ALLOW

Adaptable Pervasive Flows

213339

Deliverable D5.3a

Theory and mechanisms for vertical flow adaptation by on-demand flow creation, for horizontal flow adaptation by context-aware re-planning, and for flow evolution

Contributor ..... :	FBK
Author ..... :	Antonio Bucchiarone, Marco Pistore, Heorhi Raik and Adina Sirbu
Reference Number ..... :	ALLOW.FBK.D53.2011-01-14
Version ..... :	2.0 – Final
Date ..... :	2011-01-14
Classification..... :	Public
Circulation..... :	Project Team
Contract Start Date ..... :	2008-02-01 - Duration: 42 Months
Project Coordinator ..... :	USTUTT
Project Partner..... :	USTUTT (IPVS), USTUTT (IAAS), UNI PASSAU, FBK, Imperial, ULANC

FET - Future and Emerging Technologies

Project funded by the European Community under the FP7-Programme (2007 - 2013)

Copyright

Document History

Version	Issue Date	Author	Content and Changes
1.0	22/12/2010	FBK	Version Submitted to internal review
2.0	14/01/2011	FBK	Final Version

Executive Summary

The aim of this document is to provide theory and mechanisms for the adaptation and evolution problems of adaptable pervasive systems. We define, using automated planning techniques, the formal elements and the general architectures of two adaptation mechanisms (i.e., vertical and horizontal) and for evolution.

In the document, we first briefly describe the Car Logistics scenario that is used as a reference example in the rest of the document. Then we introduce the core concepts of a planning problem and some examples where it has been used in the past. Starting from these preliminary chapters, we present our solutions for different forms of flow adaptation as well as flow evolution.

Contents

List of Figures

Chapter 1

Introduction

Flows are models defining a set of activities to be done, and their relations with each other. Flows are deeply seated in many fields, including business processes and service oriented computing. In ALLOW we concentrate on a novel usage of flows: the usage of flows as a new programming paradigm for human-oriented pervasive applications. More precisely, Adaptable Pervasive Flows (APFs) [14] are proposed as an extension of traditional workflow concepts [25] in order to make them more flexible with respect to their pervasive execution environment. APFs are dynamic workflows situated in the real world that modify their execution in order to adapt to changes in their environment. This requires on the one hand that a flow must be context-aware: during execution it must be possible to obtain information on the underlying environment (e.g. relevant information on world entities, status of other flows, human activities). On the other hand flow models must be flexible enough to allow an easy and continuous adaptation. APFs are based on WS-BPEL [1], a well-known language for specifying flows in a Web Service setting, and extend it in order to implement all the aspects related to pervasive applications [18]. In this deliverable we take into account the problem to manage dynamic flows and provide theories and mechanisms for the following type of flow adaptation:

- Short-term adaptation (simply called flow adaptation hereafter): A concrete flow instance is adapted to fit its current context (see Figure 1.1 a).
- Long-term adaptation (also called flow evolution hereafter): The history of adapted flow instances and measures of their respective performance in a given context are used as an input to some learning mechanism in order to evolve the flow model on which future generations of concrete flow instances will be based (see Figure 1.1 b).

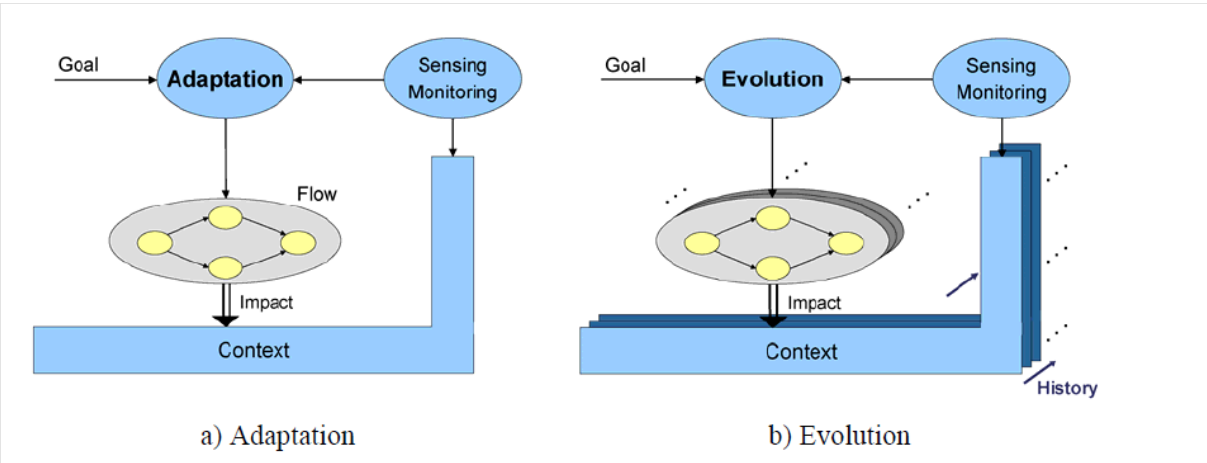


Figure 1.1: Adaptation and Evolution Life Cycles.

Regarding adaptation, we envision two basic dimensions of it: horizontal and vertical adaptation. We have horizontal adaptation when something happens that was not anticipated by the flow designer and the current flow instance has to be partially re-planned. Such a re-planning step adapts the flows structure, adding/removing activities and altering the plan. When the basic structure of the flow itself remains unchanged and only the flow composition is changed, we speak of vertical adaptation. By flow composition, we mean the mapping of flow elements to more specific sub-flows or concrete services. In a context-aware flow refinement, the flow is changed by refining (substituting) some of its elements with more specific sub-flows that fit the current context. These subflows may pre-exist in the respective environment, or they may be constructed on the fly.

The concept of flow evolution is based on the possibility to learn from histories of run-time flow adaptations and make flow models evolve continuously, thus achieving a long-term improvement of applications. Flow evolution also leads to a gain in efficiency since it avoids recurring adaptations. This reduces the overhead of executing similar adaptations repetitively and makes well-adapted flow prototypes available instantly. Flow evolution requires collecting and organizing data about histories of flow adaptations. Collected data must be analyzed in order to understand which kinds of adaptations have been successful how often, and to decide whether evolution is needed. The outcome of the learning process can result in changes to the flow structure and in the identification of adaptation patterns. The aim of this deliverable is to provide theory and mechanisms for the adaptation and evolution problems of adaptable pervasive systems. We define the formal elements and the general architectures of each of them. In detail, we try to find solutions at the following problems:

- **Vertical flow adaptation by on-demand flow creation:** For this adaptation we propose a mechanism for specifying and executing the on-demand creation of sub-flows. These mechanisms can be exploited in those situations when flow composition may not succeed since adequate sub-flows or services are not present, but new sub-flows may be created using the knowledge that is present in the flow to be adapted and in the environment.

- **Horizontal flow adaptation by context-aware re-planning:** In several situations, specific alternatives cannot be anticipated at design time, or it is impossible to predict a priori which alternatives to be followed due to lack of information on the execution contexts. In these cases, mechanisms for adaptation by pre-defined built-in context awareness (see [2, 18]) cannot be applied. In this deliverable we propose a planning-based technique able to modify the structure of the flow on the fly according to the context-specific properties while satisfying the general flow goals and constraints. Indeed, re-planning techniques should provide more flexible mechanisms for horizontal flow adaptation than built-in context awareness.

- **Flow evolution:** This problem arises when there is a need to support optimization and long-term adaptation of the flow models. Flow evolution will be based on the analysis of histories of flow instance adaptations and executions. Over time, the adaptation of flows will result in a series of different flow instances for similar tasks. Additionally, information may be collected on the success of these adaptations. The set of adapted flow instances together with the information concerning their success will be used as training cases for evolution mechanisms in order to progressively improve flow models that may then be used if a new flow shall be instantiated.

The document is structured in the following way: Chapter 2 describes the Car Logistics scenario that is used as a reference example in the rest of the document. Then we introduce, in Chapter 3 the core concepts of a planning problem and some examples where it has been used in the past. Starting from these preliminary chapters, we present our solutions for adaptation in Chapter 4 and evolution in Chapter 5. We conclude with conclusions and some open issues that we want to solve in the next months of the project.

Chapter 2

A Running Example: Car Logistics Scenario

2.1 Scenario Description

The scenario that we use to demonstrate concepts in this deliverable, has been defined after the meeting at the BIBA research center [1] in Bremen. Using it, we will illustrate the adaptation and evolution techniques presented in this deliverable. The core idea of this scenario is to organize logistic processes in a decentralized manner based on methods of autonomous control [8]. At the automobile terminal of the Bremen sea port, nearly 2 million new vehicles are handled each year; the goal is to deliver them from the manufacturer to the dealer. To achieve it a lot of intermediate processes/services are involved. These include unloading cars from a boat, storage them, apply to them treatments etc..to meet the customer's requirements for the ordered cars as well as distribute them to the retailers. The process chain of this scenario can be structured into a set of process steps as illustrated in Figure 2.1.

Figure 2.1: Process Chain of the Car Logistics Scenario.

These process steps can be described as follows:

- **Transport:** New cars are transported by ships to the terminal ares. Terminals are normally situated with very good access to the traffic infrastructure, so that cars arriving by ship can be unloaded directly onto the terminal. The expected time of arrival is sent to the terminal several days before arrival.
- **Registration:** each new vehicle is registered on the terminal and this includes the memorization of all relevant vehicle data. Each vehicle is identified by its identification number (VIN) from the terminal staff, by using mobile data entry devices, which can read barcodes.
- **Storage:** Each vehicle is allocated to a storage location in a storage area on the basis of fixed priorities. These priorities consider if there are possible technical treatment orders assigned to the vehicles and there is no differentiation regarding the type of technical treatment stations which are partially a long way away from each other. Furthermore, the parking time (i.e., the time of a vehicle in a storage location) is not taken into account in the scope of the storage allocation

[1] http://www.biba.uni-bremen.de/

process. Finally, the vehicle allocation is executed by an employee that moves the vehicle to the assigned storage location.

- **Technical Treatment:** This step is triggered by arrival of the delivery request for a vehicle. After receipt of the delivery request the vehicle is removed from stock. The sequence of the technical treatment stations a vehicle has to run through is specified in the technical treatment order of the vehicle. The technical treatment steps include e.g. recoating of vehicles, or washing and removal of separate coating layers temporarily added for vehicle protection during transport. Additional technical treatments may include installation of special equipment like radios or satellite navigation devices, and removal and replacement of certain parts or components.

- **Allocation and Delivery:** After the completion of all technical treatments tasks, the vehicle is provided to the consignment area of the terminal, where the transport service providers can take it over.

- **Transport:** in this step, each vehicle, using specialized transport service providers (using special transport vehicles like e.g. flat bed trailers or low-loading trucks) is transported from the terminal to the automobile dealer or customer.

All entities involved in the scenario can be summarized in Figure 2.2 that represents a simplification of the real scenario described before. It depicts an harbor with ships, cars, trucks and several areas (terminal, storage, commissioning, treatment and consignment). In the harbor arrive cars that must be delivered to customers (using a truck) within a specific date and with a precise set of treatments. Moreover, the harbor is served by several drivers that move cars from the ship, to a specific location of an area, until to the assigned truck.

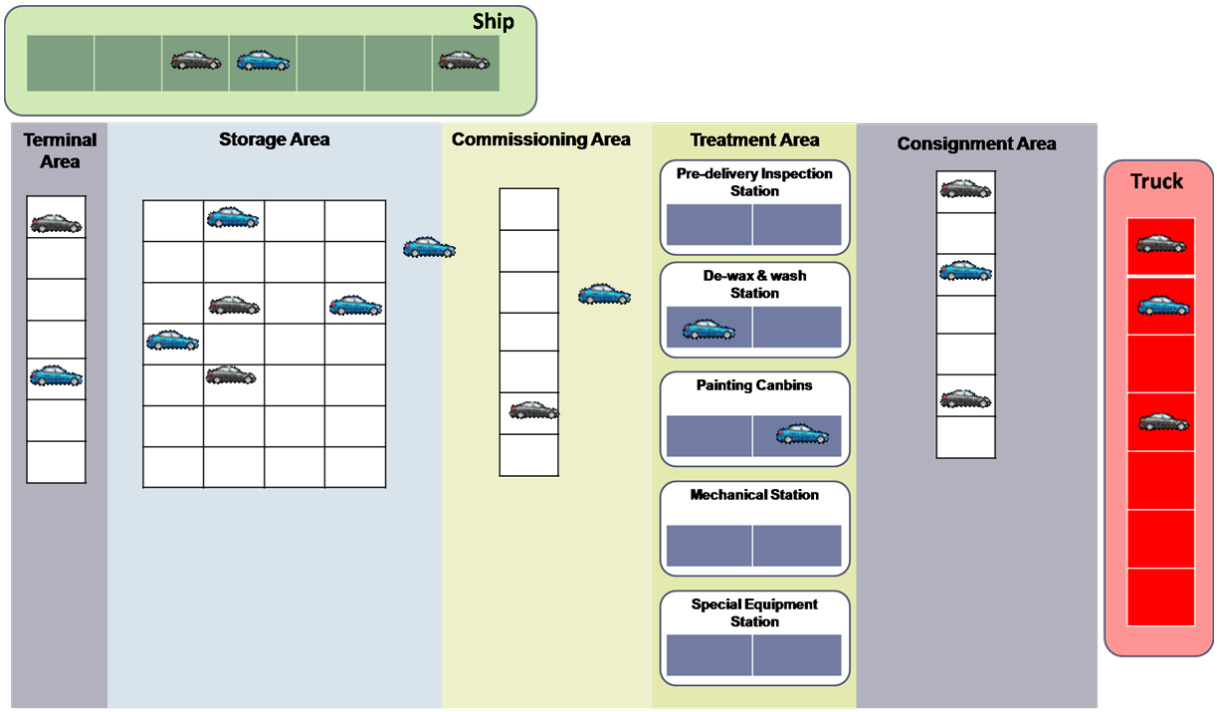


Figure 2.2: Scenario Layout.

Each car is unloaded from the ship and executes a set of steps until to be loaded on a precise truck. During the process execution each car can change its status and position triggering a set of adaptations. At the same time, to be moved from one location to another each of them invokes a precise service to book a free location. Moreover, when a customer asks for a precise set of cars with precise characteristics,

a precise treatment order is associated. This means that at each car has a precise treatments list and each treatment should be executed in a precise order. Since that the number of cars is not predictable and can change at runtime (i.e., a new ship arrives), the application should be able to adjust each processes execution respect to the runtime context. At the same time unforeseen damage to a car should be manage without to terminate its process and without to delay the others.

