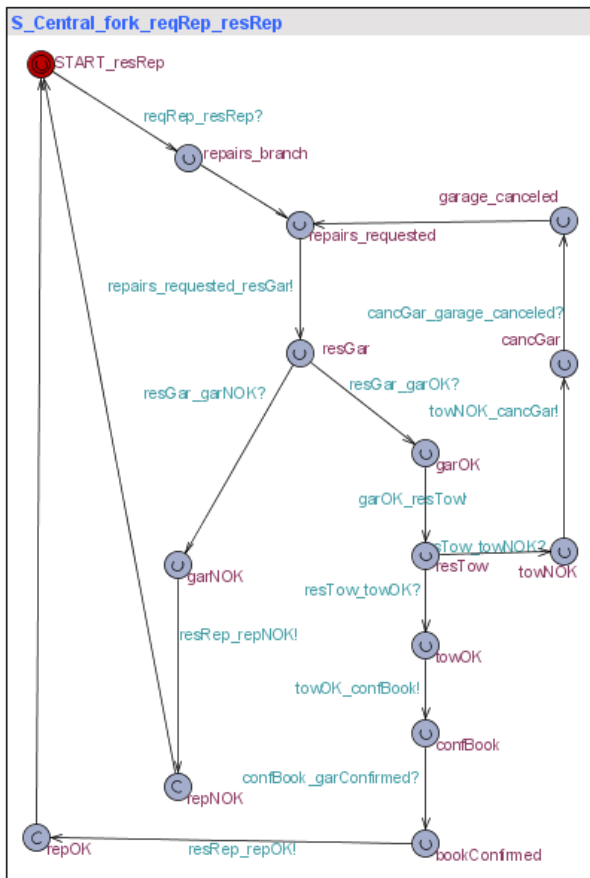


Figure 8-8 “Reserve Repair” process logic after transforming TARM model

During addition of boolean variables, the advantages of using abstraction became apparent again. As an example, template “S_Central_fork_reqRep_resRep” (“resRep” in CPN Tools) presents a clear process of right concurrent branch with 4 references to sub-pages lying on lower level (resGar, resTow, cancGar and confGar). As an experiment, the left branch modeled by template “S_Central_fork_reqTransp_resTrans” (“resTrans” in CPN Tools) models reservation of renting car and taxi without abstraction. As a result, the left branch is significantly less readable both in CPN Tools as well as in Uppaal. Issues continue further to specification of verification properties (verification conditions 16e and 16f), showing again disadvantages of less abstract approach.

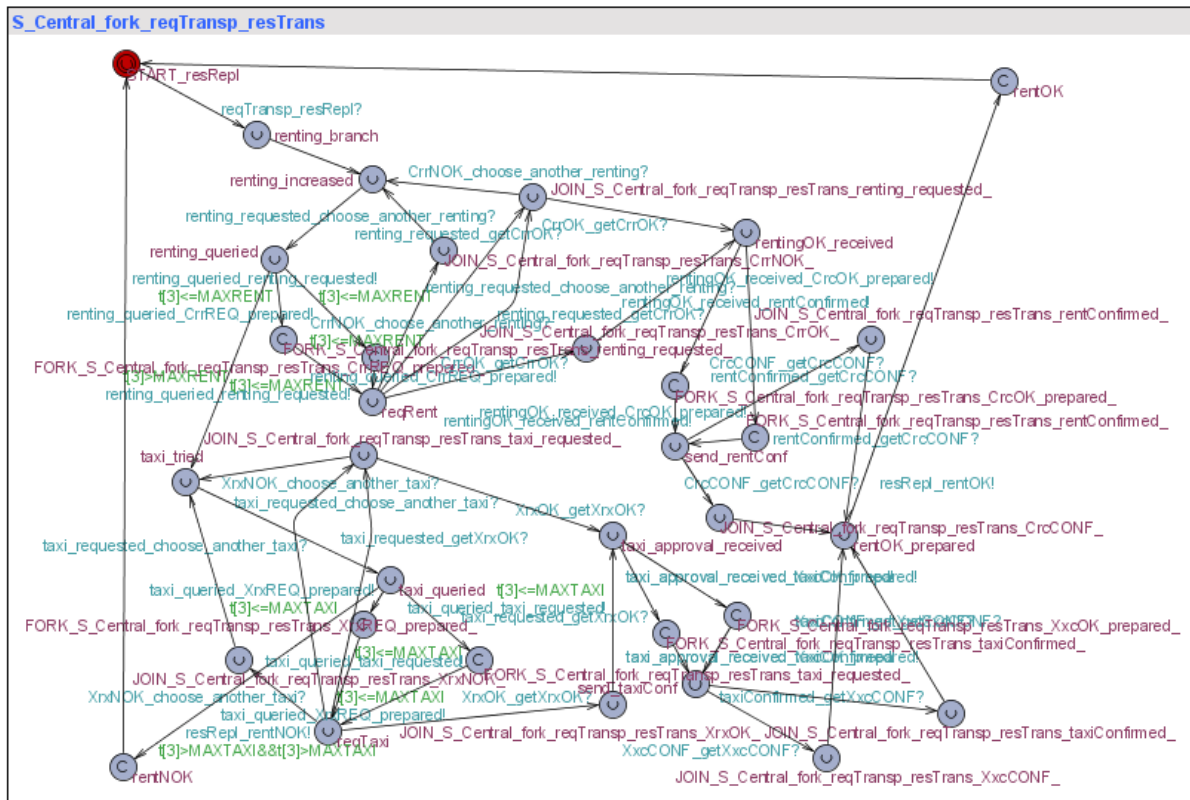


Figure 8-9 “Reserve Car” process logic after transforming TARM model

Checking verification conditions revealed both problems in input Petri net model as well as in limitation of transformation algorithm. For example, it turned out that outgoing edges from join and fork transitions cannot assign values to a token. A great amount of errors in the design was not documented due to the fact that the design was corrected simultaneously with the verification mechanism. After correcting bugs, the model passed successfully all tests, apart from the one that checks whether request is “fresh” as time analysis has not been introduced in the model.

Because of the increased size of the model more time was required to verify properties, especially those that required checking every possible behavior (safety properties). It is worth to mention that the creation of the state machine model is completely automatic.

8.7. Transformation conclusions

Chapter 8 presents an algorithm and its working implementation on how to transform a Petri net(s) compatible with CPN Tools to a state machine(s) understandable by Uppaal. Even though the algorithm has not been proven formally, its outcome has been checked in two complex models based on rather different organization and communication schemes. Additionally, the amount of elements in every model creates many different configurations that should be transformed while preserving original behavior.

Even though the transformed model is functionally equivalent to original, solutions presented by the algorithm are, in many cases, messy and different than a human would expect. Most notable is that the final model contains much more processes and elements. It results in a problematic increase of state space – a major limitation to model checking. However, differences of a model machine-made realization result from a systematic transformation that, in overall, creates an elegant, consistent and repeatable product even at a cost of human comprehensibility. Human understanding can still base on input Petri net that intuitively supports many designing techniques like abstraction, concurrency or data passing. One of the biggest advantages is that both models can be synchronized again with every change to a Petri net. In addition, the model can also be enriched with real timing properties that are not supported (to such extent) by CPN Tools.

The algorithm has still some limitations that usually require a designer to adjust a Petri net before processing. Since no other similar research is known in this area, the application had to be created from scratch during repetitive iterations of planning, implementing and debugging every functionality. Most of those limitations are caused by algorithmic complexity to predict all possible element configurations that may appear in a Petri net and in most cases can be eliminated by additional elements. Nevertheless, despite all constraints, a designer has a wide possibility of processing a Petri net with concurrency, data passing between processes and communication by either ports or fused places.

Even despite the increased state space due to the additional elements, as predicted, Uppaal provides the verification functionality that CPN Tools lacks. It allows checking all reachability, liveness and safety properties that were prepared with UML analysis. Even though the test scenario is rather small comparing to real systems, Uppaal manages to verify properties in fairly short time even on a typical home PC (P4 2,8GHz with 1GB RAM). Thus, there was no need for any extensive optimization techniques such as partial state space exploration.

Not everything can be checked automatically – there is still a need for a designer to specify verification conditions manually. Apart from checking for deadlocks, whether all places are reachable and if every final location is eventually visited, it would be computationally difficult to program logic to differentiate a correct from incorrect behavior. One could think of a system to ease designer's job by adding location names, together with their hierarchy, to a formula with one click of a mouse but that would still require manual planning of verification conditions. Nevertheless, because of the significant amount of tests, it would be useful and programmatically feasible to check all the previously specified formulae in an automatic fashion.

Lastly, the transformation addresses some of Uppaal's shortages related to SOA design such as:

- concurrency between N nets has been modeled by initializing N processes in every possible order
- lack of abstraction to design services compositionally is replaced by template names that contain also ancestors' names
- data keeping and passing between templates, that is both time and workspace consuming, has been fully automated

9. Future work

Because of the time constraints, certain assumptions and simplifications had to be made. The project funds a solid backbone to further development of many of its parts. Below is a list of possible improvements ordered according to their importance together with an estimated difficulty level (simple/average/difficult).

- correct the limitation that join and fork transitions cannot lead to input, output of fused location (average)
- develop a mechanism to merge newly transformed models with already existing and modified to save time of adding manual modifications with each transformation (simple)
- sort templates according to their hierarchy, since now they are in order the system explored them (simple)
- introduce time constraints in Petri net model and develop their transformation mechanism that would replace convenient (but simplified) urgent locations (average)
- explore ML algorithms to check properties based on a state space generated in CPN Tools (average)
- introduce test automation that would allow testing multiple verification formulae also against expected results (average)
- propose automation for complex verification: detection of deadlocks, infinite loops, automatically extend model with helpful boolean variables to be used in verification conditions with default “urgency”
- introduce correlation and timeout mechanism (average)
- introduce multiple instance support (simple)
- enable transformation of more complex concurrency configuration (average)
- create CPN optimizer that could pre-transform the net to Uppaal checking for possible problems, adding additional inscriptions and elements, etc. (simple)
- create Uppaal input data verifier to check for errors that cause Uppaal’s undefined behavior (simple)
- introduce code optimization to implemented transformation algorithm to transform much more complicated models (average)
- develop a mechanism for code generating from a state machine (difficult)
- introduce complete support of color sets in transformation to Uppaal (simple)
- develop automatic generation of WSDL descriptions from a model (average)
- generation of BPEL description of component behavior to overcome a problem of describing ports and providing semantic interface functionality to verify compatibility and composition opportunities (average)

10. Conclusions

This thesis presents an approach to SOA modeling and verification basing on a hypothetical service-oriented test scenario. It shows how SOA can unify and orchestrate services creating a flexible and manageable system. Several modeling techniques have been evaluated for their applicability towards SOA. In the absence of a recommended standard, three most prominent ones: UML, Petri nets and state machines were used together to provide desired functionality. The resulting model was completed by having all functionalities bound together in one logical entity.

To achieve desired goals a successful experiment to combine functionalities of two specialized tools: CPN Tools and Uppaal, has been performed. CPN Tools supports intuitive modeling of all abstraction levels with different style and design methodologies. However, despite modeling advantages, shortcoming in liveness verification require using external verification methods. Uppaal on the other hands proves to be an efficient model checker but lacks some of the crucial modeling primitives. Model transition from CPN Tools to Uppaal required developing a systematic transformation algorithm. A working implementation between both of aforementioned specialized tools has been tested on two diverse complex models. As a result, it was shown that a system can be intuitively designed, in a human intuitive Petri net, and efficiently verified in a low-level state machine. Additionally, both tools provide more unexplored valuable capabilities of automatically testing a model. Finally, having a model as a state machine(s) makes it suitable for automatic code generation and service discovery with semantic interfaces.

One can argue that a model could be developed straight from the beginning in Uppaal, possibly with some SOA-friendly improvements. However, with an absence of those it is hard to estimate whether that would be more efficient. Moreover, in the current solution, a system can also be analyzed in CPN Tools that supports flexible ML language, performance analysis or many other extensions. As an example of a complementing functionality, both Uppaal and CPN Tools can detect deadlocks, but while CPN Tools reveals it as a list of all dead transitions, Uppaal shows a simulation trace to only one of them at a time. Depending on the circumstances, both approaches may bring useful feedback on how to correct a problem.