Chapter 3

Background

In this chapter we introduce background notions for the understanding of the approaches presented in the adaptation and evolution chapters. All approaches that we define are based on automated planning techniques [13], for this reason we introduce the main elements of this problem solution and some techniques that we use later to execute adaptation and evolution of our flow-based pervasive systems. After that we present how, using one precise planning approaches, we can compose services considering both data- and control-flow requirements. All these background elements are important to understand better what we propose in Chapters 4 and 5.

3.1 Classical Planning

The objective of planning is to reason about actions of a system, choosing and organizing them for changing the execution state and satisfying a defined goal. Since different domains can manage different kind of actions, there are also various form of planning (i.e., path planning, navigation planning, communication planning, etc..). A *planning domain* can be modeled in terms of propositions, which characterize system *states*, of *actions*, and of a *transition relation* describing system evolution from one state to possible many differen states.

Definition 3.1.1 (Planning Domain) *A planning domain $\mathcal{D}$ is a 4-tuple $\langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ where*

- *$\mathcal{P}$ is the finite set of basic propositions,*
- *$\mathcal{S} \subseteq 2^{\mathcal{P}}$ is the set of states,*
- *$\mathcal{A}$ is the finite set of actions,*
- *$\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation.*

Act(s)=$\{a : \exists s'.R(s, a, s')\}$ is the set of actions that can be performed in state s, and Exec(s,a)=$\{s' : \mathcal{R}(s, a, s')\}$ is the set of states that can be reached from s performing action $a \in$ Act(s). Consider the Car Logistic domain, described in Section 2.1. It consists of a set of Areas (Ship, Terminal, Storage, etc..) and a set of Cars that can move between these areas performing a set of activities. Some of them are depicted in Figure 3.1. Notice that action "Move" performed in the "Ship Area" moves the car either to the "Terminal Area" or to the "Treatment Area" non-deterministically. The *state* of the domain is defined in terms of *CarLocation*, that describes in which location the car is currently in, and of *CarStatus*, that describes the state of each car which is initially "Unknown", and becomes either "Damaged" or "Ok" when it is checked, moreover it becomes "On" when the car is ready to move from the source area to the target area. Plans of this domain can be represented by state-action tables, which associate to each state the action that has to be performed in such state.

Definition 3.1.2 (Plan)
A plan π for a planning domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ is a state-action table which consists of a set of pairs $\{\langle s, a \rangle : s \in \mathcal{S}, a \in Act(s)\}$

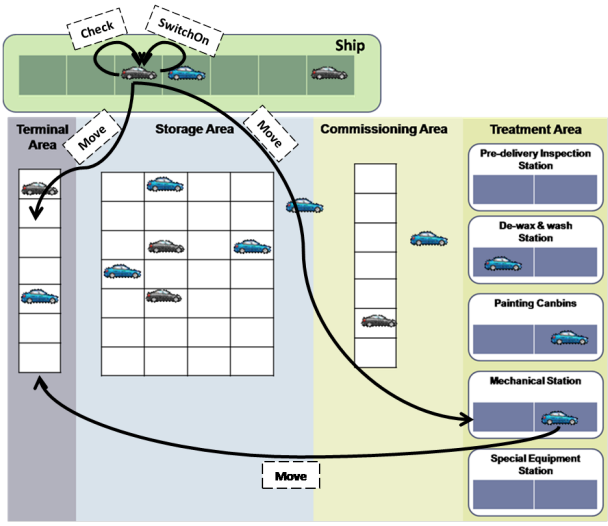


Figure 3.1: A simple domain.

If StatesOf$(\pi) = \{s : \langle s, a \rangle \in \pi\}$ is the set of states in which plan π can be executed, the possible executions of a plan is represented by the execution structure defined below.

Definition 3.1.3 (Execution Structure) *Let π be a plan for a planning domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$. The execution structure induced by π from the set of initial states $\mathcal{I} \subseteq \mathcal{S}$ is a tuple $K = \langle Q, T \rangle$, where $Q \subseteq \mathcal{S}$ and $T \subseteq \mathcal{S} \times \mathcal{S}$ are inductively defined as follows:*

1. *if $s \in \mathcal{I}$, then $s \in Q$, and*
2. *if $s \in Q$ and $\exists \langle s, a \rangle \in \pi$ and $s' \in \mathcal{S}$ such that $\mathcal{R}(s, a, s')$, then $s' \in Q$ and $T(s, s')$.*

A state $s \in Q$ is a terminal state of K if there is no $s' \in Q$ such that T(s,s').

A planning problem is defined by a planning domain $\mathcal{D}$, a set of initial states $\mathcal{I}$ and a set of goal states $\mathcal{G}$.

Definition 3.1.4 (Planning Problem) *Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. A planning problem for $\mathcal{D}$ is a triple $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{I} \subseteq \mathcal{S}$ and $\mathcal{G} \subseteq \mathcal{S}$.*

A planning problem for the domain of Figure 3.1 is the following

- $\mathcal{I} : CarLocation = Ship \wedge CarStatus = Unknown$
- $\mathcal{G} : CarLocation = Terminal$

The set of states are represented as boolean formulas on basic propositions and the intuition of the goals is that the car should move from the "Ship" to the "Terminal". Intuitively, a solution to a planning problem is a plan which can be executed from any state in the set of initial states $\mathcal{I}$ to reach states in the set of goal states $\mathcal{G}$. Due to the non-determinism in the domain, we need to specify the "quality" of the solution by applying additional restrictions on "how" the set of goal states should be reached. In particular we distinguish *weak* and *strong* solutions. A weak solution does not guarantee that the goal will be achieved, it just says that there exists at least one execution path which results in a terminal state that

is a goal state. A strong solution guarantees that the goal will be achieved in spite of non-determinism, i.e., all execution paths of the strong solution always terminate and all terminal states are in a set of goal states.

Definition 3.1.5 (Strong and Weak Solutions)
Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ and $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning domain and problem respectively. Let π be a plan for $\mathcal{D}$ and $K = \langle Q, T \rangle$ be the corresponding execution structure.

1. *π is a strong solution to P if all the paths in K are finite and their terminal states are in $\mathcal{G}$.*
2. *π is a weak solution to P if some of the paths in K terminate with states in $\mathcal{G}$.*

A state-action pair $\langle s, a \rangle \in \pi$ is *strong* if all execution paths from $\langle s, a \rangle$ terminate in the set of goal states. It is *weak* if it is not strong, and at least one execution path from $\langle s, a \rangle$ terminates in the set of goal states. Intuitively, a weak solution contains at least one weak state-action pair, while a strong solution consists of strong state-actions only. Consider the following plan π_1 for our domain:

State	Action
CarLocation = Ship $\wedge$ CarStatus = Unknown	Check
CarLocation = Ship $\wedge$ CarStatus = OK	SwitchOn
CarLocation = Ship $\wedge$ CarStatus = ON	Move

Plan π_1 causes the car to move from the ship to the treatment area (checking and switching on it). It is a weak solution for the planning problem in discussion, indeed the "CarStatus" can non-deterministically become "Damaged" in action "Move", in which case the plan execution terminates without reaching the "Terminal". This planning problem has indeed no strong solutions at all.

3.2 Planning in Non-Deterministic Domains for Extended Goals

Applications in non-deterministic domain require planners to deal with goals that are more general than sets of final desired states. At the same time a plan may result in many possible different executions. In this section we introduce how the classical planning approach presented in Section 3.1 has been extended to be used in non-deterministic domains [21]. The planners for applications in these domains need to generate plans that satisfy conditions on their whole execution paths and to provide it is mandatory to extend the notion of goal. For example, in a Car Logistic domain we may need to specify that a car should *"move to a give location while avoiding certain areas all along the path"*. For this, we might accept plans that the car has a possibility of reaching the location without being guaranteed to do so, but it is however guaranteed to do so, but it is however guaranteed to avoid certain areas. Alternatively, we may require a plan that guarantees that the car reaches the desired treatment location, just trying, if possible, to avoid certain areas.

A *non-deterministic planning domain* can be described in terms of *(basic) propositions*, which may assume different values in different states, of *actions* and of a *transitions relation* describing how an action leads from one state to possible many different states.

Definition 3.2.1 (Non-deterministic Planning Domain) *A non-deterministic planning domain D is a tuple $(B, Q, A, \rightarrow)$, where B is the finite set of (basic) propositions, $Q \subseteq 2^B$ is the set of states, A is the finite set of actions, and $\rightarrow \subseteq Q \times A \times Q$ is the transition relation.*

The transition relation $\rightarrow$ is *total* if for every $q \in Q$ there is some $a \in A$ and $q' \in Q$ such that $q \xrightarrow{a} q'$. We denote with $Act(q)$ the set of the actions that can be performed in state q: $Act(q) = \{a : \exists q'.q \xrightarrow{a} q'\}$. We denote with $Exec(q, a)$ the set of states that can be reached from q performing action $a \in Act(q) : Exec(q, a) = \{q' : q \xrightarrow{a} q'\}$. In Figure 3.2 we depict a simple planning domain, where a vehicle can be moved to a precise location. Actions $Check$ and $Repair$ are non deterministic. $Check$

can either succeed, and lead to state $S2$, or it may fail and lead to state $S3$, where the vehicle need to be repaired. Action $Repair$ can fail and stay at the same state $S3$ to retry or can succeed an lead to state $S3$. The basic propositions are $AtStorage$, $AtTreatment$, $Damaged$, $NotDamaged$. $AtStorage$ holds in state $S4$, $AtTreatment$ in state $S5$, $Damaged$ in state $S3$, $NotDamaged$ in states $S0, S1, S2, S4$ and $S5$.

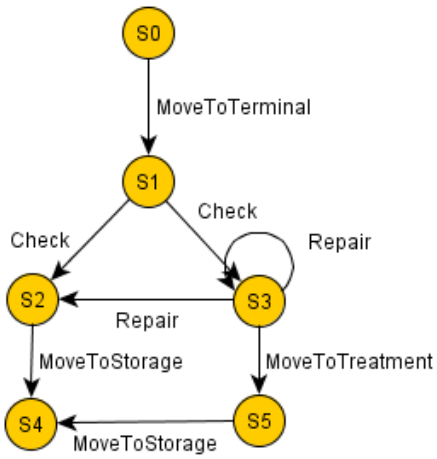


Figure 3.2: A Simple non-deterministic domain.

Extends goals take into account the fact that an action may non-deterministically result in different outcomes. We can express them with CTL [12] formulas that to express temporal conditions. In this way we are able to distinguish between temporal requirements on *"all possible executions"* and on *"some executions"* of a plan.

Definition 3.2.2 (Extended Goals) *Let B be the set of basic propositions of a domain D and let $b \in B$. The syntax of an (extended) goal g for D is the following:*

$$g ::= \top |\ \bot |\ \neg b |\ g \wedge g |\ g \vee g |\ AXg |\ EXg |\ A(gUg) |\ E(gUg) |\ A(gWg) |\ E(gWg)$$

"X", "U", and "W" are the "next time", "(strong) until", and "weak until" temporal operators, respectively. "A" and "E" are the universal and existential path quantifiers, where a path is an infinite sequence of states. They allow to specify requirements that take into account non-determinism. For example, the formula AXg (EXg) means that g holds in every (in some) immediate successor of the current state. $A(g_1Ug_2)$ ($E(g_1Ug_2)$) means that for every path (for some path) there exists an initial prefix of the path such that g_2 holds at the last state of the prefix and g_1 holds at all the other states along the prefix. The formula $A(g_1Wg_2)$ ($E(g_1Wg_2)$) is similar to $A(g_1Ug_2)$ ($E(g_1Ug_2)$) but allows for paths where g_1 holds in all the states and g_2 never holds. Formulas AFg and EFg (where the temporal operator "F" stands for "future" or "eventually") are abbreviations of $A(\top Ug)$ and $E(\top Ug)$, respectively. AGg and EGg (where "G" stands for "globally" or "always") are abbreviations of $A(gW\bot)$ and $E(gW\bot)$, respectively.

Goals as CTL formulas allow to specify different interesting requirements on plans. Let us consider first some example of *reachability goals*. $AF\ g$ ("reach g") states that a condition should be guaranteed to be reached by the plan, in spite of non-determinism. $EF\ g$ ("try to reach g") states that a condition might possibly be reached, i.e., there exists at least one execution that achieves the goal. As an example in Figure 3.2, the strong requirement $AF(AtTreatment)$ cannot be satisfied, while the weaker requirement $EF(AtTreatment)$ can be satisfied only when the car is really damaged. We can distinguish

among different kinds of *maintainability goals*, e.g., $AG\ g$ ("maintain g"), $AG\ \neg g$ ("avoid g"), $EG\ g$ ("try to maintain g"), and $EG\ \neg g$ ("try to avoid g"). For instance, a vehicle should always repaired when damaged. Weaker requirements might be needed for less critical properties, like the fact that the vehicle should try to avoid to move to the treatment area when it is light damaged. Reachability and maintainability requirements goals can be composed. For example, $AF\ AG\ g$ states that a plan should guarantee that all executions reach eventually a set of states where g can be maintained. As a further example, the goal $AG\ EF\ g$ means "maintain the possibility of reaching g". *Reachability - preserving goals* use the "until operators" ($A(g_1 U g_2)$ and $E(g_1 U g_2)$) to express reachability goals while some property must be preserved.

A *plan* describes the actions that have to be performed in a given state of the world. In order to satisfy extended goals, actions that have to be executed may also depend on the "internal state" of the executor, which can take into account, e.g., previous execution steps. In general, a plan can be defined in terms of *action function* that, given a state and an *execution context* encoding the internal state of the executor, specifies the action to be executed, and in terms of a *context function* that, depending on the action outcome, specifies the next execution context.

Definition 3.2.3 (Plan for Extended Goals) *A plan for a domain D is a tuple $\langle C, c_0, act, ctxt\rangle$, where:*

- *C is a set of (execution) contexts,*
- *$c_0 \in C$ is the initial context,*
- *$act : Q \times C \rightharpoonup A$ is the action function,*
- *$ctxt : Q \times C \times Q \rightharpoonup C$ is the context function.*

If we are in state q and in execution context c, then $act(q, c)$ returns the action to be executed by the plan, while $ctxt(q, c, q^{'})$ associates to each reached state $q^{'}$ the new execution context. Functions act and $ctxt$ may be partial, since some state-context pairs never reached in the execution of plan. In the following we define when a plan is executable and when it is complete.

Definition 3.2.4 (Executable Plan) *A plan ϕ is* executable *if, whenever $act(q, c) = a$ and $ctxt(q, c, q^{'}) = c^{'}$, then $q \xrightarrow{a} q^{'}$.*

Definition 3.2.5 (Complete Plan) *A plan ϕ is* complete *if, whenever $act(q, c) = a$ and $ctxt(q, c, q^{'}) = c^{'}$, then there is some context $c^{'}$ such that $ctxt(q, c, q^{'}) = c^{'}$ and $act(q^{'}, c^{'})$ is defined.*

Intuitively, a complete plan always specifies how to proceed for all possible outcomes of any action in the plan. The execution of the plan results in a change in the current state and in the current context. In can therefore be described in terms of transitions between pairs state-context. Formally we have the following definition:

Definition 3.2.6 (Plan Transition) *Given a domain D and a plan ϕ, a transition of a plan ϕ in D is a tuple $(q, c) \xrightarrow{a} (q^{'}, c^{'})$ such that $q \xrightarrow{a} q^{'}$, $a = act(q, c)$, and $c^{'} = ctxt(q, c, q^{'})$*

A *run* of plan ϕ from state q_0 is an infinite sequence $(q_0, c_0) \xrightarrow{a_0} (q_1, c_1) \xrightarrow{a_1} (q_2, c_2) \xrightarrow{a_2} (q_3, c_3) \ldots$ where $(q_i, c_i) \xrightarrow{a_i} (q_{i+1}, c_{i+1})$ are transitions. Given a plan, we may have an infinite number of runs due to the non-determinism of the domain. In the following we define how we can have a finite presentation of the set of all possible runs of a plan with an *execution structure* whose set of states is the set of state-context pairs, and whose transition relation corresponds to the transition of the runs.

Definition 3.2.7 (Execution Structure) *The execution structure of a plan ϕ in a domain D from state q_0 is the structure $K = \langle S, R, L\rangle$, where:*

- $S = \{(q, c) : act(q, c)$ *is defined* $\}$,
- $((q, c), (q', c')) \in Rif(q, c) \xrightarrow{a} (q', c')$ *for some* a,
- $L(q, c) = \{b : b \in q\}$

Finally, we can define the notion of plan that satisfies a given goal:

Definition 3.2.8 (Goal Satisfaction) *Let D be a planning domain and g be a goal for D. Let ϕ be a plan for D and K be the corresponding execution structure. Plan ϕ satisfies goal g from initial state q_0, written $q_0 \models g$, if $K, (q_0, c_0) \models g$. Plan ϕ satisfies goal g from the set of initial states Q_0 if $\phi, q_0 \models g$ for each $q_0 \in Q_0$.*

The main goal of a planning algorithm is to search through the domain by trying to satisfy a goal g in a state s. Goal g defines conditions on the current state and on the next states to be reached. Intuitively, if g must hold in q, then some conditions must be projected to the next states. The planning algorithm extracts the information on the conditions on the next states by "progressing" the goal g. To see details on how the algorithm works we suggest to read the following paper [21].

3.3 Planning with Goal Preferences

In this section we consider a planning problem where actions may have more than one outcome, and it is impossible for the planner to know at planning time which of the different possible outcomes will actually take place at execution time. This problem is called *planning with goal preferences* [23]. Here the user can express preferences over goals and situations, and the planner must generate plans that meet these preferences. Moreover we need to specify that a goal g_1 is better then g_2 and it means that the planner should achieve g_1 in all the cases in which g_1 can be achieved, and it should achieve g_2 in all the other cases. In the following we present the formal notion of conditional planning with some examples using our running example. We start by giving a definition of a reachability goal with preferences and of plans satisfying such a goal.

Definition 3.3.1 (Reachability Goal with Preferences)
A reachability goal with preferences $\mathcal{G}_{list}$ is an ordered list $\langle g_1, ..., g_n \rangle$, where $g_i \subseteq S$. The goals in the list are ordered by preferences, where g_1 is the most preferable goal and g_n is the worst one.

Definition 3.3.2 (Planning Problem With Preferences)
Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. A planning problem with preferences for $\mathcal{D}$ is a triple $\langle \mathcal{D}, \mathcal{I}, \mathcal{G}_{list} \rangle$, where $\mathcal{I} \subseteq \mathcal{S}$ and $\mathcal{G}_{list} = \langle g_1, ..., g_n \rangle$ is a reachability goal with preferences.

Let us consider a goal with preferences for the domain of our running example in Section 2.1 which consists of two preferences goals $\mathcal{G}_{list} = \langle g_1, g_2 \rangle$, where:

- $g_1 = \{CarLocation = Terminal\}$
- $g_2 = \{CarLocation = Treatment\}$

The intuition of this goal is that the car has to move to the terminal or treatment areas, but the terminal is a more preferable and the car has to reach the treatment area only if the terminal become unreachable since that the car becomes damaged and need to be repaired. The planning problem described in the Definition 3.1.4 can be extended to the planning problem with preferences, defined by the triple $\langle D, I, G_{list} \rangle$.

Definition 3.3.3 (Solution) *A plan π is a solution for the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \{g_1, ..., g_n\} \rangle$ if it is a strong solution to the planning problem $P' = \langle \mathcal{D}, \mathcal{I}, \bigvee_{1 \leq i \leq n} g_i \rangle$*

In the definition of the ordering relation among plans, we have to take into account that, due to non-determinism, different goal preferences can be reached by considering different executions of a plan from a given state. Formally, we denote with $\mathit{pref}(\pi, s)^{\mathit{best}} = \min\{i : \exists s' \subseteq g_i$ and s', is a terminal state of the execution structure for π that can be reached from $s\}$ the goal with best preference (i.e., of minimum index) achievable from s. The definition of goal $\mathit{pref}(\pi, s)^{\mathit{worst}}$ with worst preference reachable from s is similar. If $s \notin StatesOf(\pi)$ then $\mathit{pref}(\pi, s)^{\mathit{best}} = \mathit{pref}(\pi, s)^{\mathit{worst}} = -\infty$.

In the following definition, we compare the quality of two plans π_1 and π_2 in a specific states s of the domain. We assume an *optimistic behavior assumption*, i.e., we compare the plans according to the goals of best preferences reached by the plans (i.e., $\mathit{pref}(\pi_1, s)^{\mathit{best}}$ and $\mathit{pref}(\pi_2, s)^{\mathit{best}}$). In case the maximum possible goals are equal, we apply a *pessimistic behavior assumption*, i.e., we compare the plans according to the goals of worst precedence (i.e., $\mathit{pref}(\pi_1, s)^{\mathit{worst}}$ and $\mathit{pref}(\pi_2, s)^{\mathit{worst}}$).

Definition 3.3.4 (Plans Total Ordering Relation in a State) *Let π_1 and π_2 be plans for a problem P. Plan π_1 is better than π_2 in state s, written $\pi_1 \prec_s \pi_2$, if:*

- $\mathit{pref}(\pi_1, s)^{\mathit{best}} < \mathit{pref}(\pi_2, s)^{\mathit{best}}$, *or*
- $\mathit{pref}(\pi_1, s)^{\mathit{best}} = \mathit{pref}(\pi_2, s)^{\mathit{best}}$ *and*
  $\mathit{pref}(\pi_1, s)^{\mathit{worst}} < \mathit{pref}(\pi_2, s)^{\mathit{worst}}$

If $prio(\pi_1, s)^{\max} = prio(\pi_2, s)^{\max}$ and $prio(\pi_1, s)^{\min} = prio(\pi_2, s)^{\min}$ then π_1 and π_2 are equivalent in state s, written $\pi_1 \simeq_s \pi_2$. We write $\pi_1 \preceq_s \pi_2$ if $\pi_1 \prec_s \pi_2$ or $\pi_1 \simeq_s \pi_2$.

We can now define relations between plans π_1 and π_2 by taking into account their behaviors in the common states $S_{\mathit{common}}(\pi_1, \pi_2) = StatesOf(\pi_1) \cap StatesOf(\pi_2)$.

Definition 3.3.5 (Plans Ordering Relation) *Let π_1 and π_2 be plans for a problem P. Plan π_1 is better than plan π_2, written $\pi_1 \prec \pi_2$, if:*

- $\pi_1 \preceq_s \pi_2$ *for all states* $s \in S_{\mathit{common}}(\pi_1, \pi_2)$, *and*
- $\pi_1 \prec_{s'} \pi_2$ *for some state* $s' \in S_{\mathit{common}}(\pi_1, \pi_2)$.

If $\pi_1 \simeq_s \pi_2$ for all $s \in S_{\mathit{common}}(\pi_1, \pi_2)$, then the two plans are equally good, written $\pi_1 \simeq \pi_2$.

Notice that the plans ordering relation is not total: two plans are incomparable if there exist states $s_1, s_2 \in S_{\mathit{common}}(\pi_1, \pi_2)$ such that $\pi_1 \prec_{s_1} \pi_2$ and $\pi_2 \prec_{s_2} \pi_1$. However, in this case we can construct a plan π such that $\pi \prec \pi_1$ and $\pi \prec \pi_2$, as follows:

- if $\langle s, a\rangle \in \pi_1$ and either $s \notin StatesOf(\pi_2)$ or $\pi_1 \prec_s \pi_2$, then $\langle s, a\rangle \in \pi$;
- if $\langle s, a\rangle \in \pi_2$ and either $s \notin StatesOf(\pi_1)$ or $\pi_2 \prec_s \pi_1$, then $\langle s, a\rangle \in \pi$.

As a consequence, there exists a plan which is better or equal to any other plan and we call such plan *optimal*.

Definition 3.3.6 (Optimal Plan) *Plan π for problem P is an optimal plan if $\pi \preceq \pi'$ for any another plan π' for the same problem P.*

The planning algorithm to solve a planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G}_{list}\rangle$ consists of building a state-action table π_i for each goal g_i, and then to merge them in a single state-action table π. To see details on how the algorithm works we suggest to read the following paper [23].

3.4 Automated Composition of Services by Planning

The starting point for the emergence of a Service Oriented Computing paradigm has been the adoption of standards for the publication and access of services over the Web. A main feature of services is the reuse mechanism to build new applications, which often need to be defined out of finer-grained subtasks that are likely available as services again. Composition rules describe how to compose coherent global services. In particular, they specify the order in which, and the conditions under which, services may be invoked. In most real-life scenarios, services are stateful, realizing complex protocols; their behavior may be non-deterministic, and they may exchange messages asynchronously. This makes the composition problem significantly more complex, and requires specific ways to manage these features. In this chapter we present how AI planning techniques can been used to compose services in an automatic way. In each approach that we present, the planning problem corresponds to the automatic composition of services that are published as processes. We start presenting a classical approach and we continue introducing two different ways to describe the composition requirements based on control-flow and data-flow.

3.4.1 Classical Approach

Planning is one of the most promising techniques for the automated composition of web services. The approach that we present in this section is based on [7, 20] and is focused on a form of automated process-level composition of Web services where services can be represented as finite state automata, and composition requirements command the finite termination of (the execution of) component services. The goal of the approach is to automatically generate a new service W (called the *composite service*) that interacts with a set of published web services $W_1, \ldots, W_n$ (called the component services) and satisfies a given composition requirement. More specifically (see Figure 3.3) component services are described as WS-BPEL abstract processes [1]. Given n WS-BPEL abstract processes $W_1, \ldots, W_n$, the BPEL2STS module automatically translates each of them into a *state transition system* (STS) $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. Intuitively, each Σ_{W_i} is a compact representation of all the possible behaviors, evolutions of the component service W_i. Each Σ_{W_i} is described in terms of states, input and output actions, and internal actions. We then construct a *parallel STS* $\Sigma_{\|}$ that combines $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. Formally, this combination is a parallel product, which allows the n services to evolve concurrently. $\Sigma_{\|}$ represents therefore all the possible behaviors, evolutions of the different component services, without any control by and interaction with the composite service that will be generated, i.e., W. From $\Sigma_{\|}$, we generate a planning domain $\mathcal{D}$ that is passed in input to the planner (module MBP). The second kind of input to the planner consists of the requirements for the composite service. They are formalizes as a goal ρ in EAGLE, a language for expressing extended planning goals [16]. Is important to precise that the framework presented here also works with other kinds of extended goals, e.g. with CTL [12] goals. Given $\mathcal{D}$ and ρ, the component MBP generates a plan ϕ that is then translated into a STS Σ_c. Σ_c encodes the new service W that has to be generated, which dynamically receives and sends invocations from/to the composite services $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ and behaves depending on responses received from the external services. Σ_c is such that $\Sigma_c \rhd \Sigma_{\|}$ represents all the evolutions of the component services as they are controlled by the composite service. The STS Σ_c is then given in input to the STS2BPEL module which translates it into a concrete $WS-BPEL$ process that implements the desired composite web service.

The automated composition problem has two inputs (see Figure 3.3): the formal composition requirement ρ and the parallel STS $\Sigma_{\|}$, which represents the services $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. In the following we introduce the formal definition of the *parallel product* of two STS, which models the fact that both systems may evolve independently, and which is used to generate $\Sigma_{\|}$ from the component web services.