With a strong support for encapsulation and abstraction new verification capabilities become possible. They allow analyzing not only component behavior and protocols, but also business process that governs many services. Verification of a model has also proven to be a very efficient technique since:

- verification formulas do not have to be necessarily connected to implementation but close to real business process
- formulas are validated on an executable model and in case they are broken, specific details and conditions are revealed

In this way, two valuable features of verification, that is, simplicity and precision, are combined in one consistent mechanism. Additionally, the new model-based verification bridges the gap between business needs and technical realization allowing both groups of

users to participate in testing. Finally, programmers can focus on target platform and infrastructure details rather than on business logic created by designers.

Despite the benefits, formal reasoning of models has also some constraints. One is that working with model checkers may require some expertise of the underlying logic. However, it is believed that the time spent on training those skills will pay off with the prospect of future verification.

Being aware that a proven methodology is developed through years of best practice, it is unwise to suggest one after such a short analysis. However, basing on the used techniques, an efficient process has emerged on how to create a SOA system:

1. use UML case diagrams to specify requirements for one scenario, define logical verification conditions; create other UML diagrams as many as necessary to feel confident in the architecture
2. use UML component diagrams and sequence diagrams to define interfaces for components
3. use CPN Tools to specify all abstraction levels of a model and possibly verify it
4. transform the model to Uppaal
5. adjust verification conditions to the transformed model
6. verify the model with Uppaal, correcting each error by going back to step 2 or 3 (depending on where the error is)
7. proceed to 1 to examine another scenario, integrating it at the same time with a current system

This methodology can be further extended with additional points that will complete a life cycle of a SOA development:

8. use state machines to generate skeleton of a machine specific code
9. provide necessary code parts related to local service logic
10. deploy a system
11. test a running system

The methodology supports designing systems in following iterations:

- incrementally – process logic and system elements can be changed in several iterations; every iteration step can be efficiently verified by Uppaal after transforming a Petri net(s) to a state machine(s).
- compositionally – CPN Tools allows intuitive abstraction mechanism in both top-down as well as bottom-up approach

Model driven development has been found to open many new possibilities in terms of model description, verification and transformation. SOA approach has also proven to be suitable for model representation. Despite of the lack of a clear standard supporting both modeling and verification demands, combined functionalities of more than one tools allow achieving desired model quality. It is believed that this approach will allow creating systems not only with an increased functionality but most of all – reliability.

11. Appendix A – Glossary

Here is a list of often used or possibly ambiguous key words.

- ACID – (Atomicity, Consistency, Isolation, and Durability) is a set of properties that guarantee that transactions are processed reliably
- Actor – a computational object capable of playing several roles – both simultaneously or/and alternately [MPSG]
- Deadlock – situation that happens when a process cannot continue its process logic because it waits endlessly for a signal from another process
- DoS – (Denial of Service) is an attack to make a machine unavailable by saturating it with external communication requests so that it cannot respond to legitimate traffic or responds too slow [WIKI]
- GSM – (Goal Sequence Methodology) methodology based on a 5-step process described in [CASS] and investigated further in [FCGS]
- HCPN – (Hierarchical Colored Petri Net)
- Liveness properties – described as “something good will eventually happen” define desired behavior that will always eventually happen
- MDA – (Model Driven Architecture) open and vendor neutral approach of specifying systems by separating business and application logic from underlying platform technology[OMG]
- Model – a simplified representation used to explain the structure and behavior of a real world system
- Model checking – “is the process of checking whether a given model satisfies a given logical formula”[WIKI]
- Reachability properties – described as “something good will possibly happen” mean that a desired behavior has a chance to happen (is reachable)
- Safety properties – characterized generally as “something bad will never happen” ensure that certain behavior is restricted in all possible states
- Scc – (Strongly Connected Component) is a maximal set of nodes, where each node is reachable from any other node in the component[MCCP]
- Service – collaboration between service roles played by active objects (actors) [MPSG]
- Service goal – “the desired or successful outcome of a service invocation”[MPSG]
- Simulation – “is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works”[WIKI]
- SS – (State Space) “a description of a configuration of discrete states used as a simple model of machines”[WIKI]
- Static (code) analysis – “is the analysis of computer software that is performed without actually executing programs built from that software”[WIKI]
- TARM – (Top-down Abstraction Refining Methodology)
- Validation – process to evaluate whether model captures correctly user’s needs
- Verification – process to evaluate whether the model satisfies requirements of another formal Model

12. Appendix B – CD contents

The CD contains all additional material that has not been included in the description. A reader is welcome to generate models with a jar file of the algorithm.

- 1. Description – contains PDF and DOC file with description of the thesis
- 2. CPNTools – contains models and their state space reports created by CPN Tools (chapter 6); both methodologies (GSM and TARM) have two models each:
 - basic – is a model described in thesis
 - extended – originates in basic but is adapted for transformation algorithm
- 3. Uppaal – contains an executable Uppaal application and models created in Uppaal in chapter 7 together with their verification conditions
 - “protocolForPortG2C.xml – protocol state machine
 - “Vehicle-component.xml” – component state machine
- 4. Transformation – contains additional information from chapter 8
 - Application details
 - generated jar file with a use instruction
 - source code
 - class diagram of the application created by a free Eclipse plug-in⁶
 - Examples – contains example transformations
 - GSM – contains transformations from previously explained goal sequence methodology
 - TARM – contains transformations from previously explained top-down abstraction refining methodology
 - Note: both methodologies have basic version (right after transformation) and extended one; overlapping elements in the extended version are additionally relocated to increase model understandability; the extended version is also improved with additional verification variables, described in Appendix D – Verification properties, required to be able to verify all queries

13. Appendix C – Use cases

Case name: Request Services

Summary:

Broken vehicle requests services. Central tries to reserve any of them or cancels the deposit.

Preconditions:

1. Card information is sent

Basic course of events:

1. Request comes to the Central
2. Card information is sent to bank for authorization and reservation of deposit
3. Bank accepts the card
4. Central reserves renting, towing and garage
5. Central receives only confirmations.

⁶ <http://www.eclipsedownload.com/>

Alternative paths:

3.1: Bank does not authorize the credit card. Request is interrupted and driver receives a card error message asking for another card.

5.1: none of the services are approved

- .1: Central reserves taxi
- .2: Taxi is approved
- .3: Central sends the taxi company booking information

5.1.2: Taxi is not approved

- .1: Central cancels the deposit and apologizes the customer

5.2: only garage and towing are approved

- .1: Central reserves taxi
- .2: Taxi is approved
- .3: Central sends the Taxi Company and garage and towing booking information

5.2.2: Taxi is not approved

- .1: Central sends garage and towing booking information

5.3: only renting is approved

- .1: Central sends the renting company booking information

Postconditions:

1. Either some service is fulfilled or card payment is cancelled
-

Case name: Authorize Payment

Summary: Central sends credit card information in order to validate deposit

Preconditions:

1. A broken vehicle has just requested services (request is fresh)

Basic course of events:

1. Central asks bank to verify a credit card, debiting a deposit
2. Bank approves transaction

Alternative paths:

- 2.1: Bank rejects transaction providing a reason
-

Case name: Cancel Payment

Summary: After no services are available, deposit money debited before is canceled

Preconditions:

1. No services are available

2. Payment has previously been reserved.

Basic course of events:

1. Central requests a bank to cancel a request
2. Bank cancels a request

Postconditions:

1. No money is subtracted from customers account
-

Case name: Reserve Garage

Summary: Central requests an appointment in a garage agency.

Preconditions:

1. Deposit has been debited
2. Request has arrived

Basic course of events:

1. Asks a garage to book an appointment to fix the car
2. Garage approves booking
3. Approve garage

Alternative paths:

- 2.1: Garage rejects booking OR timeout happens
 - 2.1.1. Reject garage
-

Case name: Cancel Garage

Summary: Central requests cancellation of a previously booked garage

Preconditions:

1. Garage has been approved

Basic course of events:

1. Asks a garage agency to cancel an appointment to fix the car
2. Agency approves cancellation

Postconditions:

1. No garage is reserved for the car.
-

Case name: Confirm Garage

Summary: Central confirms a previously booked garage

Preconditions:

1. Towing has been approved

Basic course of events:

1. Asks a garage agency to confirm an appointment to fix the car

Postconditions:

1. Garage is reserved permanently.
-

Case name: Reserve Towing

Summary: Central requests an appointment in a towing agency.

Preconditions:

1. Garage has been approved

Basic course of events:

1. Asks a towing agency to book an appointment to tow the car
2. Agency approves booking
3. Approve towing

Alternative paths:

- 2.1: Agency rejects booking OR timeout happens
 - 2.1.1. Reject towing
-

Case name: Reserve Renting

Summary: Central requests an appointment in a renting agency.

Preconditions:

1. Deposit has been debited
2. Request has arrived

Basic course of events:

1. Asks renting agency to book an appointment to get a replacement car
2. Agency approves booking
3. Approve renting

Alternative paths:

- 2.1: Agency rejects booking OR timeout happens
 - 2.1.1. Reject renting
-

Case name: Reserve Taxi

Summary: Central requests a taxi.

Preconditions:

1. Renting has not been approved

Basic course of events:

1. Asks a taxi agency to book a taxi
2. Agency approves booking
3. Approve taxi

Alternative paths:

- 2.1: Agency rejects booking OR timeout happens
 - 2.1.1. Reject taxi

14. Appendix D – Verification properties

The verification properties have been inspired mostly by UML diagrams, especially use cases and adapted later to Uppaal compatible model. Some tests performed in “urgent” models (models with default locations marked as urgent) require some of the boolean variables. Due to different architecture of below models, their verification properties differ in terms of location name (but test the same properties).

Goal sequences transformation

For “leads to” verification conditions when location are urgent by default certain variables should be set to true when a model triggers transitions between locations:

- IsPaymentAuthorized – authPaym and paymOK (in template goalSequence)
 - IsPaymentRejected – authPaym and paymNOK (in template goalSequence)
 - IsTowingReserved – request_towing and towOK (in template goalSequence_fork_merge2)
 - IsTowingRejected – request_towing and towNOK (in template goalSequence_fork_merge2)
 - IsGarageReserved – request_garage and garOK (in template goalSequence_fork_merge2)
- and updated to false from garage_canceled and merge2 (in template goalSequence_fork_merge2)
- IsGarageRejected – request_garage and garNOK (in template goalSequence_fork_merge2)
 - IsGarageCanceled – cancel_garage and garCanceled (in template goalSequence_fork_merge2)
 - IsBookingConfirmed – confirm_booking and bookConf (in template goalSequence)
 - IsRentingReserved – request_renting and rentOK (in template goalSequence_fork_p1)
 - IsRentingRejected – request_renting and rentNOK(in template goalSequence_fork_p1)
 - IsTaxiReserved –request_taxi and taxiOK in template goalSequence_fork_p1)
 - IsTaxiRejected – request_taxi and taxiNOK in template goalSequence_fork_p1)
 - IsPaymentCanceled –paymCanceled and merge1 (in template goalSequence)
 - IsMerge3reached= JOIN_goalSequence_dec2_ (and also JOIN_goalSequence_dec3_) and merge3 (in template goalSequence)

Liveness and safety

1. There is no deadlock apart from the one in the final state

| Verification condition | Expected result |
|--|-----------------|
| $E \diamond (\text{deadlock} \ \&\& \ !\text{goalSequence.serviceDone})$ | false |

2. Model eventually reaches final state -there are no livelocks (like infinite loops)

| Verification condition | Expected result |
|--|-----------------|
| $\text{true} \ \text{-->} \ \text{goalSequence.serviceDone}$ | true |

3. Request is debited only once

| Verification condition | Expected result |
|--|-----------------|
| $E \diamond (\text{IsPaymentRejected} \ \parallel \ \text{IsPaymentAuthorized}) \ \&\& \ \text{goalSequence.authPaym}$ | false |

4. Payment is authorized only if request is fresh

| Verification condition | Expected result |
|--|-----------------|
| $E \diamond (\text{IsPaymentAuthorized} \ \&\& \ \text{clock} > \text{freshness})$ | false |

5. No service is requested without accepted deposit

| Verification condition | Expected result |
|---|-----------------|
| $E \diamond \ (\ ! \ \text{IsPaymentAuthorized} \ \&\& \ (\text{goalSequence_fork_merge2.request_garage} \ \parallel \ \text{goalSequence_fork_merge2.request_towing} \ \parallel \ \text{goalSequence_fork_p1.request_renting} \ \parallel \ \text{goalSequence_fork_p1.request_taxi} \ \&\& \) \)$ | false |

6. For all paths some service is fulfilled or deposit cancelled

| Verification condition | Expected result |
|--|-----------------|
| $\text{IsPaymentAuthorized} \ \text{-->} \ \text{goalSequence.serviceDone} \ \&\& \ (\text{IsTowingReserved} \ \parallel \ \text{IsRentingReserved} \ \parallel \ \text{IsTaxiReserved} \ \parallel \ \text{IsPaymentCanceled})$ | true |

7. Payment is canceled only when renting is rejected and garage is rejected and taxi is rejected

| Verification condition | Expected result |
|---|-----------------|
| $E \diamond (\text{IsPaymentCanceled} \ \&\& \ (\text{IsGarageReserved} \ \parallel \ \text{IsRentingReserved} \ \parallel \ \text{IsTaxiReserved}))$ | false |

8. Whenever a garage and taxi is unavailable then the payment will eventually be cancelled

| Verification condition | Expected result |
|---|-----------------|
| $(\text{goalSequence_fork_merge2.garNOK} \ \&\& \ \text{goalSequence_fork_p1.taxiNOK}) \ \text{-->} \ \text{IsPaymentCanceled}$ | true |

9. Taxi cannot be reserved together with renting

| Verification condition | Expected result |
|---|-----------------|
| E<> (IsRentingReserved && IsTaxiReserved) | false |

10. Garage can be requested to cancel only when it has been previously reserved

| Verification condition | Expected result |
|---|-----------------|
| E<> (!IsGarageReserved && goalSequence_fork_merge2.cancel_garage) | false |

11. Towing is reserved only when a garage has been accepted

| Verification condition | Expected result |
|--|-----------------|
| E<> (!IsGarageReserved && goalSequence_fork_merge2.request_towing) | false |

12. Garage is confirmed only when both garage and towing are accepted

| Verification condition | Expected result |
|--|-----------------|
| E<> (!IsGarageReserved && !IsTowingReserved && goalSequence.confirm_booking) | false |

Reachability

13. It is possible to successfully authorize card

| Verification condition | Expected result |
|---------------------------|-----------------|
| E<> (IsPaymentAuthorized) | true |

14. It is possible to get full service (both garage, towing and either renting or taxi)

| Verification condition | Expected result |
|--|-----------------|
| E<> (goalSequence_fork_p1.dec3 && goalSequence_fork_merge2.dec4) | true |

15. It is possible to have a garage canceled and still get a full service (another garage, towing and either renting or taxi)

| Verification condition | Expected result |
|--|-----------------|
| E<> (IsGarageCanceled==true && goalSequence_fork_p1.dec3 && goalSequence_fork_merge2.dec4) | true |

16. It is possible to reach goals from external services

| Verification condition | Expected result |
|--|-----------------|
| E<> (goalSequence_fork_merge2.garOK) | true |
| E<> (goalSequence_fork_merge2.garCanceled) | true |
| E<> (goalSequence.bookConf) | true |
| E<> (goalSequence_fork_merge2.towOK) | true |
| E<> (goalSequence_fork_p1.rentOK) | true |
| E<> (goalSequence_fork_p1.taxiOK) | true |

17. It is possible to cancel deposit

| Verification condition | Expected result |
|----------------------------------|-----------------|
| E<> (goalSequence. paymCanceled) | true |

18. It is possible to have towing reserved (in the end) even when previous garage is rejected (another garage)

| Verification condition | Expected result |
|--|-----------------|
| E<> (IsGarageCanceled && goalSequence_fork_merge2.towOK) | true |

Design specific tests

Design specific test are created by designer basing on the process logic a model should have. Table 17 shows expected states (last column) to be reached in response to possible outcome of individual process goals.

Table 17 Design specific tests

| Test/ nr. / events / | authorize credit card | request garage | request towing | request renting | request taxi | join transition number |
|----------------------------|--------------------------|-------------------|-------------------|--------------------|-----------------|------------------------------|
| 1 | NOK | - | - | - | - | none |
| 2 | OK | NOK | - | NOK | NOK | 1 |
| 3 | OK | NOK | - | OK | - | 2 |
| 4 | OK | NOK | - | NOK | OK | 2 |
| 5 | OK | OK | OK | OK | - | 3 |
| 6 | OK | OK | OK | NOK | OK | 3 |
| 7 | OK | OK | OK | NOK | NOK | 4 |

Test21

It is sufficient to find a counter example of reaching any of the join transitions. Since Uppaal does not allow checking transitions, they are replaced by their consequent locations.

E<> IsPaymentRejected && (goalSequence.merge6 || goalSequence.merge3 || goalSequence.p2)

Expected result:false

Test22

(IsPaymentAuthorized && IsGarageRejected && IsRentingRejected && IsTaxiRejected)-->

IsPaymentCanceled

Expected result: true

Test23

(IsPaymentAuthorized && IsGarageRejected && IsRentingReserved)--> IsJoin2Reached

Expected result: true

Test24

(IsPaymentAuthorized && IsGarageRejected && IsRentingRejected && IsTaxiReserved)--> IsJoin2Reached

Expected result: true

Test25

(IsPaymentAuthorized && IsGarageReserved && IsRentingReserved && IsTowingReserved)--> IsBookingConfirmed

Expected result: true

Test26

(IsPaymentAuthorized && IsGarageReserved && IsTaxiReserved && IsRentingRejected && IsTowingReserved)--> IsBookingConfirmed

Expected result: true

Test27

(IsPaymentAuthorized && IsGarageReserved && IsTaxiRejected && IsRentingRejected && IsTowingReserved)--> IsBookingConfirmed

Expected result: true

It is possible to find what is a maximum number N of garage that can be reserved:

E<> goalSequence_fork_merge2.garOK && goalSequence_fork_merge2.token[1] > N

Basing on results shows in Figure 14-1 the maximum number of garage is 2.

```
E<> goalSequence_fork_merge2.garOK && goalSequence_fork_merge2.t[1]==2
Property is satisfied.
E<> goalSequence_fork_merge2.garOK && goalSequence_fork_merge2.t[1]>2
Property is not satisfied.
```

Figure 14-1 Results of a query to find maximum number of reserved garage

Top-down abstraction refining

For “leads to” verification conditions when location are urgent by default certain variables should be set to true when a model triggers transitions between locations:

- IsPaymentAuthorized – authPaym and paymOK (in template S_Central)
- IsPaymentRejected – authPaym and paymNOK (in template S_Central)
- IsTowingReserved – resTow and towOK (in template S_Central_fork_reqRep_resRep)
- IsTowingRejected – resTow and towNOK (in template S_Central_fork_reqRep_resRep)
- IsGarageReserved – resGar and garOK (in template S_Central_fork_reqRep_resRep) and updated to false from garage_canceled and repairs_requested (in template S_Central_fork_reqRep_resRep)
- IsGarageRejected – resGar and garNOK (in template S_Central_fork_reqRep_resRep)
- IsGarageCanceled – garage_canceled and repairs_requested (in template S_Central_fork_reqRep_resRep)
- IsBookingConfirmed – confBook and bookConfirmed (in template S_Central_fork_reqRep_resRep)
- IsRentingReserved – JOIN_S_Central_fork_reqTransp_resTrans_CrcCONF_ and JOIN_S_Central_fork_reqTransp_resTrans_rentConfirmed_ to rentOK_prepared (in template S_Central_fork_reqTransp_resTrans)

- IsRentingRejected – renting_queried and taxi_tried (in template S_Central_fork_reqTransp_resTrans)
- IsTaxiReserved – JOIN_S_Central_fork_reqTransp_resTrans_XxcCONF_ (and JOIN_S_Central_fork_reqTransp_resTrans_taxiConfirmed_)(in template S_Central_fork_reqTransp_resTrans)
- IsTaxiRejected – taxi_queried and rentNOK (in template S_Central_fork_reqTransp_resTrans)
- IsPaymentCanceled – cancPaym and payment_canceled (in template S_Central)
- IsJoin2Reached –join2_reached and Cra (in template S_Central)

Liveness and safety

1. There is no deadlock apart from the one in the final state

| Verification condition | Expected result |
|---|-----------------|
| E<> deadlock && !S_Vehicle.denial_received && !S_Vehicle.apologies_received && !S_Vehicle.everything_received && !S_Vehicle.garage_received && !S_Vehicle.renting_received | false |

2. Model eventually reaches final state -there are no livelocks (like infinite loops)

| Verification condition | Expected result |
|---|-----------------|
| true --> (S_Vehicle.denial_received S_Vehicle.apologies_received S_Vehicle.everything_received S_Vehicle.garage_received S_Vehicle.renting_received) | true |

3. Request is debited only once

| Verification condition | Expected result |
|--|-----------------|
| E<> (IsPaymentRejected IsPaymentAuthorized) && S_Central.authPaym | false |

4. Payment is authorized only if request is fresh

| Verification condition | Expected result |
|--|-----------------|
| E<> (IsPaymentAuthorized && clock > freshness) | false |

5. No service is requested without accepted deposit

| Verification condition | Expected result |
|---|-----------------|
| E<> (! IsPaymentAuthorized && (S_Central_fork_reqRep_resRep_resGar.reqGar S_Central_fork_reqRep_resRep_resTow.reqTow S_Central_fork_reqTransp_resTrans.reqRent S_Central_fork_reqTransp_resTrans.reqTaxi)) | false |

6. For all paths some service(s) is fulfilled or deposit cancelled

| Verification condition | Expected result |
|--|-----------------|
| IsPaymentAuthorized --> goalSequence.serviceDone && (IsTowingReserved IsRentingReserved IsTaxiReserved IsPaymentCanceled) | true |

7. Payment is canceled only when renting is rejected and garage is rejected and taxi is rejected

| Verification condition | Expected result |
|--|-----------------|
| E<> (IsPaymentCanceled && (IsGarageReserved IsRentingReserved IsTaxiReserved)) | false |

8. Whenever a garage and taxi is unavailable then the payment will eventually be cancelled

| Verification condition | Expected result |
|--|-----------------|
| (S_Central_fork_reqRep_resRep.garNOK && S_Central_fork_reqTransp_resTrans.rentNOK) --> IsPaymentCanceled | true |

9. Taxi cannot be reserved together with renting

| Verification condition | Expected result |
|---|-----------------|
| E<> (IsRentingReserved && IsTaxiReserved) | false |

10. Garage can be requested to cancel only when it has been previously reserved

| Verification condition | Expected result |
|---|-----------------|
| E<> (!IsGarageReserved && S_Central_fork_reqRep_resRep.garage_canceled) | false |

11. Towing is reserved only when a garage has been accepted

| Verification condition | Expected result |
|---|-----------------|
| E<> (!IsGarageReserved && S_Central_fork_reqRep_resRep.towOK) | false |

12. Garage is confirmed only when both garage and towing are accepted

| Verification condition | Expected result |
|--|-----------------|
| E<> (!IsGarageReserved && !IsTowingReserved && S_Central_fork_reqRep_resRep.bookConfirmed) | false |

Reachability**13. It is possible to successfully authorize card**

| Verification condition | Expected result |
|---------------------------|-----------------|
| E<> (IsPaymentAuthorized) | true |

14. It is possible to get full service (both garage, towing and either renting or taxi)

| Verification condition | Expected result |
|------------------------|-----------------|
| E<> (S_Central.Caa) | true |

15. It is possible to have a garage canceled and still get a full service (another garage, towing and either renting or taxi)

| Verification condition | Expected result |
|---|-----------------|
| E<> (IsGarageCanceled==true && S_Central.Caa) | true |

16. It is possible to reach goals from all external services

| | Verification condition | Expected result |
|---|---|-----------------|
| a | E<> (S_Central_fork_reqRep_resRep.garOK) | true |
| b | E<> (S_Central_fork_reqRep_resRep.garage_canceled) | true |
| c | E<> (S_Central_fork_reqRep_resRep.bookConfirmed) | true |
| d | E<> (S_Central_fork_reqRep_resRep.towOK) | true |
| e | E<> (S_Central_fork_reqTransp_resTrans. JOIN_S_Central_fork_reqTransp_resTrans_rentConfirmed_ S_Central_fork_reqTransp_resTrans. JOIN_S_Central_fork_reqTransp_resTrans_CrcCONF_) | true |
| f | E<> (S_Central_fork_reqTransp_resTrans. JOIN_S_Central_fork_reqTransp_resTrans_XxcCONF_ S_Central_fork_reqTransp_resTrans. JOIN_S_Central_fork_reqTransp_resTrans_taxiConfirmed_) | true |