Definition 3.4.1 (Parallel Product) *Let* $\Sigma_1 = \langle \mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{R}_1, \mathcal{F}_1 \rangle$ *and* $\Sigma_2 = \langle \mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{R}_2, \mathcal{F}_2 \rangle$ *be two STSs with* $(\mathcal{I}_1 \cup \mathcal{O}_1) \cap (\mathcal{I}_2 \cup \mathcal{O}_2) = \emptyset$. *The parallel product* $\Sigma_1 \parallel \Sigma_2$ *of* Σ_1 *and* Σ_2 *is defined as:*

$$\Sigma_1 \| \Sigma_2 = \langle \mathcal{S}_1 \times \mathcal{S}_2, \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{R}_1 \| \mathcal{R}_2, \mathcal{F}_1 \| \mathcal{F}_2 \rangle$$

where:

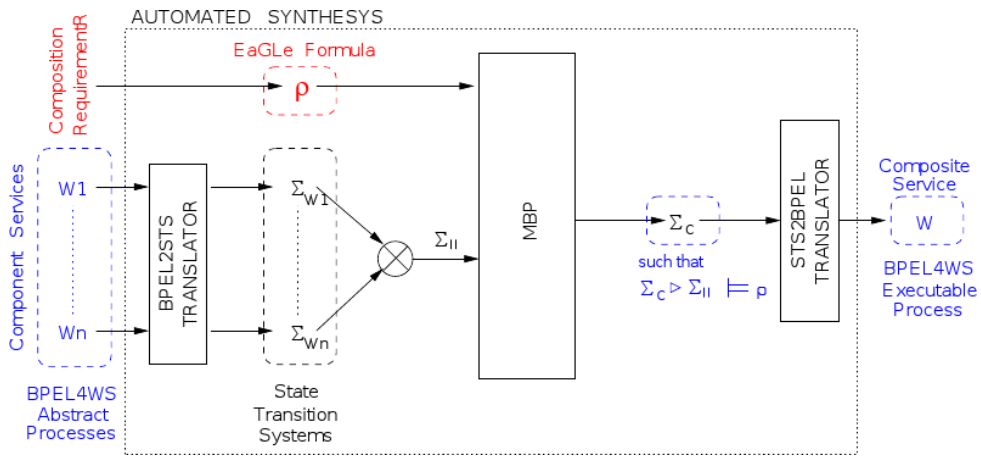


Figure 3.3: The Approach.

- $\langle (s_1, s_2), a, (s_1', s_2) \rangle \in (\mathcal{R}_1 \| \mathcal{R}_2)$ *if* $\langle s_1, a, s_1' \rangle \in \mathcal{R}_1$*;*
- $\langle (s_1, s_2), a, (s_1, s_2') \rangle \in (\mathcal{R}_1 \| \mathcal{R}_2)$ *if* $\langle s_2, a, s_2' \rangle \in \mathcal{R}_2$*;*

and $(\mathcal{F}_1 \| \mathcal{F}_2)(s_1, s_2) = \mathcal{F}_1(s_1) \cup \mathcal{F}_2(s_2)$.

The system representing (the parallel evolutions of) the component services $W_1, \ldots, W_n$ of Figure 3.3 is formally defined as $\Sigma_{\|} = \Sigma_{W_1} \| \ldots \| \Sigma_{W_n}$.

The automated composition problem consists in generating a STS Σ_c that controls $\Sigma_{\|}$ so that its executions satisfy the requirement ρ. In the following definition we define formally the behaviors of a STS σ when controlled by σ_c.

Definition 3.4.2 (Controlled System) *Let* $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{F} \rangle$ *and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{F}_\emptyset \rangle$ *be two state transition systems, where* $\mathcal{F}_\emptyset(s_c) = \emptyset$ *for all* $s_c \in \mathcal{S}_c$*. The STS* $\Sigma_c \triangleright \Sigma$*, describing the behaviors of system* Σ *when controlled by* Σ_c*, is defined as:*

$$\Sigma_c \triangleright \Sigma = \langle \mathcal{S}_c \times \mathcal{S}, \mathcal{S}_c^0 \times \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c \triangleright \mathcal{R}, \mathcal{F} \rangle$$

where:

- $\langle (s_c, s), \tau, (s_c', s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s_c, \tau, s_c' \rangle \in \mathcal{R}_c$*;*
- $\langle (s_c, s), \tau, (s_c, s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s, \tau, s' \rangle \in \mathcal{R}$*;*
- $\langle (s_c, s), a, (s_c', s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$*, with* $a \neq \tau$*, if* $\langle s_c, a, s_c' \rangle \in \mathcal{R}_c$ *and* $\langle s, a, s' \rangle \in \mathcal{R}$*.*

A STS Σ_c may not be adequate to control a system Σ. Indeed, we need to guarantee that, whenever Σ_c performs an output transition, then Σ is able to accept it, and vice-versa.

We define the condition under which a state s of Σ is able to accept a message according to our asynchronous model, which abstracts away queues. We assume that s can accept a message a if there is some successor s' of s in Σ, reachable from s through a chain of τ transitions, such that s can perform an input transition labelled with a. Vice-versa, if state s has no such successor s', and message a is sent to Σ, then a deadlock situation is reached.

In the following definition, and in the rest of the paper, we denote by τ-closure(s) the set of the states reachable from s through a sequence of τ transitions, and by τ-closure(S) with $S \subseteq \mathcal{S}$ the union of τ-closure(s) on all $s \in S$.

Definition 3.4.3 (Deadlock-free Controller) *Let* $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{F} \rangle$ *be a STS and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{F}_\emptyset \rangle$ *be a controller for* Σ*.* Σ_c *is said to be* deadlock free *for* Σ *if all states* $(s_c, s) \in \mathcal{S}_c \times \mathcal{S}$ *that are reachable from the initial states of* $\Sigma_c \triangleright \Sigma$ *satisfy the following conditions:*

- *if* $\langle s, a, s' \rangle \in \mathcal{R}$ *with* $a \in \mathcal{O}$ *then there is some* $s'_c \in \tau$*-closure*(s_c) *such that* $\langle s'_c, a, s''_c \rangle \in \mathcal{R}$ *for some* $s''_c \in \mathcal{S}_c$*; and*
- *if* $\langle s_c, a, s'_c \rangle \in \mathcal{R}_c$ *with* $a \in \mathcal{I}$ *then there is some* $s' \in \tau$*-closure*(s) *such that* $\langle s', a, s'' \rangle \in \mathcal{R}$ *for some* $s'' \in \mathcal{S}$*.*

A controller is a solution for the requirement ρ iff it guarantees that ρ is achieved. We can formally express this by requiring that every execution of the controlled system $\Sigma_c \triangleright \Sigma$ ends up in a state where ρ holds.

Definition 3.4.4 (Satisfiability) *An STS* Γ *satisfies a requirement* ρ*, denoted with* $\Gamma \models \rho$*, if and only if*

- *there exists no infinite run of* Γ*;*
- *every final state of* Γ *satisfies* ρ *according to Definition* **??**.

Definition 3.4.5 (Solution Controller) *A controller* Γ_c *is a solution for goal* ρ *iff* $\Sigma_c \triangleright \Sigma \models \rho$*.*

We can now formally characterize a (Web Service) composition problem as follows:

Definition 3.4.6 (Composition Problem) *Let* $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ *be a set of state transitions systems, and let* ρ *be a composition requirement. The composition problem for* $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ *and* ρ *is the problem of finding a deadlock-free controller* Γ_c *such that* $\Sigma_c \triangleright (\Sigma_1, \ldots, \Sigma_n) \models \rho$*.*

3.4.2 Control Flow Requirements for Service Composition

In this section we present an extension of the classical approach [6], presented in the section above, that resolves around the central notion of a *domain object*, which is introduced to explicitly model the key elements of the composition problem and their evolution. Objects may have a complex life-cycle: they may be created, modified and deleted. The idea is that, while activities performed by component services may make objects evolve, the modeling of the objects does not depend on a particular service implementation. This makes it natural to express *control-flow composition requirements* in terms of the domain objects and their evolution. Specifically, the evolution of an object is modeled, which includes its creation and deletion as well as reactions to specific actions, with a state diagram, which defines possible object states and transitions between them. The transitions correspond to the activities that can be performed over the object and to external events affecting the state of the object. To link objects to services, the services activities are annotated with elements of the object diagrams, implicitly defining a mapping between the execution of service operations and the evolution of objects. In this way it becomes easy to modify a scenario to account for different services implementations: it is enough that services are annotated appropriately, while it is not necessary to affects object models nor requirements on them. Indeed, given objects and labeled services, control-flow composition requirements are defined on top of the object state diagrams. Object states and events are used to represent both the tasks to be performed over the objects, and to specify coordination requirements defining a consistent evolution of sets of related objects. Also *data flow requirements* must be considered, to define data dependencies between various activities. Next section will be focused on this while here we consider only control-flow requirements. Once domain objects and composition requirements are specified and component services are annotated, the specifications are converted into a formal representation which is passed to a synthesis engine. Such engine automatically identifies and generates a composite service that satisfies the composition requirements by orchestrating the component services. The engine is based on the asynchronous planning framework presented in Section 3.4.1. In the following sections we present the elements, that are part of the framework depicted in Figure 3.5, for modeling a service composition problem, that are, the domain objects and their evolution, their associated services, and the corresponding control-flow composition requirements. After these, the approach to compose services in automatic way, that use them, is presented. Formally, domain objects are represented with *object diagrams*.

Definition 3.4.7 (Object Diagram)
An object diagram *representing a domain object O is a tuple $\langle L, L_0, \mathcal{E}, T\rangle$, where*

- *L is a finite set of object configurations and $L_0 \subseteq L$ is a set of initial configurations;*
- *$\mathcal{E}$ is a set of possible events that reflect the evolution of the object;*
- *$T \subseteq L \times \mathcal{E}^+ \times L$ is a transition relation that defines the evolution of an object, based on events.*

There exists a predefined event $to^{\mathsf{e}}(l, o)$, with $l \in L$, to define that the object o moves to a configuration l. Moreover, is important that transitions leaving a configuration are annotated with mutually disjoint sets of events. The object diagram of Figure 3.4 contains four configurations, namely *SHIP*, *TREATMENT*, *TERMINAL* and *UNKNOWN*. The object moves to the configuration *TERMINAL* upon the event "MoveToTerminal", while it moves to a configuration *UNKNOWN* upon the event "MoveToUnknown", etc...

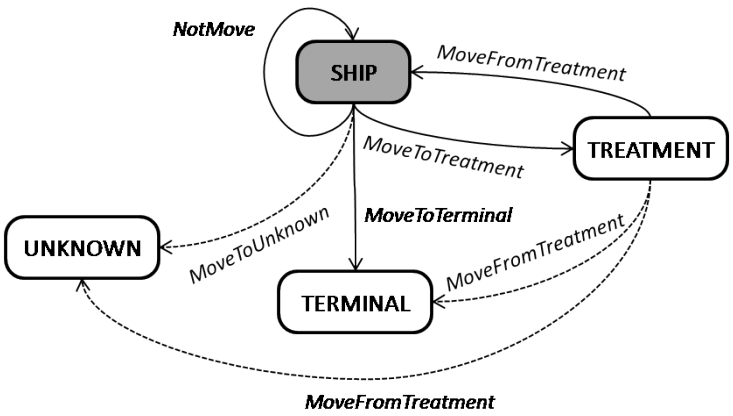


Figure 3.4: Example of Object Diagram.

Services and Service Annotations

We assume that the description of the services associated with the considered domain objects consists of a stateful service protocol (e.g., a WS-BPEL process), associated to a stateless service interface (e.g., a WSDL document). Each service description is related with a corresponding object and its dynamics, through special annotations. These annotations appear within the activities of the service protocol: an activity may be annotated with a set of events pertaining to the corresponding object. This implicitly defines how the evolution of the service reflects over the object. Formally, we model services as *annotated state transition system* (ASTS), similarly to the STS defined in Section 3.1. The transitions of ASTS may be labelled with *object events*, thus stating that when the transition takes place, the corresponding object is changed. If, for example, the transition is annotated with an event $to^{\mathsf{e}}(l, o)$, then the object o moves to a configuration l when this transition takes place.

Definition 3.4.8 (Annotated State Transition System) *An annotated STS Σ is a tuple $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{E}, \mathcal{R}\rangle$ where:*

- *$\mathcal{E}$ is the set of events;*
- *$\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{E}^* \times \mathcal{S}$ is the transition relation.*

The semantics of the annotated transition is intuitively described as follows. Assume an object o with a set of events $\mathcal{E}$, and a service transition $(s, a, \varepsilon, s') \in \mathcal{R}$, where $s, s' \in \mathcal{S}$, $a \in (\mathcal{I} \cup \mathcal{O} \cup \{\tau\})$, and

$\varepsilon \subseteq \mathcal{E}$. We say that the transition is *applicable* to the object o if the object is in some configuration l, and either $\varepsilon = \emptyset$, or there exists an object transition (l, ε', l'), such that $\varepsilon' \subseteq \varepsilon$. As a result of performing this transition, in the first case the object will remain in the same configuration, while in the second case it will evolve to the configuration l'.

Composition Requirements

We now present a simple language that allows fulfilling our desiderata to model in an easy-to-specify, compositional and implementation-independent way complex requirements, namely (i) in which "stable" situations we intend to see our objects, possibly ordered according certain preferences; (ii) requirements on the evolution of objects, linking the behaviors of different objects; and (iii) reaction rules, which define how the composed service shall react to object events in different situations.

Definition 3.4.9 (Composition Requirement) *A composition requirement is defined with the following generic constraint template*

$$clause \Longrightarrow (clause_1 \succ \ \ldots \ \succ clause_n),$$

where $clause \equiv \top \mid s^{\mathsf{s}}(o) \mid e^{\mathsf{e}}(o) \mid cl_1 \vee cl_2 \mid cl_1 \wedge cl_2$.

Here cl_1 and cl_2 are clauses, $s^{\mathsf{s}}(o)$ is used to define the fact that the object o is in the configuration s, and $e^{\mathsf{e}}(o)$ defines that the event e of the object o has taken place. The left side of the constraint defines the "premise" of the requirement. In case it is empty (i.e., defined as $\top$), the requirement expresses the need to unconditionally reach a particular state or to achieve a particular effect, defined by the right side. Otherwise, it defines a "reaction rule": whenever the corresponding situation or events take place, the composite service should try to "recover" from it by achieving the effects/situations defined by the right side. In both cases, the right side of the constraint defines the expected results. Each of them logically groups simpler results, which may express either a certain state (require to reach a configuration) or a certain effect (require that an event happens). These results are ordered according to the order of preference denoted by the $\succ$ symbol, from the most preferred to the least preferred.

Automated Service Composition

The requirements, defined above, impose constraints on the evolution of objects, which must be achieved by executing services associated to those objects. In order to recast this in terms of planning, these objects and the ASTSs of the component services must be transformed into state transition systems, and the composition requirements must be transformed in terms of the states of such STSs. In the following, the transformation of services, objects and composition requirements are presented in turn.