17. It is possible to cancel deposit

| Verification condition | Expected result |
|----------------------------------|-----------------|
| E<> (S_Central.payment_canceled) | true |

18. It is possible to have towing reserved (in the end) even when previous garage is rejected (another garage)

| Verification condition | Expected result |
|--|-----------------|
| E<> (IsGarageCanceled && S_Central_fork_reqRep_resRep.towOK) | true |

15. Appendix F – State space reports

Goal sequence methodology (basic version)

```

CPN Tools state space report for:                               Arcs: 2302
F:\!!DTU\!thesis\cpntools\Samples\goalSeq11.cp              Secs: 2
n                                                            Status: Full
Report generated: Wed Nov 07 13:26:19 2007

-----
Statistics                                                    Scc Graph
Nodes: 582
Arcs: 1592
Secs: 0

State Space
Nodes: 1086
Boundedness Properties
    
```

```

-----
Best Integer Bounds
      Upper Lower
Bank'BapREQ 1 1 0
Bank'BcpCONF 1 1 0
Bank'BcpREQ 1 1 0
Bank'CapNOK 1 1 0
Bank'CapOK 1 1 0
Garage'Gcb 1 1 0
Garage'GcbCONF 1 1 0
Garage'Gcg 1 1 0
Garage'GcgCONF 1 1 0
Garage'Ggc 1 1 0
Garage'GgcCONF 1 1 0
Garage'Grg 1 1 0
Garage'GrgNOK 1 1 0
Garage'GrgOK 1 1 0
Garage'garage_approval_sent 1 1 0
Garage'garage_confirmation_received 1 1 0
Renting'RrcCONF 1 1 0
Renting'RrcREQ 1 1 0
Renting'Rrr 1 1 0
Renting'RrrNOK 1 1 0
Renting'RrrOK 1 1 0
Renting'renting_accepted 1 1 0
Taxi'Xrx 1 1 0
Taxi'XrxNOK 1 1 0
Taxi'XrxOK 1 1 0
Taxi'XxcCONF 1 1 0
Taxi'XxcREQ 1 1 0
Taxi'taxi_accepted 1 1 0
Towing'Trt 1 1 0
Towing'TrtNOK 1 1 0
Towing'TrtOK 1 1 0
Towing'TtcCONF 1 1 0
Towing'TtcREQ 1 1 0
Towing'await_towConf 1 1 0
authPaym'CapNOK 1 1 0
authPaym'CapOK 1 1 0
authPaym'CapREQ 1 1 0
authPaym'awaitResponse 1 1 0
cancGar'Ccg 1 1 0
cancGar'CcgCONF 1 1 0
cancGar'await_garage_cancelation_confirmation 1 1 0
cancGar'garage_cancelation_received 1 1 0
cancPaym'CcpCONF 1 1 0
cancPaym'CcpREQ 1 1 0
cancPaym'await_cancelation_reply 1 1 0
confBook'Ccb 1 1 0
confBook'CcbCONF 1 1 0
confBook'await_bookConf_reply 1 1 0
confBook'bookConf_confirmed 1 1 0
goalSequence'booking_confirmed 1 1 0
goalSequence'dec1 1 1 0
goalSequence'dec2 1 1 0
goalSequence'dec3 1 1 0
goalSequence'dec4 1 1 0
goalSequence'garNOK 1 1 0
goalSequence'garOK 1 1 0

```

```

goalSequence'garage_canceled 1 1 0
goalSequence'merge1 1 1 0
goalSequence'merge2 1 1 0
goalSequence'merge3 1 1 0
goalSequence'merge6 1 1 0
goalSequence'p1 1 1 0
goalSequence'p2 1 1 0
goalSequence'paymNOK 1 1 0
goalSequence'paymOK 1 1 0
goalSequence'payment_canceled 1 1 0
goalSequence'rentNOK 1 1 0
goalSequence'rentOK 1 1 0
goalSequence'serviceDone 1 1 0
goalSequence'serviceReq 1 1 0
goalSequence'start 1 1 0
goalSequence'taxiNOK 1 1 0
goalSequence'taxiOK 1 1 0
goalSequence'towNOK 1 1 0
goalSequence'towOK 1 1 0
reqGar'Cgc 1 1 0
reqGar'Crg 1 1 0
reqGar'CrgNOK 1 1 0
reqGar'CrgOK 1 1 0
reqGar'GgcCONF 1 1 0
reqGar'UDDI_queried 1 1 0
reqGar'awaiting_garage_reply 1 1 0
reqGar'gar_approval_received_and_sent_confirmation 1 1 0
reqGar'garage_rejected 1 1 0
reqRent'CrcCONF 1 1 0
reqRent'CrcREQ 1 1 0
reqRent'Crr 1 1 0
reqRent'CrrNOK 1 1 0
reqRent'CrrOK 1 1 0
reqRent'await_rentConf_confirmation 1 1 0
reqRent'rentingDenial_decided 1 1 0
reqRent'renting_Requested 1 1 0
reqRent'renting_queried 1 1 0
reqTaxi'Crx 1 1 0
reqTaxi'CrxNOK 1 1 0
reqTaxi'CrxOK 1 1 0
reqTaxi'CxcCONF 1 1 0
reqTaxi'CxcREQ 1 1 0
reqTaxi'await_taxiConf_reply 1 1 0
reqTaxi'taxiDenial_decided 1 1 0
reqTaxi'taxi_Requested 1 1 0
reqTaxi'taxi_queried 1 1 0
reqTow'Crt 1 1 0
reqTow'CrtNOK 1 1 0
reqTow'CrtOK 1 1 0
reqTow'CtcCONF 1 1 0
reqTow'CtcREQ 1 1 0
reqTow'await_towConf_reply 1 1 0
reqTow'towingDenial_decided 1 1 0
reqTow'towing_Requested 1 1 0

```

```

                                1          0          1^(1,1,"")+
reqTow'towing_queried 1 1      0          1^(1,2,"")
                                1^(1,1,"")+
Best Upper Multi-set Bounds
Bank'BapREQ 1          1^(1,0,"")
Bank'BcpCONF 1        1^(1,3,"")
Bank'BcpREQ 1          1^(1,3,"")
Bank'CapNOK 1         1^(1,0,"")
Bank'CapOK 1          1^(1,0,"")
Garage'Gcb 1          1^(1,1,"")+
1^(1,2,"")
Garage'GcbCONF 1     1^(1,1,"")+
1^(1,2,"")
Garage'Gcg 1          1^(1,1,"")+
1^(1,2,"")
Garage'GcgCONF 1     1^(1,1,"")+
1^(1,2,"")
Garage'Ggc 1          1^(1,1,"")+
1^(1,2,"")
Garage'GgcCONF 1     1^(1,1,"")+
1^(1,2,"")
Garage'Grg 1          1^(1,1,"")+
1^(1,2,"")
Garage'GrgNOK 1      1^(1,1,"")+
1^(1,2,"")
Garage'GrgOK 1       1^(1,1,"")+
1^(1,2,"")
Garage'garage_approval_sent 1
                                1^(1,1,"")+
1^(1,2,"")
Garage'garage_confirmation_received 1
                                1^(1,1,"")+
1^(1,2,"")
Renting'RrcCONF 1    1^(1,0,"")
Renting'RrcREQ 1     1^(1,0,"")
Renting'Rrr 1        1^(1,0,"")
Renting'RrrNOK 1     1^(1,0,"")
Renting'RrrOK 1      1^(1,0,"")
Renting'renting_accepted 1
                                1^(1,0,"")
Taxi'Xrx 1           1^(1,0,"")
Taxi'XrxNOK 1        1^(1,0,"")
Taxi'XrxOK 1         1^(1,0,"")
Taxi'XxcCONF 1       1^(1,0,"")
Taxi'XxcREQ 1        1^(1,0,"")
Taxi'taxi_accepted 1
                                1^(1,0,"")
Towing'Trt 1         1^(1,1,"")+
1^(1,2,"")
Towing'TrtNOK 1      1^(1,1,"")+
1^(1,2,"")
Towing'TrtOK 1       1^(1,1,"")+
1^(1,2,"")
Towing'TtcCONF 1     1^(1,1,"")+
1^(1,2,"")
Towing'TtcREQ 1      1^(1,1,"")+
1^(1,2,"")
Towing'await_towConf 1
                                1^(1,1,"")+
1^(1,2,"")
authPaym'CapNOK 1    1^(1,0,"")
authPaym'CapOK 1     1^(1,0,"")
authPaym'CapREQ 1    1^(1,0,"")
authPaym'awaitResponse 1
                                1^(1,0,"")
cancGar'Ccg 1        1^(1,1,"")+
1^(1,2,"")
cancGar'CcgCONF 1   1^(1,1,"")+
1^(1,2,"")
cancGar'await_garage_cancelation_confirmation 1
1
                                1^((1,0,""),"depRes")+
                                1^((1,0,""),"depUnres")+

                                1^(1,1,"")+
cancGar'garage_cancelation_received 1
                                1^(1,1,"")+
1^(1,2,"")
cancPaym'CcpCONF 1  1^(1,3,"")
cancPaym'CcpREQ 1   1^(1,3,"")
cancPaym'await_cancelation_reply 1
                                1^(1,3,"")
confBook'Ccb 1       1^(1,1,"")+
1^(1,2,"")
confBook'CcbCONF 1  1^(1,1,"")+
1^(1,2,"")
confBook'await_bookConf_reply 1
                                1^(1,1,"")+
1^(1,2,"")
confBook'bookConf_confirmed 1
                                1^(1,1,"")+
1^(1,2,"")
goalSequence'booking_confirmed 1
                                1^(1,1,"")+
1^(1,2,"")
goalSequence'dec1 1 1^(1,0,"")
goalSequence'dec2 1 1^(1,3,"")
goalSequence'dec3 1 1^(1,0,"")
goalSequence'dec4 1 1^(1,1,"")+
1^(1,2,"")
goalSequence'garNOK 1
                                1^(1,3,"")
goalSequence'garOK 1
                                1^(1,1,"")+
1^(1,2,"")
goalSequence'garage_canceled 1
                                1^(1,1,"")+
1^(1,2,"")
goalSequence'merge1 1
                                1^(1,0,"")+
1^(1,2,"")+
1^(1,3,"")
goalSequence'merge2 1
                                1^(1,0,"")+
1^(1,1,"")+
1^(1,2,"")
goalSequence'merge3 1
                                1^(1,1,"")+
1^(1,2,"")+
1^(1,3,"")
goalSequence'merge6 1
                                1^(1,1,"")+
1^(1,2,"")
goalSequence'p1 1    1^(1,0,"")
goalSequence'p2 1    1^(1,3,"")
goalSequence'paymNOK 1
                                1^(1,0,"")
goalSequence'paymOK 1
                                1^(1,0,"")
goalSequence'payment_canceled 1
                                1^(1,3,"")
goalSequence'rentNOK 1
                                1^(1,0,"")
goalSequence'rentOK 1
                                1^(1,0,"")
goalSequence'serviceDone 1
                                1^(1,0,"")+
1^(1,1,"")+
1^(1,2,"")+
1^(1,3,"")
goalSequence'serviceReq 1

1^((1,0,""),"depRes")+
1^((1,0,""),"depUnres")+