The transformation of the component services (i.e., ASTSs) is performed as follows. Given an ASTS $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{E}, \mathcal{R} \rangle$, the corresponding STS is a tuple $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}', \mathcal{S}^F, \mathcal{F} \rangle$, where for each transition $(s, a, \varepsilon, s') \in \mathcal{R}$ exists a corresponding (non-guarded) transition $(s, \top, a, s') \in \mathcal{R}'$, and the states are not labeled (for each $s \in \mathcal{S}$ $\mathcal{F}(s) = \emptyset$). In order to require that the service protocols are either unused or fully completed, all the terminating and initial states of the ASTS are marked as accepting.

The transformation of the object diagram into STS is more complex. First, to capture the states of the objects in the requirements, there is a set of atomic propositions $\mathcal{L} \equiv \{{s_j}^{\mathsf{s}}(o_i)\}$, which specify that an object o_i is in state s_j for all objects and their states. Then, given an object diagram $\langle L, L_0, \mathcal{E}, T \rangle$ there is an STS $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$, where $\mathcal{S} = L$, $\mathcal{S}^0 = L_0$, each state is labelled with the corresponding proposition (i.e., $\forall s \in \mathcal{S} : \mathcal{F}(s) = \{s^{\mathsf{s}}(o)\}$), and all object configurations are accepting (i.e., $\mathcal{S}^F = \mathcal{S}$). Moreover, the transition relation to capture the effects of the evolution of component services on the object is defined as: for each $(l, \varepsilon', l') \in T$ and for any transition (s, a, ε, s') of some ASTS such that $\varepsilon' \subseteq \varepsilon$ we define a (non-guarded) transition $(l, \top, a, l') \in \mathcal{R}$.
The composition requirements speak of both the object states and of occurrences of object events. In order to capture this information, for every requirement there is a corresponding STS that reflects the

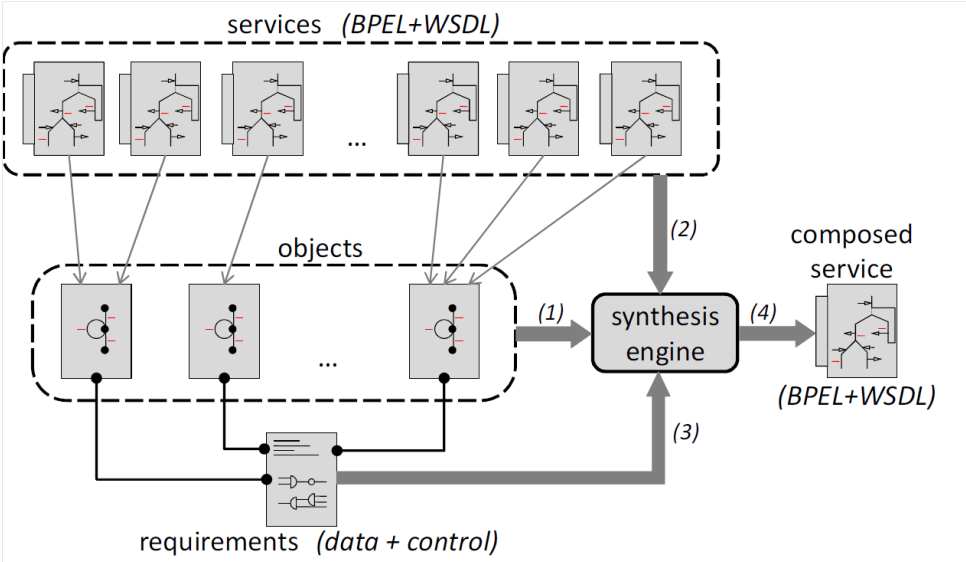


Figure 3.5: Composition framework.

satisfiability of the requirement. Given a clause cl, we define a corresponding STS that contains a single output action e_{cl} representing the completion of the clause. The diagrams corresponding to the different clauses, to their combinations, and to the representing diagram itself are represented in Fig. 3.6.

Intuitively, they have the following meaning.

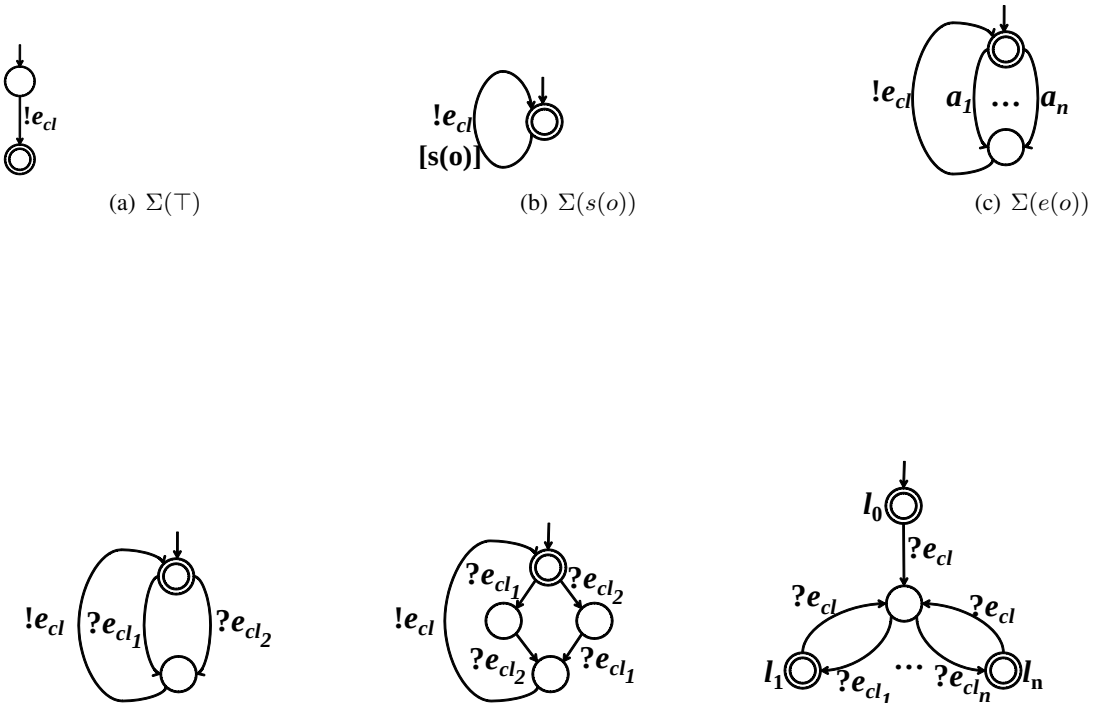
- The STS for the $\top$ clause (Fig.3.6(a)) is completed immediately.
- The STS for $s^{\mathsf{s}}(o)$ (Fig.3.6(b)) is blocked until the object is not in the required state: the transition is guarded with the corresponding proposition.
- The STS for $e^{\mathsf{e}}(o)$ (Fig.3.6(c)) waits for any of the service actions that contain the corresponding event in its effects: for any transition (s, a, ε, s') of some ASTS such that $e^{\mathsf{e}}(o) \in \varepsilon$ a corresponding transition is defined. When it happens, a completion is reported.
- The STS for $cl_1 \vee cl_2$ (Fig.3.6(d)) waits for any of the sub-clauses to complete, while the STS of the $cl_1 \wedge cl_2$ (Fig.3.6(e)) waits for both of them to be completed.

The STS that represents the evolution of a composition requirement is represented in Fig. 3.6(f). The STS is initially in an accepting location (l_0). If the premise takes place (e_{cl} is reported), then it moves to a non-accepting state, from which it may be satisfied by completing one of the clauses $e_{cl_1}, \ldots, e_{cl_n}$ (moving to locations $l_1, \ldots, l_n$ respectively). The corresponding goal with preferences will have the following form:

$$\rho_c = (l_0, l_1, \ldots, l_n). \tag{3.1}$$

That is, we require that whenever the premise take place, the composition tries to move the STS to one of the accepting states, respecting the ordering of preferences. To integrate the approach described above into the automated composition framework described in Section 3.4.1, we essentially need to include the STS-encoded object diagrams and composition requirements within the composition domain.

In particular, given n composite services $W_1, \ldots, W_n$, m objects $O_1, \ldots, O_m$ and k (event) composition requirements $C_1, \ldots, C_k$ we encode each component service W_i as the corresponding state transition systems Σ_{W_i} (Fig. 3.5, step 2); each object diagram O_i as Σ_{O_i} (Fig. 3.5, step 1); each composition requirement C_i as Σ_{C_i} (Fig. 3.5, step 3). The translation is defined according to the rules presented above. Then, we build a planning domain and goal. Namely, the planning domain Σ is defined as a synchronous product of the all the STSs of the component services, objects, and requirements, and the



(a) $\Sigma(\top)$ (b) $\Sigma(s(o))$ (c) $\Sigma(e(o))$

(d) $\Sigma(cl_1 \vee cl_2)$ (e) $\Sigma(cl_1 \wedge cl_2)$ (f) $\Sigma(cl \Longrightarrow (cl_1 \succ \ldots \succ cl_n))$

Figure 3.6: STS diagrams of composition requirements

composition goal is constructed from the requirements defined according to the formula (3.1):

$$\Sigma = \Sigma_{W_1} \| \ldots \| \Sigma_{W_n} \| \Sigma_{O_1} \| \ldots \| \Sigma_{O_m} \| \Sigma_{C_1} \| \ldots \| \Sigma_{C_k}$$

$$\rho = \bigwedge_c \rho_c.$$

Finally, given the domain Σ and the planning goal ρ, we apply the approach presented in Section 3.4.1 to generate a controller Σ_c, which is such that $\Sigma_c \triangleright \Sigma \models \rho$. Once the state transition system Σ_c has been generated, it is translated into executable WS-BPEL process to obtain the new process which implements the required composition (Fig. 3.5, step 4). The translation is conceptually simple; intuitively, input actions in Σ_c model the receiving of a message from a component service, output actions in Σ_c model the sending of a message to a component service.

3.4.3 Data-Flow Requirements for Service Composition

In this section we present another way to compose services which addresses one of the key aspects of composition requirements, namely the data flow among the component services. We present the graphical notations and their formal theory [19], moreover we show how this new model can be integrated in the general composition framework already presented in Section 3.4.1. The aim of the data flow modeling

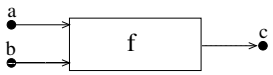
language is to allow for the specification of complex requirements concerning data manipulation and exchange. In particular, data flow requirements specify how output messages (messages sent to component services) are obtained from input messages (messages received from component services). This includes several important aspects: whether an input message can be used several times or just once, how several input messages must be combined to obtain an output message, whether all messages received must be processed and sent, etc.. In the following the basic elements of the language are described. We show also how they can be composed to obtain complex expressions with their intuitive semantics.

- *Connection Node*

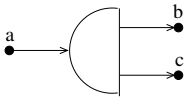
  A *connection node* can be external or internal. An external connection node is associated to an output (or an input) external port. Intuitively, an external input (output) node characterizes an external source (target) of data and it is used to exchange data with the outside world.

- *Identity*
  It is connected to one connection node in input and one node in output. The requirement states that data received from the input node should be forwarded to the output node. The graphical notation for the data-flow identity element `id(a)(b)`, with input node `a` and output node `b`, is the following:

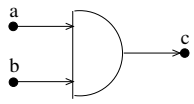
- *Operation*
  It is related to a function definition; it is connected to as many input nodes as the number of function parameters and only to one output node corresponding to the function result. The requirement states that, when data is received from all the input nodes, the result of the operation should be forwarded to the output node. The graphical notation for the data-flow operation element `oper[f](a,b)(c)` characterizing function `f`, with input nodes `a` and `b` and output node `c`, is the following:



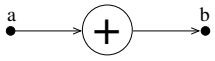
- *Fork*
  It is connected to a node in input and to as many nodes as necessary in output. It forwards data received on the input node to all the output nodes. The graphical notation for the data-flow fork element `fork(a)(b,c)`, with input node `a` and output nodes `b` and `c`, is the following:



- *Merge*
  It is connected to one node in output and as many nodes as necessary in input. It forwards data received on some input node to the output node. It preserves the temporal order of data arriving on input nodes (if it receives data on two or more input nodes at the same time, the order is nondeterministic). We represent the data-flow merge element `merge(a,b)(c)`, with input nodes `a` and `b` and output node `c` as:

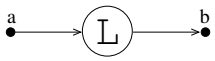


- *Cloner*
  It is connected to one node in input and one node in output. It forwards, one or more times, data received from the input node to the output node. The data-flow cloner element `clone(a)(b)`, with input node `a` and output node `b` is represented as:



- *Filter*
  It is connected to one node in input and one node in output. When it receives data on the input node, it either forwards it to the output node or discards it. We represent the data-flow filter element `filt(a)(b)`, having input node `a` and output node `b` as:

- *Last*
  It is connected to one node in input and one node in output. It requires that at most one data is forwarded to the output node: the last data received on the input node. All other data that are received should be discarded. The graphical notation for the data-flow last element `last(a)(b)`, with input node `a` and output node `b`, is the following:



The diagram obtained by suitably composing data-flow elements by means of connection nodes is called *data net*. A data net is characterized by a set of external connection nodes associated to input ports N^I_{ext}, a set of external connection nodes associated to output ports N^O_{ext}, a set of internal connection nodes N_{int}, a set of data-flow elements D (corresponding to the basic elements described in this section) and a set of data values V. Given a data-flow element d, we denote with $in\_nodes(d)$ the set of input connection nodes of d and with $out\_nodes(d)$ the set of output connection nodes of d.

Definition 3.4.10 (Data Net) *A* data net $\mathcal{D}$ *is a tuple* $\langle N^I_{ext}, N^O_{ext}, N_{int}, D, V\rangle$ *where:*

- *for each* $n \in N^I_{ext}$ *there exists a unique data-flow element* $d \in D$ *s.t.* $n \in in\_nodes(d)$*;*
- *for each* $n \in N^O_{ext}$ *there exists a unique data-flow element* $d \in D$ *s.t.* $n \in out\_nodes(d)$*;*
- *for each* $n \in N_{int}$ *there exists a unique data-flow element* $d_1 \in D$ *s.t.* $n \in in\_nodes(d_1)$ *and there exists a unique data-flow element* $d_2 \in D$ *s.t.* $n \in out\_nodes(d_2)$*;*
- *for each* $d \in D$*,* $in\_nodes(d) \subseteq N^I_{ext} \cup N_{int}$ *and* $out\_nodes(d) \subseteq N^O_{ext} \cup N_{int}$*.*

Notice that it is possible to associate a type to each connection node in the network. Indeed, external nodes inherit the types from the corresponding WS-BPEL ports, and the types of internal nodes can be deduced from the structure of the data net. We do not consider this aspect to make the formalization more understandable; completing the model to handle typed connection nodes is straightforward.