```

```

1`((1,1,""),"depRes")++
1`((1,2,""),"depRes")++
1`((1,3,""),"depRes")
goalSequence'start 1
goalSequence'taxiNOK 1
goalSequence'taxiOK 1
goalSequence'towNOK 1
1`(1,2,"")
goalSequence'towOK 1
1`(1,2,"")
reqGar'Cgc 1
1`(1,2,"")
reqGar'Crg 1
1`(1,2,"")
reqGar'CrgNOK 1
1`(1,2,"")
reqGar'CrgOK 1
1`(1,2,"")
reqGar'GgcCONF 1
1`(1,2,"")
reqGar'UDDI_queried 1
1`(1,2,"")++
1`(1,3,"")
reqGar'awaiting_garage_reply 1
1`(1,2,"")

reqGar'gar_approval_received_and_sent_confirmation 1
1`(1,2,"")
reqGar'garage_rejected 1
1`(1,2,"")
reqRent'CrcCONF 1
reqRent'CrcREQ 1
reqRent'Crr 1
reqRent'CrrNOK 1
reqRent'CrrOK 1
reqRent'await_rentConf_confirmation 1
reqRent'rentingDenial_decided 1
reqRent'renting_Requested 1
reqRent'renting_queried 1
reqTaxi'Crx 1
reqTaxi'CrxNOK 1
reqTaxi'CrxOK 1
reqTaxi'CxcCONF 1
reqTaxi'CxcREQ 1
reqTaxi'await_taxiConf_reply 1
reqTaxi'taxiDenial_decided 1
reqTaxi'taxi_Requested 1
reqTaxi'taxi_queried 1
reqTow'CrT 1
reqTow'CrTOK 1
1`(1,2,"")
reqTow'CrTOK 1
1`(1,2,"")

reqTow'CtcCONF 1
1`(1,2,"")
reqTow'CtcREQ 1
1`(1,2,"")
reqTow'await_towConf_reply 1
1`(1,2,"")
reqTow'towingDenial_decided 1
1`(1,2,"")
reqTow'towing_Requested 1
1`(1,2,"")
reqTow'towing_queried 1
1`(1,2,"")

Best Lower Multi-set Bounds
Bank'BapREQ 1
Bank'BcpCONF 1
Bank'BcpREQ 1
Bank'CapNOK 1
Bank'CapOK 1
Garage'Gcb 1
Garage'GcbCONF 1
Garage'Gcg 1
Garage'GcgCONF 1
Garage'Ggc 1
Garage'GgcCONF 1
Garage'Grg 1
Garage'GrgNOK 1
Garage'GrgOK 1
Garage'garage_approval_sent 1
Garage'garage_confirmation_received 1
Renting'RrcCONF 1
Renting'RrcREQ 1
Renting'Rrr 1
Renting'RrrNOK 1
Renting'RrrOK 1
Renting'renting_accepted 1
Taxi'Xrx 1
Taxi'XrxNOK 1
Taxi'XrxOK 1
Taxi'XxcCONF 1
Taxi'XxcREQ 1
Taxi'taxi_accepted 1
Towing'Trt 1
Towing'TrtNOK 1
Towing'TrtOK 1
Towing'TtcCONF 1
Towing'TtcREQ 1
Towing'await_towConf 1
authPaym'CapNOK 1
authPaym'CapOK 1
authPaym'CapREQ 1
authPaym'awaitResponse 1
cancGar'Ccg 1
cancGar'CcgCONF 1
cancGar'await_garage_cancelation_confirmation 1
cancGar'garage_cancelation_received 1
cancPaym'CcpCONF 1
cancPaym'CcpREQ 1

```

| | | | |
|--|-------|---------------------------------------|-------|
| cancPaym'await_cancelation_reply 1 | | reqRent'CrrNOK 1 | empty |
| | empty | reqRent'CrrOK 1 | empty |
| confBook'Ccb 1 | empty | reqRent'await_rentConf_confirmation 1 | |
| confBook'CcbCONF 1 | empty | | empty |
| confBook'await_bookConf_reply 1 | | reqRent'rentingDenial_decided 1 | |
| | empty | | empty |
| confBook'bookConf_confirmed 1 | | reqRent'renting_Requested 1 | |
| | empty | | empty |
| goalSequence'booking_confirmed 1 | | reqRent'renting_queried 1 | |
| | empty | | empty |
| goalSequence'dec1 1 | empty | reqTaxi'Crx 1 | empty |
| goalSequence'dec2 1 | empty | reqTaxi'CrxNOK 1 | empty |
| goalSequence'dec3 1 | empty | reqTaxi'CrxOK 1 | empty |
| goalSequence'dec4 1 | empty | reqTaxi'CxcCONF 1 | empty |
| goalSequence'garNOK 1 | | reqTaxi'CxcREQ 1 | empty |
| | empty | reqTaxi'await_taxiConf_reply 1 | |
| goalSequence'garOK 1 | | | empty |
| | empty | reqTaxi'taxiDenial_decided 1 | |
| goalSequence'garage_canceled 1 | | | empty |
| | empty | reqTaxi'taxi_Requested 1 | |
| goalSequence'merge1 1 | | | empty |
| | empty | reqTaxi'taxi_queried 1 | |
| goalSequence'merge2 1 | | | empty |
| | empty | reqTow'CrT 1 | empty |
| goalSequence'merge3 1 | | reqTow'CrTNOK 1 | empty |
| | empty | reqTow'CrTOK 1 | empty |
| goalSequence'merge6 1 | | reqTow'CtcCONF 1 | empty |
| | empty | reqTow'CtcREQ 1 | empty |
| goalSequence'p1 1 | empty | reqTow'await_towConf_reply 1 | |
| goalSequence'p2 1 | empty | | empty |
| goalSequence'paymNOK 1 | | reqTow'towingDenial_decided 1 | |
| | empty | | empty |
| goalSequence'paymOK 1 | | reqTow'towing_Requested 1 | |
| | empty | | empty |
| goalSequence'payment_canceled 1 | | reqTow'towing_queried 1 | |
| | empty | | empty |
| goalSequence'rentNOK 1 | | | |
| | empty | | |
| goalSequence'rentOK 1 | | | |
| | empty | | |
| goalSequence'serviceDone 1 | | | |
| | empty | | |
| goalSequence'serviceReq 1 | | | |
| | empty | | |
| goalSequence'start 1 | | | |
| | empty | | |
| goalSequence'taxiNOK 1 | | | |
| | empty | | |
| goalSequence'taxiOK 1 | | | |
| | empty | | |
| goalSequence'towNOK 1 | | | |
| | empty | | |
| goalSequence'towOK 1 | | | |
| | empty | | |
| reqGar'Cgc 1 | empty | | |
| reqGar'Crg 1 | empty | | |
| reqGar'CrgNOK 1 | empty | | |
| reqGar'CrgOK 1 | empty | | |
| reqGar'GgcCONF 1 | empty | | |
| reqGar'UDDI_queried 1 | | | |
| | empty | | |
| reqGar'awaiting_garage_reply 1 | | | |
| | empty | | |
| reqGar'gar_approval_received_and_sent_confirmation 1 | | | |
| | empty | | |
| reqGar'garage_rejected 1 | | | |
| | empty | | |
| reqRent'CrcCONF 1 | empty | | |
| reqRent'CrcREQ 1 | empty | | |
| reqRent'Crr 1 | empty | | |

| | | | |
|--|--|---------------------------------------|------|
| | | Home Properties | |
| | | ----- | |
| | | Home Markings | |
| | | None | |
| | | Liveness Properties | |
| | | ----- | |
| | | Dead Markings | |
| | | [17,1046,1082,1086] | |
| | | Dead Transition Instances | |
| | | None | |
| | | Live Transition Instances | |
| | | None | |
| | | Fairness Properties | |
| | | ----- | |
| | | Bank'authorize_payment 1 | |
| | | Fair | |
| | | Bank'confirm_cancelation 1 | |
| | | Fair | |
| | | Bank'reject_payment 1 | Fair |
| | | Garage'await_garage_confirmation 1 | |
| | | No Fairness | |
| | | Garage'receive_booking_cancelation 1 | |
| | | No Fairness | |
| | | Garage'receive_booking_confirmation 1 | |
| | | Fair | |
| | | Garage'send_garage_approval 1 | |


```

                No Fairness
Garage'send_garage_denial 1
                No Fairness
Renting'acceptRenting2 1
                No Fairness
Renting'await_rentConf 1
                No Fairness
Renting'rejectRenting2 1
                No Fairness
Taxi'acceptTaxi2 1    No Fairness
Taxi'await_taxiConf 1 No Fairness
Taxi'rejectTaxi2 1    No Fairness
Towing'acceptTowing 1 No Fairness
Towing'await_towing_confirmation 1
                No Fairness
Towing'rejectTowing 1 No Fairness
authPaym'receive_payment_approval 1
                Fair
authPaym'receive_payment_rejection 1
                Fair
authPaym'send_request 1
                Fair

cancGar'receive_garage_cancelation_confirmatio
n 1
                No Fairness
cancGar'request_cancelation 1
                No Fairness
cancGar'send_cancelation_request 1
                No Fairness
cancPaym'finalize_payment_cancelation 1
                Fair
cancPaym'send_cancelation 1
                Fair
confBook'approve_booking 1
                Fair
confBook'receive_bookConf_reply 1
                Fair
confBook'sent_booking_confirmation 1
                Fair
goalSequence'aux1 1    No Fairness
goalSequence'aux10 1   Fair
goalSequence'aux11 1   No Fairness
goalSequence'aux2 1    No Fairness
goalSequence'aux3 1    Fair
goalSequence'aux4 1    Fair
goalSequence'aux5 1    No Fairness
goalSequence'aux6 1    No Fairness
goalSequence'aux7 1    Fair
goalSequence'aux8 1    No Fairness
goalSequence'aux9 1    Fair
goalSequence'fork 1    Fair
goalSequence'join1 1   Fair
goalSequence'join2 1   Fair
goalSequence'join3 1   Fair
goalSequence'join4 1   Fair
reqGar'contact_garage 1
                No Fairness
reqGar'query_UDDI 1    No Fairness
reqGar'query_UDDI_again 1
                No Fairness
reqGar'receive_garage_approval 1
                No Fairness
reqGar'receive_garage_confirmation_reply 1
                No Fairness
reqGar'receive_garage_rejection 1
                No Fairness
reqGar'reject_garage 1 No Fairness
reqRent'choose_next_renting 1
                No Fairness
reqRent'contact_renting 1
                No Fairness
reqRent'lookup_renting 1
                No Fairness
reqRent'receive_renting_approval 1
                No Fairness
reqRent'receive_renting_confirmation 1
                No Fairness
reqRent'receive_renting_rejection 1
                No Fairness
reqRent'reject_renting 1
                No Fairness
reqServ't1 1           Fair
reqServ't2 1           Fair
reqTaxi'choose_next_taxi 1
                No Fairness
reqTaxi'contact_taxi 1 No Fairness
reqTaxi'lookup_taxi 1  No Fairness
reqTaxi'receive_taxi_approval 1
                No Fairness
reqTaxi'receive_taxi_confirmation 1
                No Fairness
reqTaxi'receive_taxi_rejection 1
                No Fairness
reqTaxi'reject_taxi 1  No Fairness
reqTow'choose_next_towing 1
                No Fairness
reqTow'contact_towing 1
                No Fairness
reqTow'lookup_towing 1 No Fairness
reqTow'receive_towing_approval 1
                No Fairness
reqTow'receive_towing_confirmation 1
                No Fairness
reqTow'receive_towing_rejection 1
                No Fairness
reqTow'reject_towing 1 No Fairness

```

Abstraction refining (basic version)

CPN Tools state space report for:
F:\!!DTU\!thesis\cpntools\Samples\myWay14.cpn
Report generated: Thu Nov 08 16:39:47 2007

Arcs: 3333
Secs: 0

Statistics

```

-----
State Space
Nodes: 2093
Arcs: 4383
Secs: 4
Status: Full

```

```

Scc Graph
Nodes: 1317

```

Boundedness Properties

```

-----
Best Integer Bounds
                Upper      Lower
Bank'payment_approved 1 1      0
Central'join1_reached 1 1      0
Central'paymNOK 1      1      0
Central'paymOK 1      1      0
Central'payment_canceled 1
                1      0
Central'rentNOK 1      1      0

```

| | | | | | |
|---|---|---|-----------------------------------|---|---|
| Central'rentOK 1 | 1 | 0 | | 1 | 0 |
| Central'renting_branch 1 | | | reserve_towing'towing_queried 1 | | |
| | 1 | 0 | | 1 | 0 |
| Central'repNOK 1 | 1 | 0 | reserve_towing'towing_requested 1 | | |
| Central'repOK 1 | 1 | 0 | | 1 | 0 |
| Central'repairs_branch 1 | | | system'BapNOK 1 | 1 | 0 |
| | 1 | 0 | system'BapOK 1 | 1 | 0 |
| Central'request_received 1 | | | system'BapREQ 1 | 1 | 0 |
| | 1 | 0 | system'BcpCONF 1 | 1 | 0 |
| Garage'confirmation_received 1 | | | system'BcpREQ 1 | 1 | 0 |
| | 1 | 0 | system'Caa 1 | 1 | 0 |
| Garage'garage_approved 1 | | | system'CapNOK 1 | 1 | 0 |
| | 1 | 0 | system'CapOK 1 | 1 | 0 |
| Renting_agency'renting_approved 1 | | | system'CapREQ 1 | 1 | 0 |
| | 1 | 0 | system'CcbCONF 1 | 1 | 0 |
| Taxi'taxi_approved 1 | 1 | 0 | system'CcbREQ 1 | 1 | 0 |
| Towing_agency'towing_approved 1 | | | system'CcgCONF 1 | 1 | 0 |
| | 1 | 0 | system'CcgREQ 1 | 1 | 0 |
| Vehicle'apologies_received 1 | | | system'CcpCONF 1 | 1 | 0 |
| | 1 | 0 | system'CcpREQ 1 | 1 | 0 |
| Vehicle'denial_received 1 | | | system'Cga 1 | 1 | 0 |
| | 1 | 0 | system'CgcCONF 1 | 1 | 0 |
| Vehicle'everything_received 1 | | | system'CgcREQ 1 | 1 | 0 |
| | 1 | 0 | system'Cra 1 | 1 | 0 |
| Vehicle'garage_received 1 | | | system'CrcCONF 1 | 1 | 0 |
| | 1 | 0 | system'CrcREQ 1 | 1 | 0 |
| Vehicle'renting_received 1 | | | system'CrgNOK 1 | 1 | 0 |
| | 1 | 0 | system'CrgOK 1 | 1 | 0 |
| Vehicle'service_requested 1 | | | system'CrgREQ 1 | 1 | 0 |
| | 1 | 0 | system'CrrNOK 1 | 1 | 0 |
| Vehicle'start 1 | 1 | 0 | system'CrrOK 1 | 1 | 0 |
| authorize_card'authorization_requested 1 | | | system'CrrREQ 1 | 1 | 0 |
| | 1 | 0 | system'Crs 1 | 1 | 0 |
| cancel_garage'garage_cancellation_sent 1 | | | system'CrtNOK 1 | 1 | 0 |
| | 1 | 0 | system'CrtOK 1 | 1 | 0 |
| cancel_payment'paymCanc_requested 1 | | | system'CrtREQ 1 | 1 | 0 |
| | 1 | 0 | system'CrxNOK 1 | 1 | 0 |
| confirm_garage'garage_confirmation_sent 1 | | | system'CrxOK 1 | 1 | 0 |
| | 1 | 0 | system'CrxREQ 1 | 1 | 0 |
| reserve_garage'garage_confirmed 1 | | | system'Csd 1 | 1 | 0 |
| | 1 | 0 | system'Csu 1 | 1 | 0 |
| reserve_garage'garage_queried 1 | | | system'CtcCONF 1 | 1 | 0 |
| | 1 | 0 | system'CtcREQ 1 | 1 | 0 |
| reserve_garage'garage_requested 1 | | | system'CxcCONF 1 | 1 | 0 |
| | 1 | 0 | system'CxcREQ 1 | 1 | 0 |
| reserve_renting'renting_confirmed 1 | | | system'GcbCONF 1 | 1 | 0 |
| | 1 | 0 | system'GcbREQ 1 | 1 | 0 |
| reserve_renting'renting_queried 1 | | | system'GcgCONF 1 | 1 | 0 |
| | 1 | 0 | system'GcgREQ 1 | 1 | 0 |
| reserve_renting'renting_requested 1 | | | system'GgcCONF 1 | 1 | 0 |
| | 1 | 0 | system'GgcREQ 1 | 1 | 0 |
| reserve_renting'taxi_confirmed 1 | | | system'GrgNOK 1 | 1 | 0 |
| | 1 | 0 | system'GrgOK 1 | 1 | 0 |
| reserve_renting'taxi_queried 1 | | | system'GrgREQ 1 | 1 | 0 |
| | 1 | 0 | system'RrcCONF 1 | 1 | 0 |
| reserve_renting'taxi_requested 1 | | | system'RrcREQ 1 | 1 | 0 |
| | 1 | 0 | system'RrrNOK 1 | 1 | 0 |
| reserve_renting'taxi_tried 1 | | | system'RrrOK 1 | 1 | 0 |
| | 1 | 0 | system'RrrREQ 1 | 1 | 0 |
| reserve_repairs'garConfirmed 1 | | | system'TrtNOK 1 | 1 | 0 |
| | 1 | 0 | system'TrtOK 1 | 1 | 0 |
| reserve_repairs'garNOK 1 | | | system'TrtREQ 1 | 1 | 0 |
| | 1 | 0 | system'TtcCONF 1 | 1 | 0 |
| reserve_repairs'garOK 1 1 | | | system'TtcREQ 1 | 1 | 0 |
| reserve_repairs'garage_canceled 1 | | | system'Vaa 1 | 1 | 0 |
| | 1 | 0 | system'Vga 1 | 1 | 0 |
| reserve_repairs'repairs_requested 1 | | | system'Vra 1 | 1 | 0 |
| | 1 | 0 | system'Vrs 1 | 1 | 0 |
| reserve_repairs'towNOK 1 | | | system'Vsd 1 | 1 | 0 |
| | 1 | 0 | system'Vsu 1 | 1 | 0 |
| reserve_repairs'towOK 1 1 | | | system'XrxNOK 1 | 1 | 0 |
| reserve_towing'towing_confirmed 1 | | | system'XrxOK 1 | 1 | 0 |

| | | | |
|---|-------------|-------------|-------------------------------------|
| system'XrxREQ 1 | 1 | 0 | reserve_renting'renting_confirmed 1 |
| system'XxcCONF 1 | 1 | 0 | 1^(1,0,0) |
| system'XxcREQ 1 | 1 | 0 | reserve_renting'renting_queried 1 |
| | | | 1^(1,0,0) |
| Best Upper Multi-set Bounds | | | reserve_renting'renting_requested 1 |
| Bank'payment_approved 1 | | | 1^(1,0,0) |
| | 1^(1,0,0) | | reserve_renting'taxi_confirmed 1 |
| Central'join1_reached 1 | | | 1^(1,0,0) |
| | 1^(1,3,0) | | reserve_renting'taxi_queried 1 |
| Central'paymNOK 1 | 1^(1,0,0) | | 1^(1,0,0) |
| Central'paymOK 1 | 1^(1,0,0) | | reserve_renting'taxi_requested 1 |
| Central'payment_canceled 1 | | | 1^(1,0,0) |
| | 1^(1,3,0) | | reserve_renting'taxi_tried 1 |
| Central'rentNOK 1 | 1^(1,0,0) | | 1^(1,0,0) |
| Central'rentOK 1 | 1^(1,0,0) | | reserve_repairs'garConfirmed 1 |
| Central'renting_branch 1 | | | 1^(1,1,0)++ |
| | 1^(1,0,0) | 1^(1,2,0) | |
| Central'repNOK 1 | 1^(1,3,0) | | reserve_repairs'garNOK 1 |
| Central'repOK 1 | 1^(1,1,0)++ | | 1^(1,3,0) |
| 1^(1,2,0) | | | reserve_repairs'garOK 1 |
| Central'repairs_branch 1 | | | 1^(1,1,0)++ |
| | 1^(1,0,0) | 1^(1,2,0) | |
| Central'request_received 1 | | | reserve_repairs'garage_canceled 1 |
| | 1^(1,0,0) | | 1^(1,1,0)++ |
| Garage'confirmation_received 1 | | 1^(1,2,0) | |
| | 1^(1,1,0)++ | | reserve_repairs'repairs_requested 1 |
| 1^(1,2,0) | | | 1^(1,0,0)++ |
| Garage'garage_approved 1 | | 1^(1,1,0)++ | |
| | 1^(1,1,0)++ | 1^(1,2,0) | |
| 1^(1,2,0) | | | reserve_repairs'towNOK 1 |
| Renting_agency'renting_approved 1 | | | 1^(1,1,0)++ |
| | 1^(1,0,0) | 1^(1,2,0) | |
| Taxi'taxi_approved 1 | | | reserve_repairs'towOK 1 |
| | 1^(1,0,0) | | 1^(1,1,0)++ |
| Towing_agency'towing_approved 1 | | 1^(1,2,0) | |
| | 1^(1,1,0)++ | | reserve_towing'towing_confirmed 1 |
| 1^(1,2,0) | | | 1^(1,1,0)++ |
| Vehicle'apologies_received 1 | | 1^(1,2,0) | |
| | 1^(1,3,0) | | reserve_towing'towing_queried 1 |
| Vehicle'denial_received 1 | | | 1^(1,1,0)++ |
| | 1^(1,0,0) | 1^(1,2,0) | |
| Vehicle'everything_received 1 | | | reserve_towing'towing_requested 1 |
| | 1^(1,1,0)++ | | 1^(1,1,0)++ |
| 1^(1,2,0) | | 1^(1,2,0) | |
| Vehicle'garage_received 1 | | | system'BapNOK 1 |
| | 1^(1,1,0)++ | | 1^(1,0,0) |
| 1^(1,2,0) | | | system'BapOK 1 |
| Vehicle'renting_received 1 | | | 1^(1,0,0) |
| | 1^(1,3,0) | | system'BapREQ 1 |
| | 1^(1,0,0) | | 1^(1,0,0) |
| Vehicle'service_requested 1 | | | system'BcpCONF 1 |
| | 1^(1,0,0) | | 1^(1,3,0) |
| Vehicle'start 1 | 1^(1,0,0) | 1^(1,2,0) | system'BcpREQ 1 |
| authorize_card'authorization_requested 1 | | | 1^(1,3,0) |
| | 1^(1,0,0) | | system'Caa 1 |
| cancel_garage'garage_cancelation_sent 1 | | | 1^(1,1,0)++ |
| | 1^(1,1,0)++ | 1^(1,2,0) | |
| 1^(1,2,0) | | | system'CapNOK 1 |
| cancel_payment'paymCanc_requested 1 | | | 1^(1,0,0) |
| | 1^(1,3,0) | 1^(1,2,0) | system'CapOK 1 |
| | 1^(1,0,0) | | 1^(1,0,0) |
| confirm_garage'garage_confirmation_sent 1 | | | system'CapREQ 1 |
| | 1^(1,1,0)++ | | 1^(1,0,0) |
| 1^(1,2,0) | | | system'CcbCONF 1 |
| reserve_garage'garage_confirmed 1 | | | 1^(1,1,0)++ |
| | 1^(1,1,0)++ | 1^(1,2,0) | |
| 1^(1,2,0) | | | system'CcbREQ 1 |
| reserve_garage'garage_queried 1 | | | 1^(1,1,0)++ |
| | 1^(1,1,0)++ | 1^(1,2,0) | |
| 1^(1,2,0)++ | | | system'CcgCONF 1 |
| 1^(1,3,0) | | | 1^(1,1,0)++ |
| reserve_garage'garage_requested 1 | | | system'CcgREQ 1 |
| | 1^(1,1,0)++ | 1^(1,2,0) | |
| 1^(1,2,0) | | | system'CcpCONF 1 |
| | | | 1^(1,3,0) |
| | | | system'CcpREQ 1 |
| | | | 1^(1,3,0) |
| | | | system'Cga 1 |
| | | | 1^(1,1,0)++ |
| | | | system'CgcCONF 1 |
| | | | 1^(1,1,0)++ |
| | | | system'CgcREQ 1 |
| | | | 1^(1,1,0)++ |
| | | | system'Cra 1 |
| | | | 1^(1,3,0) |
| | | | system'CrcCONF 1 |
| | | | 1^(1,0,0) |