Semantics

We now formalize the semantics of the data flow modeling language. Given a data net $\mathcal{D} = \langle N^I_{ext}, N^O_{ext}, N_{int}, D, V \rangle$, we denote with N_{ext} the sets of all external connection nodes, formally $N_{ext} = N^I_{ext} \cup N^O_{ext}$. An *event* e of $\mathcal{D}$ is a couple $\langle n, v \rangle$, where $n \in N_{ext} \cup N_{int}$, and $v \in V$, which models the fact that the data value v passes through the connection node n. An *execution* ρ of $\mathcal{D}$ is a finite sequence of events $e_0, ..., e_n$. Given an execution ρ we define its projection on a set of connection nodes $N \subseteq N_{ext} \cup N_{int}$, and denote it with $\Pi_N(\rho)$, the ordered sequence $e'_0, ..e'_m$ representing the events in ρ which correspond to nodes in N.

We formally define the semantics of our language in terms of *accepted executions* of a data net $\mathcal{D}$. In the following definition, we exploit regular expressions to define the accepted execution. We use notation $\Sigma_{v \in V}$ to express alternatives that range over all the possible values $v \in V$ that can flow thorough the net.

Definition 3.4.11 (Data Net Accepting Execution) *An execution ρ is accepted by a data net $\mathcal{D} = \langle N^I_{ext}, N^O_{ext}, N_{int}, D, V \rangle$ if it satisfies all the following properties:*

- *for each identity element* `id(a)(b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v \in V} \langle a, v \rangle \cdot \langle b, v \rangle \right)^*$$

- *for each operation element* `oper[f](a,b)(c)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b,c\}}(\rho) = \left(\sum_{v,w \in V} (\langle a, v \rangle \cdot \langle b, w \rangle + \langle b, w \rangle \cdot \langle a, v \rangle) \cdot \langle c, f(v, w) \rangle \right)^*$$

- *for each fork element* `fork(a)(b,c)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b,c\}}(\rho) = \left(\sum_{v \in V} \langle a, v \rangle \cdot (\langle b, v \rangle \cdot \langle c, v \rangle + \langle c, v \rangle \cdot \langle b, v \rangle) \right)^*$$

- *for each merge element* `merge(a,b)(c)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b,c\}}(\rho) = \left(\sum_{v \in V} (\langle a, v \rangle \cdot \langle c, v \rangle + \langle b, v \rangle \cdot \langle c, v \rangle) \right)^*$$

- *for each cloner element* `clone(a)(b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v \in V} \langle a, v \rangle \cdot \langle b, v \rangle \cdot \langle b, v \rangle^* \right)^*$$

- *for each filter element* `filt(a)(b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v \in V} \langle a, v \rangle \cdot (\langle b, v \rangle + \epsilon) \right)^*$$

- *for each last element* `last(a)(b)` *in* $\mathcal{D}$*:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v \in V} \langle a, v \rangle\right)^* \cdot \left(\sum_{v \in V} \langle a, v \rangle \cdot \langle b, v \rangle\right) + \epsilon$$

Notice that this definition considers data net elements having at most two input/output nodes, however it can easily be extended to handle elements of the data net having more input/output nodes.

Data Net Satisfiability

A data net can be used to specify the desired behavior of a service or everything that concerns the exchange of data with its communication partners. In particular external connection nodes are associated to input (or output) ports which model WS-BPEL messages, or message parts, which are used to store data received (or sent) by the process while interacting with other services.

Since the behavioral aspect we are interested in concerns the data flow among the process and its partners, we characterize an execution of a WS-BPEL process W, denoted with $exec(W)$, as the set of all possible ordered sequence of input/output messages (or message parts) received and sent by the process from its activation to its termination. Notice that each message carries both the information about the external port on which it has been sent/received and about its content (value).

Definition 3.4.12 (Data Net Satisfiability)*Let W be a* WS-BPEL *process and $\mathcal{D} = \langle N^I_{ext}, N^O_{ext}, N_{int}, D, V \rangle$ a data net. We say that W* satisfies *$\mathcal{D}$ if for each process execution $\rho_W \in exec(W)$ there exists an accepting execution ρ of $\mathcal{D}$ such that $\Pi_{N_{ext}}(\rho) = \rho_W$.*

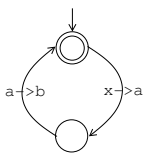
Data Requirements as STSs

As we have seen in previous sections, a data net $\mathcal{D}$ of a particular composition problem specifies how messages received from the component services can be used by the new composite process to generate outgoing messages. Therefore, it is possible to represent $\mathcal{D}$ as a STS $\Sigma_{\mathcal{D}}$, which models the allowed data flow actions. In particular, input actions in $\Sigma_{\mathcal{D}}$ represent messages received by the component services, output actions represent messages sent by the component services and internal actions represent assignments that the composite process performs on its internal variables.

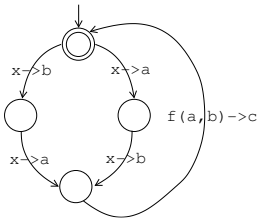
We assume that, in the WS-BPEL specification of the composite service, a variable will exist for each connection node in $\mathcal{D}$; variables associated to external connection nodes are those used by the new composite process to store received messages and to prepare the messages to be sent, while variables associated to internal connection nodes are those used to manipulate messages by means of internal functions and assignments. Then $\Sigma_{\mathcal{D}}$ defines constraints on the possible operations that the composite process can perform on these variables. A nice feature of our approach is that this can be done compositionally, i.e., a "small" automaton can be associated to each element of the data net, and STS $\Sigma_{\mathcal{D}}$ is obtained as the product of all these small automata.

More precisely, for each output operation of a component service, which is associated to some external input port in the data net, we define a STS which represents the sending of the message (as an output action) and the storing of all message parts (as internal actions). Finally, we define a STS for each data-flow element of the data net. These STSs have no input/ouput actions since they model manipulation of variables through assignments. In particular:

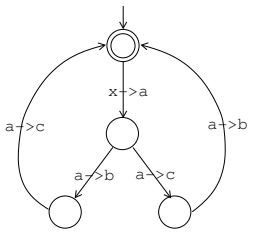
- for each identity element `id(a)(b)` in the data net we define the following STS:



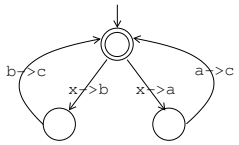
- for each operation element `oper[f](a,b)(c)` in the data net we define the following STS:



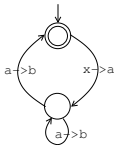
- for each fork element `fork(a)(b,c)` in the data net we define the following STS:



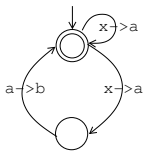
- for each merge element `merge(a,b)(c)` in the data net we define the following STS:



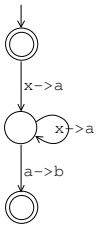
- for each cloner element `clone(a)(b)` in the data net we define the following STS:



- for each filter element `filt(a)(b)` in the data net we define the following STS:



- for each last element `last(a)(b)` in the data net we define the following STS:



The STS $\Sigma_{\mathcal{D}}$ modeling the data net $\mathcal{D}$ is the synchronized product of all the STSs corresponding to external connection nodes and to data-flow elements of $\mathcal{D}$. The synchronized product $\Sigma_1||\Sigma_2$ models the fact that the systems Σ_1 and Σ_2 evolve simultaneously on common actions and independently on actions belonging to a single system.

Generating the Composite Process

We are ready to show how we can integrate the proposed composition approach within the automated composition framework presented in Section 3.4.1. Given n component services $W_1, ..., W_n$ and a data net $\mathcal{D}$ modeling the data-flow composition requirements, we encode each component service W_i as a STS Σ_{W_i} and the data net $\mathcal{D}$ as a STS $\Sigma_{\mathcal{D}}$. The composition domain Σ for the automated composition problem is the synchronized product of all these STSs. Formally, $\Sigma = \Sigma_{\mathcal{D}} \parallel \Sigma_{W_1} \parallel .. \parallel \Sigma_{W_n}$. The planning goal ρ is the EAGLE formalization of the composition termination requirements, enriched with the requirements that all the data flow STS need to terminate in a final state.

Given the domain Σ and the planning goal ρ we can apply the approach presented in Section 3.4.1 to generate a controller Σ_c, which is such that $\Sigma_c \triangleright \Sigma \models \rho$. Once the state transition system Σ_c has been generated, it is translated into WS-BPEL to obtain the new process which implements the required composition. The translation is conceptually simple; intuitively, input actions in Σ_c model the receiving of a message from a component service, output actions in Σ_c model the sending of a message to a component service, internal actions model manipulation of data by means expressions and assignments.

Chapter 4

Adaptation

Adaptable pervasive systems change their behavior, reconfigure their structure and evolve over time reacting to changes in the operating conditions, so to always meet users' expectations. This is fundamental since those systems live in distributed and mobile devices, such as mobile phones, PDAs, laptops etc., thus their environment may change frequently. Also, user goals and needs may change dynamically, and systems should adapt their functionalities accordingly, without intervention from technicians. The flow modeling paradigm is often used either implicitly or explicitly in many real life situations. In ALLOW a novel usage of flows is being investigated [2]: the usage of flows as a new programming paradigm for human-oriented pervasive applications. More precisely, Adaptable Pervasive Flows (APFs) [10, 14] have been proposed as an extension of traditional workflow concepts [25] in order to make them more flexible with respect to their pervasive execution environment. APFs are dynamic workflows situated in the real world that modify their execution in order to adapt to changes in their environment. This requires on the one hand that a flow must be context-aware: during execution it must be possible to obtain information on the underlying environment (e.g. relevant information on world entities, status of other flows, human activities). On the other hand flow models must be flexible enough to allow an easy and continuous adaptation. APFs are based on WS-BPEL [1], a well-known language for specifying flows in a Web Service setting, and extend it in order to implement all the aspects related to pervasive applications.

In order to achieve the required degree of flexibility in ALLOW an approach called *"built-in"* has been provided [3, 18] and allows to adapt the application if the conditions change in some expected way. However, since the adaptation logic is hard-wired into the application, it is not possible to adapt to unforeseen changes in the operating conditions. Dynamic adaptation instead aims at adapting the system to unexpected changes [9].

In this Chapter we propose approaches to dynamic adaptation, based on the separation between the application and the adaptation specification. To be adaptable, an application should provide information to the environment on part of its structure and its behavior. The adaptation logic instead should be developed separately, for instance as a set of adaptation rules [15], by some *adaptation engineer*, and can be created and/or changed after the application has been deployed without modifying (and even without stopping) the running application. At runtime, the *adaptation manager* should check the environment conditions and the user needs, control whether some adaptation rule has to be applied to the application, and exploit the adaptation information provided by the application to reconfigure it in the desired way.

Here we provide theories and mechanisms for two types of dynamic adaptation called: *Vertical* and *Horizontal*. The aim of vertical adaptation is to refine abstract activities of a flow model to obtain a concrete flow that can be executed. Horizontal adaptation become necessary when there is a need to revise the flow model respect to context changes or if non-covered problems in the flow arises. In the next sections we discuss and illustrate our proposals to solve these two adaptation problems. We start presenting how to compose flow fragment models automatically according to goals, then we continue illustrating how to use the fragment-composition approach to provide vertical adaptation. We conclude this chapter showing how, using planning techniques, we can re-plan a flow execution considering the runtime context of the application. Each section below starts with the description of the problem that

we want to solve and proceeds providing both formal and technical solution. Each approach is validated using the Scenario described in Section 2.1 and presents some possible extensions.

4.1 Goal-based Fragment Composition

4.1.1 Problem Description

Process fragments [11] represent a tool for modeling incomplete and local process knowledge. The knowledge is incomplete since the modeler is allowed to leave gaps in the process specification. Further, it is local, since the availability and usability of the fragments is determined by context (e.g., location, time, situation, people). For example, the process fragment execution may be bound to a certain location.

Process fragment knowledge can be dynamically integrated. This requires enriching processes with goals which specify what is pursued by the process execution. It also requires enriching the fragments with information on how they contribute to the process outcome. The integration of process fragment knowledge can occur either at design time or at runtime. Being able to integrate knowledge at runtime makes process fragments a particularly suitable framework for pervasive applications, due to the incompleteness and locality on the fragment knowledge.

Flow fragment models are the result of applying the process fragment definition to the paradigm of adaptable pervasive flows. In this section, we provide a solution for composing flow fragment models automatically according to complex flow goals. We compute the solution by encoding fragment models, domain knowledge and goals into an AI planning problem. As a result, the fragment models are integrated into an overall flow model that guarantees to achieve the goals.

Our approach is based on previous results for service composition [22], and in particular on the work introduced in section 3.4, which addressed the problem of Web service composition with complex composition requirements. This approach needs however to be significantly extended in order to deal with the specific case of flow fragment composition. At the same time it refines and concretizes concepts already defined in the Deliverable D3.2 [5]. Fragment composition is conceptually different from Web service composition, since the components are not orchestrated, but integrated into one overall flow model.

The proposed approach is based on a three-layer representation, where the first layer captures the specific domain knowledge required for the composition task, the second layer describes the abstract flow in terms of the goals its should achieve, and the third layer is the concrete, context-dependent part of the flow definition, represented in terms of flow fragments.

For exemplification, we consider the car logistics scenario introduced in Section 2.1. The basic flow for handling a car depends on context information and may be distributed at different locations on the automobile terminal. Therefore, the complete flow model for treating a particular car is not known from the beginning. What is known is the goal of the flow: *"to deliver the car to the dealer or customer"*. The precise flow model that can achieve this goal is created at execution time, based on the available fragment models.

In the following, we briefly describe our three-layer representation (in Section 4.1.2), followed by a detailed description of each of the elements involved: entities, goals, and flow fragments. We then present a fragment composition technique based on planning (in Section 4.1.3) and discuss future work (in Section 4.1.5).