| | | | |
|------------------|-------------|---|-----------|
| system'CrcREQ 1 | 1^(1,0,0) | system'XxcCONF 1 | 1^(1,0,0) |
| system'CrgNOK 1 | 1^(1,1,0)++ | system'XxcREQ 1 | 1^(1,0,0) |
| 1^(1,2,0) | | | |
| system'CrgOK 1 | 1^(1,1,0)++ | Best Lower Multi-set Bounds | |
| 1^(1,2,0) | | Bank'payment_approved 1 | empty |
| system'CrgREQ 1 | 1^(1,1,0)++ | Central'join1_reached 1 | empty |
| 1^(1,2,0) | | Central'paymNOK 1 | empty |
| system'CrrNOK 1 | 1^(1,0,0) | Central'paymOK 1 | empty |
| system'CrrOK 1 | 1^(1,0,0) | Central'payment_canceled 1 | empty |
| system'CrrREQ 1 | 1^(1,0,0) | Central'rentNOK 1 | empty |
| system'Crs 1 | 1^(1,0,0) | Central'rentOK 1 | empty |
| system'CrtNOK 1 | 1^(1,1,0)++ | Central'renting_branch 1 | empty |
| 1^(1,2,0) | | Central'repNOK 1 | empty |
| system'CrtOK 1 | 1^(1,1,0)++ | Central'repOK 1 | empty |
| 1^(1,2,0) | | Central'repairs_branch 1 | empty |
| system'CrtREQ 1 | 1^(1,1,0)++ | Central'request_received 1 | empty |
| 1^(1,2,0) | | Garage'confirmation_received 1 | empty |
| system'CrxNOK 1 | 1^(1,0,0) | Garage'garage_approved 1 | empty |
| system'CrxOK 1 | 1^(1,0,0) | Renting_agency'renting_approved 1 | empty |
| system'CrxREQ 1 | 1^(1,0,0) | Taxi'taxi_approved 1 | empty |
| system'Csd 1 | 1^(1,0,0) | Towing_agency'towing_approved 1 | empty |
| system'Csu 1 | 1^(1,3,0) | Vehicle'apologies_received 1 | empty |
| system'CtcCONF 1 | 1^(1,1,0)++ | Vehicle'denial_received 1 | empty |
| 1^(1,2,0) | | Vehicle'everything_received 1 | empty |
| system'CtcREQ 1 | 1^(1,1,0)++ | Vehicle'garage_received 1 | empty |
| 1^(1,2,0) | | Vehicle'renting_received 1 | empty |
| system'CxcCONF 1 | 1^(1,0,0) | Vehicle'service_requested 1 | empty |
| system'CxcREQ 1 | 1^(1,0,0) | Vehicle'start 1 | empty |
| system'GcbCONF 1 | 1^(1,1,0)++ | authorize_card'authorization_requested 1 | empty |
| 1^(1,2,0) | | cancel_garage'garage_cancellation_sent 1 | empty |
| system'GcbREQ 1 | 1^(1,1,0)++ | cancel_payment'paymCanc_requested 1 | empty |
| 1^(1,2,0) | | confirm_garage'garage_confirmation_sent 1 | empty |
| system'GcgCONF 1 | 1^(1,1,0)++ | reserve_garage'garage_confirmed 1 | empty |
| 1^(1,2,0) | | reserve_garage'garage_queried 1 | empty |
| system'GcgREQ 1 | 1^(1,1,0)++ | reserve_garage'garage_requested 1 | empty |
| 1^(1,2,0) | | reserve_renting'renting_confirmed 1 | empty |
| system'GgcCONF 1 | 1^(1,1,0)++ | reserve_renting'renting_queried 1 | empty |
| 1^(1,2,0) | | reserve_renting'renting_requested 1 | empty |
| system'GgcREQ 1 | 1^(1,1,0)++ | reserve_renting'taxi_confirmed 1 | empty |
| 1^(1,2,0) | | reserve_renting'taxi_queried 1 | empty |
| system'GrgNOK 1 | 1^(1,1,0)++ | reserve_renting'taxi_requested 1 | empty |
| 1^(1,2,0) | | reserve_renting'taxi_tried 1 | |
| system'GrgOK 1 | 1^(1,1,0)++ | | |
| 1^(1,2,0) | | | |
| system'GrgREQ 1 | 1^(1,1,0)++ | | |
| 1^(1,2,0) | | | |
| system'RrcCONF 1 | 1^(1,0,0) | | |
| system'RrcREQ 1 | 1^(1,0,0) | | |
| system'RrrNOK 1 | 1^(1,0,0) | | |
| system'RrrOK 1 | 1^(1,0,0) | | |
| system'RrrREQ 1 | 1^(1,0,0) | | |
| system'TrtNOK 1 | 1^(1,1,0)++ | | |
| 1^(1,2,0) | | | |
| system'TrtOK 1 | 1^(1,1,0)++ | | |
| 1^(1,2,0) | | | |
| system'TrtREQ 1 | 1^(1,1,0)++ | | |
| 1^(1,2,0) | | | |
| system'TtcCONF 1 | 1^(1,1,0)++ | | |
| 1^(1,2,0) | | | |
| system'TtcREQ 1 | 1^(1,1,0)++ | | |
| 1^(1,2,0) | | | |
| system'Vaa 1 | 1^(1,1,0)++ | | |
| 1^(1,2,0) | | | |
| system'Vga 1 | 1^(1,1,0)++ | | |
| 1^(1,2,0) | | | |
| system'Vra 1 | 1^(1,3,0) | | |
| system'Vrs 1 | 1^(1,0,0) | | |
| system'Vsd 1 | 1^(1,0,0) | | |
| system'Vsu 1 | 1^(1,3,0) | | |
| system'XrxNOK 1 | 1^(1,0,0) | | |
| system'XrxOK 1 | 1^(1,0,0) | | |
| system'XrxREQ 1 | 1^(1,0,0) | | |

```

empty
reserve_repairs'garConfirmed 1
empty
reserve_repairs'garNOK 1
empty
reserve_repairs'garOK 1
empty
reserve_repairs'garage_canceled 1
empty
reserve_repairs'repairs_requested 1
empty
reserve_repairs'towNOK 1
empty
reserve_repairs'towOK 1
empty
reserve_towing'towing_confirmed 1
empty
reserve_towing'towing_queried 1
empty
reserve_towing'towing_requested 1
empty
system'BapNOK 1
empty
system'BapOK 1
empty
system'BapREQ 1
empty
system'BcpCONF 1
empty
system'BcpREQ 1
empty
system'Caa 1
empty
system'CapNOK 1
empty
system'CapOK 1
empty
system'CapREQ 1
empty
system'CcbCONF 1
empty
system'CcbREQ 1
empty
system'CcgCONF 1
empty
system'CcgREQ 1
empty
system'CcpCONF 1
empty
system'CcpREQ 1
empty
system'Cga 1
empty
system'CgcCONF 1
empty
system'CgcREQ 1
empty
system'Cra 1
empty
system'CrcCONF 1
empty
system'CrcREQ 1
empty
system'CrgNOK 1
empty
system'CrgOK 1
empty
system'CrgREQ 1
empty
system'CrrNOK 1
empty
system'CrrOK 1
empty
system'CrrREQ 1
empty
system'Crs 1
empty
system'CrtNOK 1
empty
system'CrtOK 1
empty
system'CrtREQ 1
empty
system'CrxNOK 1
empty
system'CrxOK 1
empty
system'CrxREQ 1
empty
system'Csd 1
empty
system'Csu 1
empty
system'CtcCONF 1
empty
system'CtcREQ 1
empty
system'CxcCONF 1
empty
system'CxcREQ 1
empty
system'GcbCONF 1
empty
system'GcbREQ 1
empty
system'GcgCONF 1
empty
system'GcgREQ 1
empty
system'GgcCONF 1
empty
system'GgcREQ 1
empty
system'GrgNOK 1
empty
system'GrgOK 1
empty
system'GrgREQ 1
empty
system'RrcCONF 1
empty
system'RrcREQ 1
empty

```

```

system'RrrNOK 1
empty
system'RrrOK 1
empty
system'RrrREQ 1
empty
system'TrtNOK 1
empty
system'TrtOK 1
empty
system'TrtREQ 1
empty
system'TtcCONF 1
empty
system'TtcREQ 1
empty
system'Vaa 1
empty
system'Vga 1
empty
system'Vra 1
empty
system'Vrs 1
empty
system'Vsd 1
empty
system'Vsu 1
empty
system'XrxNOK 1
empty
system'XrxOK 1
empty
system'XrxREQ 1
empty
system'XxcCONF 1
empty
system'XxcREQ 1
empty

```

Home Properties

Home Markings
None

Liveness Properties

Dead Markings
7 [22,2093,2079,2062,1934,...]

Dead Transition Instances
None

Live Transition Instances
None

Fairness Properties

Bank'approve_payment 1 Fair
Bank'cancel_payment 1 Fair
Bank'reject_payment 1 Fair
Central'apologize 1 Fair
Central'deny_services 1
Fair
Central'fork 1 Fair
Central'join1 1 Fair
Central'join2 1 Fair
Central'join3 1 Fair
Central'join4 1 Fair
Central'receive_request 1
Fair
Garage'approve_garage 1
No Fairness
Garage'receive_booking_confirmation 1
No Fairness
Garage'receive_garage_cancellation 1
No Fairness
Garage'receive_garage_confirmation 1
No Fairness
Garage'reject_garage 1 No Fairness
Renting_agency'approve_renting 1
No Fairness
Renting_agency'confirm_renting_confirmation 1
No Fairness
Renting_agency'reject_renting 1
No Fairness
Taxi'approve_taxi 1 No Fairness
Taxi'confirm_taxi_confirmation 1
No Fairness
Taxi'reject_taxi 1 No Fairness

| | | | |
|--|-------------|---|-------------|
| Towing_agency'approve_towing 1 | No Fairness | reserve_renting'request_taxi 1 | No Fairness |
| | No Fairness | | No Fairness |
| Towing_agency'confirm_towing_confirmation 1 | No Fairness | reserve_renting'send_renting_confirmation 1 | No Fairness |
| Towing_agency'reject_towng 1 | No Fairness | reserve_renting'send_taxi_confirmation | No Fairness |
| Vehicle'receive_all 1 | Fair | | No Fairness |
| Vehicle'receive_apologies 1 | Fair | reserve_renting'try_taxi_instead 1 | No Fairness |
| Vehicle'receive_denial 1 | Fair | reserve_repairs'approve_repairs 1 | No Fairness |
| Vehicle'receive_garage 1 | Fair | reserve_repairs'reject_repairs 1 | No Fairness |
| Vehicle'receive_renting 1 | Fair | reserve_repairs'request_another_repairs | No Fairness |
| Vehicle'req 1 | Fair | reserve_repairs'request_repairs 1 | No Fairness |
| authorize_card'receive_card_approval 1 | Fair | reserve_towing'approve_towing 1 | No Fairness |
| authorize_card'receive_card_rejection 1 | Fair | reserve_towing'choose_another_towing 1 | No Fairness |
| authorize_card'request_payment_authorization 1 | Fair | reserve_towing'choose_towing 1 | No Fairness |
| cancel_garage'finalize_garage_cancelation 1 | No Fairness | reserve_towing'reject_towing 1 | No Fairness |
| cancel_garage'send_garage_cancelation 1 | No Fairness | reserve_towing'request_towing 1 | No Fairness |
| cancel_payment'finalize_payment_cancelation 1 | Fair | reserve_towing'send_towing_confirmation | No Fairness |
| cancel_payment'request_payment_cancelation 1 | Fair | system'BCapNOK 1 | Fair |
| confirm_garage'finalize_garage_confirmation 1 | No Fairness | system'BCapOK 1 | Fair |
| confirm_garage'send_garage_confirmation | No Fairness | system'BCapREQ 1 | Fair |
| reserve_garage'approve_garage 1 | No Fairness | system'BCcpCONF 1 | Fair |
| reserve_garage'reject_garage 1 | No Fairness | system'BCcpREQ 1 | Fair |
| reserve_garage'request_garage 1 | No Fairness | system'GCcbCONF 1 | No Fairness |
| reserve_garage'search_another_garage 1 | No Fairness | system'GCcbREQ 1 | No Fairness |
| reserve_garage'search_garage 1 | No Fairness | system'GCcgCONF 1 | No Fairness |
| reserve_garage'send_garage_confirmation | No Fairness | system'GCcgREQ 1 | No Fairness |
| reserve_renting'approve_renting 1 | No Fairness | system'GCgcCONF 1 | No Fairness |
| reserve_renting'approve_taxi 1 | No Fairness | system'GCgcREQ 1 | No Fairness |
| reserve_renting'choose_another_renting | No Fairness | system'GCrgNOK 1 | No Fairness |
| reserve_renting'choose_another_taxi 1 | No Fairness | system'GCrgOK 1 | No Fairness |
| reserve_renting'choose_renting 1 | No Fairness | system'GCrgREQ 1 | No Fairness |
| reserve_renting'choose_taxi 1 | No Fairness | system'RCrcCONF 1 | No Fairness |
| reserve_renting'reject_renting 1 | No Fairness | system'RCrcREQ 1 | No Fairness |
| reserve_renting'request_renting 1 | No Fairness | system'RCrrNOK 1 | No Fairness |
| | | system'RCrrOK 1 | No Fairness |
| | | system'RCrrREQ 1 | No Fairness |
| | | system'TCrtNOK 1 | No Fairness |
| | | system'TCrtOK 1 | No Fairness |
| | | system'TCrtREQ 1 | No Fairness |
| | | system'TCtcCONF 1 | No Fairness |
| | | system'TCtcREQ 1 | No Fairness |
| | | system'VCaa 1 | Fair |
| | | system'VCga 1 | Fair |
| | | system'VCra 1 | Fair |
| | | system'VCrs 1 | Fair |
| | | system'VCsd 1 | Fair |
| | | system'VCsu 1 | Fair |
| | | system'XCrxNOK 1 | No Fairness |
| | | system'XCrxOK 1 | No Fairness |
| | | system'XCrxREQ 1 | No Fairness |
| | | system'XCxcCONF 1 | No Fairness |
| | | system'XCxcREQ 1 | No Fairness |