4.1.2 Application Representation

For modeling an application, we consider that there exist three layers, arising from two distinctions (Fig. 4.1). First, we distinguish between domain knowledge, or knowledge about the entities in the domain, and process knowledge, or knowledge about business logic. Second, we distinguish between knowledge that is concrete and context-dependent, and knowledge that is common, abstract, independent of context. The first layer is thus the domain knowledge, which is also stable and abstract. The second layer corresponds to the part of process knowledge which is stable and abstract, represented using goals.

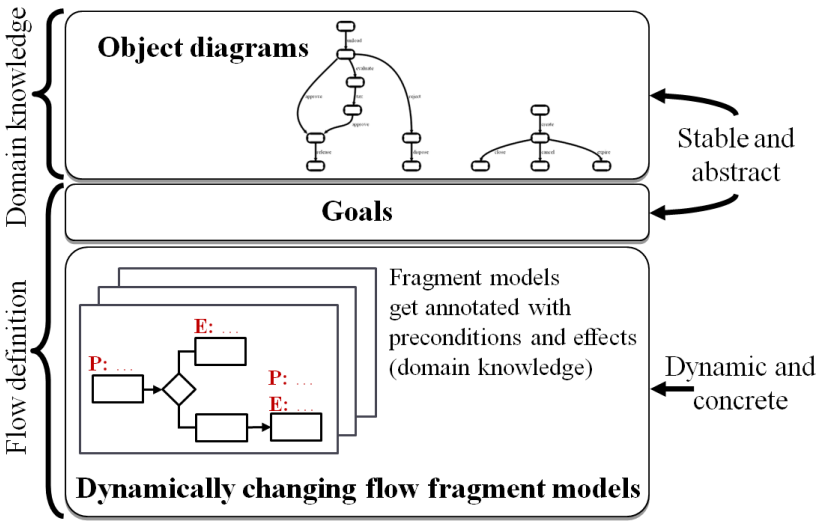


Figure 4.1: Application Representation.

The third corresponds to the part which is dynamic and concrete, represented using flow fragments. In the following, we give an overview of each of these layers.

The domain knowledge consists of the types of entities in the domain. The concept of entity plays an important role for adaptable pervasive flows. In fact, the main purpose of a flow is to model the behavior and goals of a specific entity, and move with it through different contexts. Entities are characterized by hierarchically structured types. An entity type includes a set of properties (for a car these can be type, position, color). It also includes a set of events that represent sources of change for the characteristics of the entity (e.g., if a car is moved its position will change, but potentially also its status, since the car may be damaged). Properties correspond to states that last for a period of time, while events are actions that are in effect only at a certain point and make the entity evolve from one state to another.

We use this domain knowledge to define our flow goals. A goal can be used to model a flow or a part of the flow for which the exact content is not known at design-time, and may potentially not be known until the time when the flow instance is executed. At runtime, the goal will be substituted with a concrete realization, which is computed by composing available flow fragment models. Using goals, we can specify the target state for our flow, as well as coordination requirements on the evolution of a set of entities. While a target state corresponds to a situation we want to achieve at the end of the flow execution, a coordination requirement is a property that we want to ensure during the entire execution. Such a requirement can be used, for example, to model the consistent evolution of a set of entities.

To formally model entities and goals, we borrow the formalization presented in Section 3.4. In particular, we use a simplified version of the object diagrams, presented in the Definition 3.4.7, to represent entities. Further, we express goals using the same language introduced, in the definition 3.4.9, for representing composition requirements. Despite the different application domain, this formalization perfectly captures the core characteristics of our contextual entities and flow goals.

We use the domain knowledge also for annotating flow fragment models. Flow fragment models consist of activities and control elements. Each fragment model has one or more associated entities, and its execution is directly reflected in the evolution of the entities. In order to relate fragment models to entities we use preconditions and effects. These can be defined on any activity in the fragment. Preconditions are properties that have to hold in order for the activity to be applied in the composition. Effects are properties that are made true by applying the corresponding activity.

Using this representation, both goals and object diagrams are specified independently from fragment models. Therefore, using the same object diagrams and goals, we can achieve different fragment compositions for different contexts. For this purpose it is sufficient for the fragment models to be linked (through preconditions and effects) to the object diagrams.

Object Diagrams

Fig. 4.2 displays some object diagrams of our scenario. For representing a car we make use of two object diagrams: the car status (*CS*) and the car position (*CP*). We consider that there exists a technical center nearby the terminal, and that the treatment queue (*TQ*) corresponds to a parking area adjacent to this technical center. Finally, the terminal location (*TL*) corresponds to a single car parking place on the terminal.

The object diagrams move from one configuration to another on certain events. For example, the car status diagram may be in the configuration *NOK-LIGHT*, which signals that the car has a light damage. If the diagram received the event *repair*, it will move to the configuration *OK*.

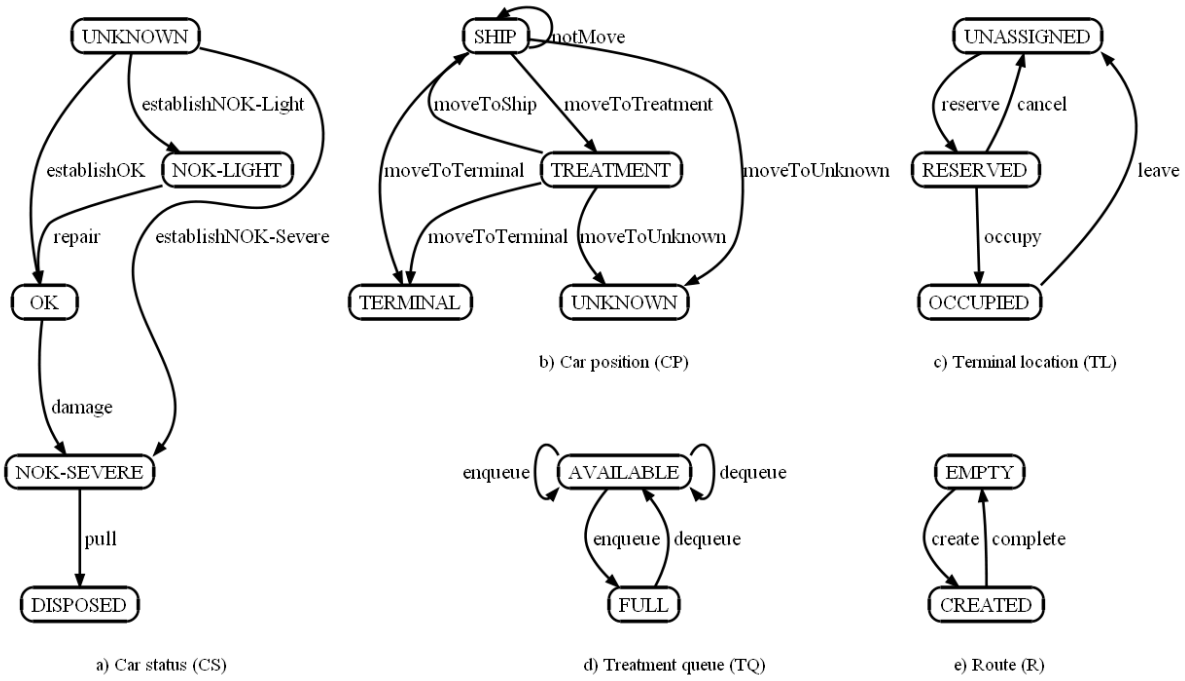


Figure 4.2: Object diagrams.

Goals

Consider the object diagrams from Fig. 4.2. To demonstrate fragment composition, we consider a simplified goal, for the car to reach safely the terminal location. This means that the car status *CS* should be in the configuration *OK*, while the car position *CP* is in the configuration *TERMINAL*. If this is not possible, we at least want to have the car disposed of, meaning that the car status *CS* should be in configuration *DISPOSED*. This goal can be written as follows:

$$\top \Rightarrow ok^s(CS) \wedge terminal^s(CP) \succ disposed^s(CS) \quad (G_1)$$

Flow Fragments

Flow fragments are the result of applying the process fragment definition from [11] to the paradigm of adaptable pervasive flows. Flow fragments models provide a means to encode incomplete, local flow

knowledge. They can be integrated into complete flow models by means of composition, and as such can be seen as the building blocks of flow models.

Process fragments allow three options for modelling incomplete process knowledge. First, in a process fragment it is possible for control connectors to have either no source or no target activity. Then, the modeler has the freedom to not model control flow relations at all. Finally, process fragments may contain an element called region, which stands for business logic that is not defined. Regions can be connected to other activities using control connectors. Our flow fragments incorporate the first two options. In our future work, we plan to add also the third option, the region.

In order to compose flow fragment models automatically according to goals, we extend the fragment definition with relations to objects: preconditions and effects. These can be defined on any activity in the fragment model, as in Fig. 4.3.

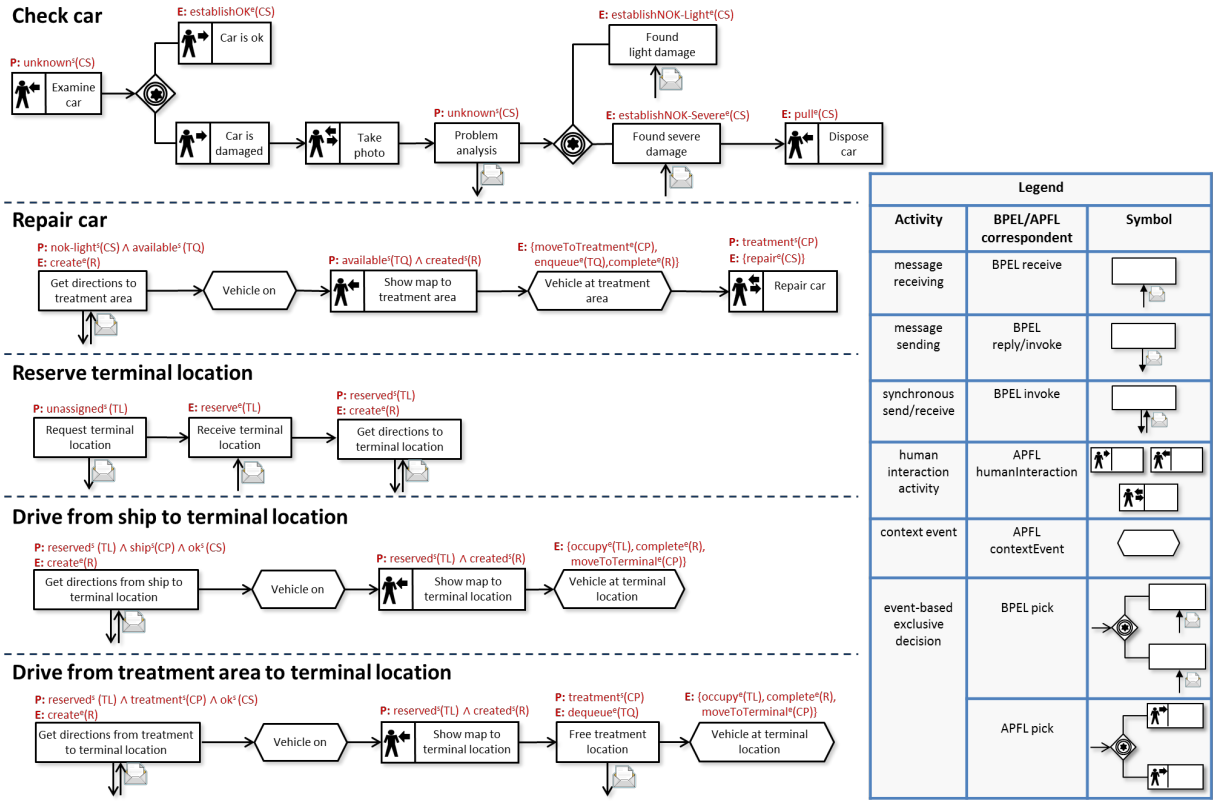


Figure 4.3: Annotated fragments available for composition.

Preconditions (denoted with *P:*) refer to configurations in the object diagrams. They are propositional formulas over the set of propositions $\{s_j{}^{\mathsf{s}}(o_i)\}$, where o_i are object diagrams and s_j configurations. By adding a precondition to an activity, we require that the objects manipulated by the activity are in particular configurations. If the precondition does not hold, the activity can not be applied in the composition. For example, in the first fragment from Fig. 4.3, the activity *Examine car* requires the object diagram *CS* to be in the configuration *UNKNOWN*.

Effects (denoted with *E:*) refer to events in the object diagrams. They are sets of propositions from $\{e_j{}^{\mathsf{e}}(o_i)\}$, where o_i are object diagrams and e_j events. By annotating an activity with effects, we encode the fact that the associated objects may move to different configurations as a result of executing the activity. Again for the first fragment, activity *Car is ok* triggers the event *establishOK* on the object diagram *CS*. This can happen only if in the current configuration of the object there exists a transition on the event *establishOK*, i.e., if *CS* is in configuration *UNKNOWN*. Like in our example, such conditions can be left implicit. However, it is also possible to make them explicit, using the preconditions.

Overlapping activities

One of the key issues about fragments is that they can include overlapping activities. The reason is that fragment modelers have only a local view of the entire flow and may therefore model the same information. Informally, two activities are overlapping if and only if there exists at least one object diagram for which they have the same effects, and their preconditions and effects are consistent. Two preconditions are considered to be consistent if they do not require any object to be in different configurations. In this setting, by preconditions we refer to the conjunction of explicit conditions (specified by the fragment modeler), and implicit conditions (required by the effects of the activity). Further, two effects are considered to be consistent if they do not trigger different transitions in the same object diagram.

For defining the consistency of preconditions, we first introduce a helper formula Xor. For an object diagram $o = \langle L, L_0, \mathcal{E}, T\rangle$, $Xor(o)$ states that the object can only be in only one configuration at a time:

$$Xor(o) = (\bigvee_{s_i \in L} s_i^s(o)) \wedge \bigwedge_{s_i \in L} (s_i^s(o) \rightarrow \bigwedge_{s_j \in L, s_j \neq s_i} \neg s_j^s(o))$$

Let a_1 be an activity with preconditions P_1 and effects E_1 defined on a set of object diagrams $o_1, \ldots, o_k, o_{k+1} \ldots, o_n$. Let a_2 be a second activity with preconditions P_2 and effects E_2 defined on $o_1, \ldots, o_k, o'_{k+1}, \ldots, o'_m$. We say that:

- P_1 and P_2 are consistent iff $P_1 \wedge P_2 \wedge \bigwedge_{1 \leq i \leq k} Xor(o_i)$ is satisfiable;
- E_1 and E_2 are consistent iff for all $o_i \in \{o_1, \ldots, o_k\}$, $o_i = \langle L, L_0, \mathcal{E}, T\rangle$, $E(o_i) = \{e^e(o_i) | \forall e \in \mathcal{E}\}$, we have $E_1 \cap E(o_i) = E_2 \cap E(o_i)$.