16. References

- [ACGC] Automatic Code Generation from Coloured Petri Nets for an Access Control System, Kjeld H. Mortensen, University of Aarhus, Department of Computer Science, Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark, k.h.mortensen@daimi.au.dk
- [ASWS] Paolucci, M., Sycara, K.: Autonomous Semantic Web services, *IEEE Internet Comput.* 7(5), 34–41 (2003)
- [AUSM] Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services, Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk, Norwegian University of Science and Technology (NTNU), Department of Telematics, N-7491 Trondheim, Norway, {kraemer, herrmann, rolv.braek}@item.ntnu.no, 2006
- [BPDT] IBM Webcast, Best Practices for the Development and Testing of Web Services, Webcast Date/Time: September 18, 2007
- [BRIT] <http://wiki.daimi.au.dk/britney/britney.wiki>
- [CASS] A Collaboration based Approach to Service Specification and Detection of Implied Scenarios, Humberto Nicolás Castejón and Rolv Bræk, Department of Telematics, Norwegian University of Science and Technology, N7491 Trondheim, Norway, {humberto.castejon, rolv.braek}@item.ntnu.no, 2006
- [CASV] A Compositional Approach to Service Validation, Jacqueline Floch and Rolv Bræk, 1. SINTEF ICT NO-7465 Trondheim, Norway, jacqueline.floch@sintef.no, 2. Norwegian University of Science and Technology, Department of Telematics NO-7491 Trondheim, Norway, rolv.braek@item.ntnu.no, 2005
- [COWS] A Calculus for Orchestration of Web Services, May 21, 2007, Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi, Dipartimento di Sistemi e Informatica Università degli Studi di Firenze, flapadula, pugliese, tiezzig@dsi.unifi.it
- [CPNT] <http://daimi.au.dk/CPNTools/>
- [CTSSM] CPN Tools State Space Manual, Authors: Kurt Jensen, Søren Christensen and Lars M. Kristensen., University of Aarhus, Department of Computer Science
- [DAM] Design/CPN ASK-CTL Manual, Version 0.9, University of Aarhus, Computer Science Department, Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark, 1996 University of Aarhus,
- [DMWS] Deploying and managing Web services: issues, solutions, and directions, Qi Yu · Xumin Liu · Athman Bouguettaya · Brahim Medjahed, Received: 12 August 2005
- [EIUC] http://www.daimi.au.dk/CPnets/intro/example_indu.html
- [FCGS] Formalizing Collaboration Goal Sequences for Service Choreography, Humberto Nicolás Castejón and Rolv Bræk, NTNU, Department of Telematics, N-7491 Trondheim, Norway, {humberto.castejon, rolv.braek}@item.ntnu.no, 2006
- [FMTP] A framework for modeling transfer protocols, Peter Herrmann *, Heiko Krumm, University at Dortmund, Fachbereich Informatik, D-44221 Dortmund, Germany, 2004
- [FSPV] Formal Security Policy Verification of Distributed Component-Structured Software, Peter Herrmann, University of Dortmund, Computer Science Department, 44221 Dortmund, Germany, Peter.Herrmann@udo.edu, 2003
- [FSWS] Formal Specification of a Web Services Protocol, James E. Johnson, David E. Langworthy, Leslie Lamport, Friedrich H. Vogt, University of Technology Hamburg-Harburg, February, 2004
- [GRAP] <http://www.graphviz.org/>

- [HSLI] High-Level Specifications: Lessons from Industry, Brannon Batson - Intel Corporation, Leslie Lamport - Microsoft Research, 11 Mar 2003
- [HUGO] <http://www.pst.ifi.lmu.de/projekte/hugo/>
- [ICS] Implementing and Combining Specifications, Leslie Lamport, 02 Sep 2004
- [IPUC] An Introduction to the Practical Use of Coloured Petri Nets, Kurt Jensen, Department of Computer Science, University of Aarhus, Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark, Phone: +45 89 42 32 34, Telefax: +45 89 42 32 55, E-mail: kjensen@daimi.aau.dk, WWW: <http://www.daimi.aau.dk/~kjensen/>
- [ISDD] Is SOA Dead, Doomed, or Misnamed? By: John Michelsen <http://soa.sys-con.com/read/439677.htm> used: Oct. 6, 2007 04:00 PM
- [JEN90] K. Jensen, Coloured Petri Nets: A high-level Language for System Design and Analysis, LNCS vol. 483, Springer Verlag 1990
- [MCCP] Model Checking Coloured Petri Nets, Exploiting Strongly Connected Components, Allan Cheng, Sren Christensen, Kjeld Hyer Mortensen, [facheng](mailto:facheng@daimi.aau.dk), [schristensen](mailto:schristensen@daimi.aau.dk), khmg@daimi.aau.dk, Computer Science Department, Aarhus University, Ny Munkegade, Building 530{540, DK{8000 Aarhus C, Denmark www.daimi.aau.dk/~kjensen/
- [MCTS] Model Checking TLA+ Specifications, Yuan Yu¹, Panagiotis Manolios², and Leslie Lamport¹, ¹ Compaq Systems Research Central, {yuanyu, lamport}@pa.dec.com, ² Department of Computer Sciences, University
- [MPSG] Modeling Peer-to-peer Service Goals in UML, Richard Torbjørn Sanders, SINTEF ICT / NTNU, N-7465/N-7491 Trondheim, Norway, richard.sanders@sintef.no or sanders@item.ntnu.no, Rolv Bræk, Norwegian University of Science and Technology (NTNU), N-7491 Trondheim, Norway, rolv.braek@item.ntnu.no, 2004
- [OMG] www.omg.org
- [PAVT] A Practical Approach to Validating and Testing Software Systems Using Scenarios, Johannes Ryser Martin Glinz, Department of Computer Science, University of Zurich, Winterthurerstrasse 190, CH-8057 Zurich, Switzerland, {ryser, glinz}@ifi.unizh.ch, 1999
- [PN] <http://www.informatik.uni-hamburg.de/TGI/PetriNets/>
- [RAMZ] Ramses: The Integrated Tool Suite for Service Development , Frank Alexander Kraemer, NTNU, Department of Telematics, N-7491 Trondheim, Norway, kraemer@item.ntnu.no
- [REPT] A roadmap to electronic payment transaction guarantees and a Colored Petri Net model checking approach, Panagiotis Katsaros, Department of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece, katsaros@csd.auth.gr
- [RMCR] Real-Time Model Checking is Really Simple, Leslie Lamport, Microsoft Research, 13 June 2005
- [RSA] <http://www.ibm.com/software/awdtools/architect/swarchitect/>
- [RSM] <http://www-306.ibm.com/software/awdtools/modeler/swmodeler/>
- [RVEU] Requirements Validation: Execution of UML Models with CPN Tools, Ricardo J. Machado · Kristian Bisgaard Lassen · Sérgio Oliveira · Marco Couto · Patrícia Pinto, Published online: 13 March 2007
- [SDCR] Service Discovery and Component Reuse with Semantic Interfaces, Richard T. Sanders, Rolv Bræk, Gregor von Bochmann, and Daniel Amyot, {bochmann, damyot}@site.uottawa.ca,
- [SENS] <http://www.sensoria-ist.eu/> used: 20-10-2007

- [SESO] Sensoria 016004, Software Engineering for Service-Oriented Overlay Computers, D8.0: Case studies scenario description, Lead contractor for deliverable: ISTI-CNR, Author(s): Stefania Gnesi, Maurice ter Beek (ISTI-CNR), Hubert Baumeister (DTU), Matthias Hoelzl (LMU), Corrado Moiso (TILab), Nora Koch (LMU and FAST), Angelika Zobel (FAST) and Michel Alessandrini (S&N)
- [SHSC] Specification of Hybrid Systems in cTLA+, Peter Hemmann and Heiko Krumm, Universitat Dortmund - Fachbereich Informatik, { herrmannlkrumm } @ ls4.informatik.uni-dortmund.de, <http://ls4-www.inforatik.uni-dortmund.de/~VS/agrvse.html>, 1997
- [SOAP] <http://www.w3.org/TR/soap/>
- [SPIN] <http://www.spinroot.com/>
- [SSCC] Service Specification by Composition of Collaborations—An Example, Frank Alexander Kraemer Peter Herrmann, Norwegian University of Science and Technology (NTNU), Department of Telematics, N-7491 Trondheim, Norway, {kraemer, herrmann}@item.ntnu.no, 2006
- [SSM] Chapter on TLA+ from, ‘Software Specification Methods’, Leslie Lamport, An Overview Using a Case Study, Hermes, April 2006
- [SSTL] Specifying systems: The TLA+ Language and Tools for hardware and Software Engineers, Leslie Lamport, 2002
- [SVER] Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networks, Lars Michael Kristensen and Kurt Jensen, Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK, {lmkristensen, kjensen}@daimi.au.dk
- [TCSS] Transforming Collaborative Service Specifications, into Efficiently Executable State Machines, Frank Alexander Kraemer and Peter Herrmann, Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques, 2007
- [TFVT] Telephone Feature Verification: Translating SDL to TLA+, Gibson, Méry (1997),
- [TLSV] Temporal Logic-Based Specification and Verification of Trust Models, Peter Herrmann, Norwegian University of Science and Technology (NTNU), Telematics Department, 7491 Trondheim, Norway, herrmann@item.ntnu.no
- [TOU] A Tutorial on Uppaal, Gerd Behrmann, Alexandre David, and Kim G. Larsen, Department of Computer Science, Aalborg University, Denmark, {behrmann, adavid, kgl}@cs.auc.dk, Updated 17th November 2004
- [TOU] A Tutorial on Uppaal, Updated 17th November 2004, Gerd Behrmann, Alexandre David, and Kim G. Larsen, Department of Computer Science, Aalborg University, Denmark, {behrmann, adavid, kgl}@cs.auc.dk.
- [TVEU] Transformation and Verification of Executable UML Models, Gunter Graw - ARGE ISKV, Essen, Germany, Peter Herrmann - University of Dortmund, Dortmund, Germany, 2004
- [UDDI] <http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm>
- [UIPB] User Interface Prototyping based on UML Scenarios and High-level Petri Nets 1, Mohammed Elkoutbi and Rudolf K. Keller, Département d’informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal, Québec H3C 3J7, Canada
- [UML2] <http://www.uml.org/#UML2.0>
- [UPP] <http://www.uppaal.com/>

[USWS] Understanding SOA with Web Services, Eric Newcomer, Greg Lomow, Publisher

[UCC] Using UML 2.0 Collaborations for Compositional Service Specification, Richard Torbjørn Sanders¹, Humberto Nicolás Castejón², Frank Alexander Kraemer², and Rolv Bræk², ¹SINTEF ICT, N-7465 Trondheim, Norway, richard.sanders@sintef.no, ²NTNU, Department of Telematics, N-7491 Trondheim, Norway, {humberto.castejon, kraemer, rolv.braek}@item.ntnu.no

[VURS] Verification of UML-based real-time system designs by means of cTLA, Gunter Graw, Peter Herrmann, and Heiko Krumm, University at Dortmund, Fachbereich Informatik, D-44221 Dortmund, fgrawjherrmannjkrumm@ls4.cs.uni-dortmund.de

[WIKI] <http://en.wikipedia.org>

[WS] <http://www.w3.org/TR/ws-arch/>

[WS-ADD] <http://www-128.ibm.com/developerworks/library/specification/ws-add/>

[WS-BPEL] <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>

[WSCA] Web Services: Concepts, Architecture, and Applications, Alonso, G., Casati, F., Kuno, H., Machiraju, V, Springer, Berlin, Heidelberg New York (ISBN: 35404440089) (2003)

[WSDL] <http://www.w3.org/TR/wsdl20/>

[WS-POL] <http://schemas.xmlsoap.org/ws/2004/09/policy>

[WS-REL] <http://docs.oasis-open.org/ws-rx/wsrn/200702/wsrn-1.1-spec-os-01.pdf>

[WS-ROUT] <http://www.devx.com/DevX/Link/16184>

[WS-SEC] <http://www-128.ibm.com/developerworks/library/specification/ws-secure/>

[WS-TRAN] <http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html>