Further, a_1 and a_2 are overlapping iff:

- $k \geq 1$,
- P_1 and P_2 are consistent,
- E_1 and E_2 are consistent,
- $\exists o_i \in \{o_1, \ldots, o_k\}$, $o_i = \langle L, L_0, \mathcal{E}, T\rangle$, $E(o_i) = \{e^e(o_i) | \forall e \in \mathcal{E}\}$, such that $E_1 \cap E(o_i) \neq \emptyset$.

Consider for example the activity called *Get directions to terminal location* in the third fragment model (Fig. 4.3). This activity overlaps with the activity *Get directions from ship to terminal location* in the forth fragment, as well as with the activity *Get directions from treatment to terminal location* in the last fragment. *Get directions to terminal location* does not have the same name, or the same preconditions as the latter two. However, the consistency requirements are satisfied. The activities have in common two object diagrams: the terminal location *TL* and the route *R*. On these diagrams, they have the same conditions (*TL* must be in state *RESERVED* and *R* in state *EMPTY*) and trigger the same event (*create*). Therefore the consistency requirements are satisfied. Notice however that *Get directions from ship to terminal location* does not overlap with *Get directions from treatment to terminal location*, since they have in common also the diagram car position *CP*, which is assumed to be in different configurations.

4.1.3 Solution

The goal-based composition problem can be stated as follows. Given a set of flow fragment models, a set of object diagrams, and a set of composition goals, the problem is to integrate the flow fragment models into a flow model that is guaranteed to achieve the goals.

Our solution is presented schematically in Figure 4.4. First, we encode fragment models, object diagrams and goals into a planning domain Σ (steps 1-4). Further, we create the planning goal ρ based on the goals given as input to fragment composition (step 5). Finally, on the domain Σ and the goal ρ we apply the approach presented in Section 3.4, which generates a separate controller Σ_c for controlling

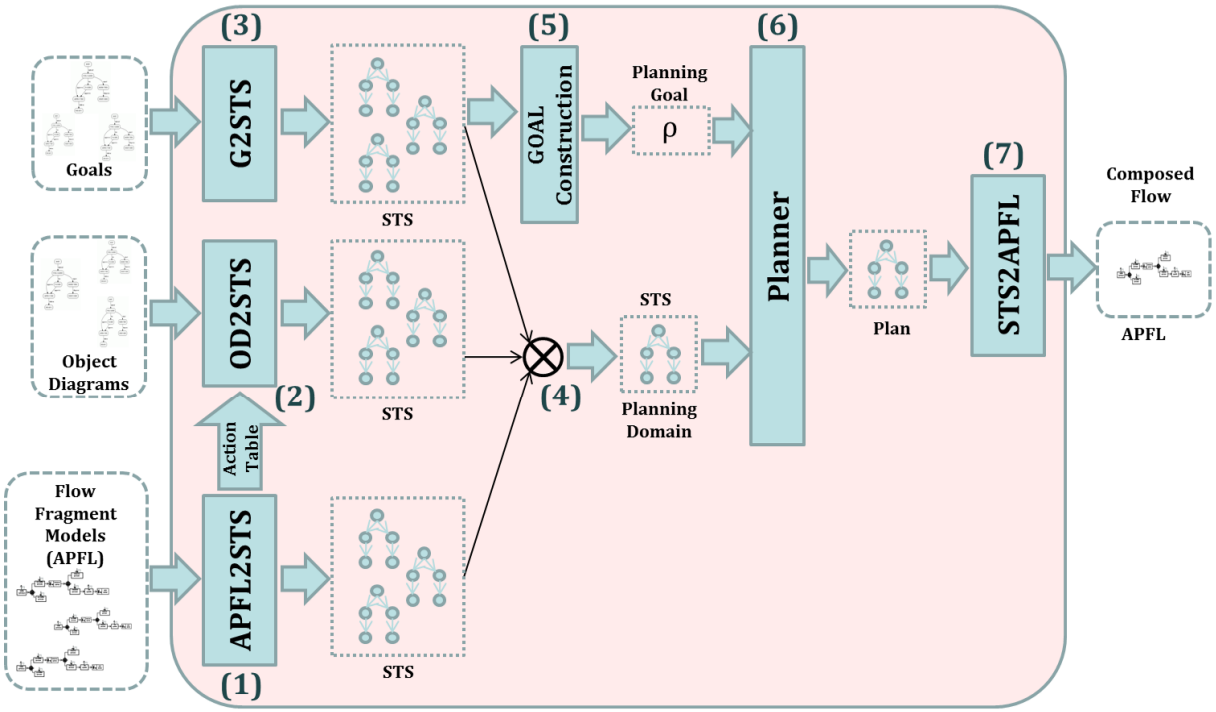


Figure 4.4: Solution overview.

the planning domain in such a way as to satisfy the goal ρ (step 6). In difference to Section 3.4, here we are not interested in retrieving the controller Σ_c, but in analyzing the controlled domain. If Σ_c exists, then the controlled domain can be used to generate an arrangement of the fragments which achieves the composition goals (step 7).

The planning domain Σ is defined as a state transition system (STS). An STS contains a set of states, some of which are marked as initial and/or final. Each state is labeled with sets of properties that hold in that state. The STS can evolve to new states as a result of performing actions. The actions are of two types: *input* (controllable) or *output* (not controllable).

Definition 4.1.1 (STS) *Let $\mathcal{L}$ be a set of proposition symbols and $Bool(\mathcal{L})$ the set of boolean expressions over $\mathcal{L}$. A state transition system (STS) is a tuple $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$, where*

- *$\mathcal{S}$ is the set of states and $\mathcal{S}^0 \subseteq \mathcal{S}$ are the initial states,*
- *$\mathcal{I}$ and $\mathcal{O}$ are the input and output actions respectively,*
- *$\mathcal{R} \subseteq \mathcal{S} \times Bool(\mathcal{L}) \times (\mathcal{I} \cup \mathcal{O}) \times \mathcal{S}$ is the transition relation,*
- *$\mathcal{S}^F \subseteq \mathcal{S}$ is the set of accepting states,*
- *$\mathcal{F} : \mathcal{S} \to 2^{\mathcal{L}}$ is the labeling function.*

In our planning domain Σ, the set $\mathcal{L}$ will consist of all propositions of the form ${s_j}^{\mathsf{s}}(o_i)$, encoding the fact that the object diagram o_i is in configuration s_j. The labeling function $\mathcal{F}$ determines whether a boolean expression $b \in Bool(\mathcal{L})$ holds in a particular state s. We write $s, \mathcal{F} \models b$ to denote that b is satisfied at state s given $\mathcal{F}$. Satisfiability of a formula is determined according to the following standard inductive rules:

- $s, \mathcal{F} \models \top$;

- $s, \mathcal{F} \models p$, iff $p \in \mathcal{F}(s)$;
- $s, \mathcal{F} \models \neg b$, iff $s, \mathcal{F} \not\models b$;
- $s, \mathcal{F} \models b_1 \vee b_2$, iff $s, \mathcal{F} \models b_1$ or $s, \mathcal{F} \models b_2$.

The transitions in the STS are guarded: a transition $\langle s, b, a, s' \rangle$ is possible in state s only if the guard expression b is satisfied in that state, $s, \mathcal{F} \models b$.

To create our planning domain, we first need to transform fragment models, object diagrams and goals into STSs. The planning domain will then be the parallel product of these STSs, capturing their simultaneous evolution. Creating the planning domain in this manner was inspired by the work in [6], approach presented in Section 3.4.2. In difference to [6], and due to the fact that we enforce preconditions on the activities, all the transitions in our STSs will be guarded. Also, in order to allow fragment activities to overlap, we will introduce a new definition for the parallel product of the STSs.

After building the planning domain and the planning goal, we determine if a controller exists such that the controlled domain satisfies the goal. Here, we use the following notion of a controlled system.

Definition 4.1.2 (Controlled System)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$ and $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c, \mathcal{S}_c^F, \mathcal{F}_c \rangle$ be two STSs. STS $\Sigma_c \triangleright \Sigma$, describing the behaviors of system Σ when controlled by Σ_c, is defined as follows:

$$\Sigma_c \triangleright \Sigma = \langle \mathcal{S}_c \times \mathcal{S}, \mathcal{S}_c^0 \times \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c \triangleright \mathcal{R}, \mathcal{S}_c^F \times \mathcal{S}^F, \mathcal{F}_c \cup \mathcal{F} \rangle$$

where: $\langle (s_c, s), (b_c \wedge b), a, (s'_c, s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$, if $\langle s_c, b_c, a, s'_c \rangle \in \mathcal{R}_c$ and $\langle s, b, a, s' \rangle \in \mathcal{R}$

In the following, we present the transformation of fragment models, object diagrams and goals as STSs, as well as the construction of the planning domain and the planning goal. Finally, we give some details on our implementation of the Car Logistics scenario.

Transforming the fragment models We translate the flow fragment models into STSs of the form $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$. Using this transformation, we lose some information, namely that encoded in the effects of activities. We capture this information using a second data structure, which will be used later on, when transforming the object diagrams and the goals.

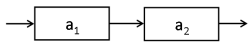
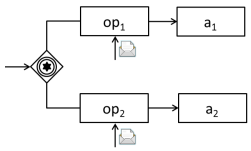
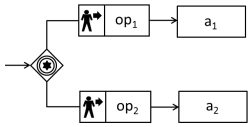
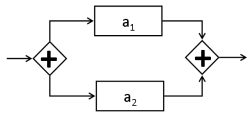
In order to encode an APFL fragment model as an STS, we recursively translate the flow activities. Activity preconditions will be copied as transition guards in the STS. In this setting, as for overlapping activities, with activity preconditions we refer to the conjunction of explicit and implicit conditions. In the following, we will give first the intuition and then the formal description for the translation of APFL basic and structured activities.

We encode flow activities using input and output actions. Input actions correspond to activities that can be controlled by the fragment composition, such as the *reply*, *invoke*, *assign*, *contextEvent*, *humanInteraction* activities. Output actions correspond to activities that can not be controlled by the fragment composition, such as the *onMessage* and *onHumanInteraction* within a *pick*, respectively an *apfPick* activity. The idea here is that once a *pick* activity is selected for composition, all its *onMessage* branches must be considered. In this sense, it is not under the control of the fragment composition whether to include the activity *onMessage* or not. [1]

Table 4.1 contains the APFL activities and their translation to STS. To differentiate between an input and an output action, we will prepend the names with a "?", respectively a "!". We denote the start state of an activity with s_b, and the end states with s_e, respectively s'_e. With $s_b \xrightarrow{a} s_e$ we denote the recursive translation of activity a. Finally, we denote the boolean formula corresponding to the precondition of a basic activity with b.

[1] Note that this encoding as input/output actions is different from the encoding of Web services in [**?**]. The reason is that in Web service composition the purpose is not to integrate the Web services, but to orchestrate them using an external controller. Therefore, actions are considered to be input (output) if they are controllable (non-controllable) by the external orchestrator.

Table 4.1: Translating APFL to STS.

APFL Activity		STS Transitions
receive, reply, invoke, assign, contextEvent, humanInteraction		$(s_b, b, ?op, s_e)$
sequence activity a_1 activity a_2		$s_b \xrightarrow{a_1} s', s' \xrightarrow{a_2} s_e$
pick onMessage operation="op_1" activity a_1 onMessage operation="op_2" activity a_2		$(s_b, \top, ?pick, s_0)$ $(s_b, b_1, !op_1, s_1)$ $(s_b, b_2, !op_2, s_2)$ $s_1 \xrightarrow{a_1} s_e, s_2 \xrightarrow{a_2} s'_e$
apfPick onHumanInteraction op_1 activity a_1 onHumanInteraction op_2 activity a_2		
flow activity a_1 activity a_2		$(s_b, \top, ?flow, s_0)$ $(s_0, \top, !order_1, s_1)$ $(s_0, \top, !order_2, s_2)$ $s_1 \xrightarrow{a_1} s'_1, s'_1 \xrightarrow{a_2} s_e$ $s_2 \xrightarrow{a_2} s'_2, s'_2 \xrightarrow{a_1} s_e$

Note that we generate new action names for each basic activity. This way, each action name corresponds to one appearance of an activity in a fragment. This assumption simplifies the construction of the parallel product, as well as the process of identifying the fragments in the final composition.

We capture the effects of basic activities using a second data structure, which we call action table. We add one entry in the action table for each basic activity with non-empty effects. An entry has the form $\langle b, op, \varepsilon \rangle$, where b and ε are the precondition, respectively the effects of the activity, and op is the action corresponding to the activity.

After translating all fragments, we capture the information regarding overlapping activities with a binary relation $Overlap$ defined on the set of actions. For every pair $\langle a_1, a_2 \rangle \in Overlap$, a_1 and a_2 are actions corresponding to two overlapping activities.

By default, we allow fragments to be used also partially, and therefore $\mathcal{S}^F$ contains all the states of the fragment. However, when the fragment contains the structured activity *flow*, we remove from $\mathcal{S}^F$ all the states included in the translation of the activity, with the exception of the initial and the final state.

Consider for example the fragment *Check car* in Fig. 4.3. The translation of the fragment is an STS $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F} \rangle$, where:

- $\mathcal{S} = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}\}$
- $\mathcal{S}^0 = \{s_0\}$
- $\mathcal{I} = \{?F_1.Examine\_car, ?F_1.Pick_1, ?F_1.Take\_photo, ?F_1.Problem\_analysis, ?F_1.Pick_2, ?F_1.Dispose_car\}$
- $\mathcal{O} = \{!F_1.Car\_is\_ok, !F_1.Car\_is\_damaged, !F_1.Found\_light\_damage, !F_1.Found\_severe\_damage\}$

- $\mathcal{R}$ contains the following transitions:
  $(s_0, unknown^s(CS), ?F_1.Examine\_car, s_1)$,
  $(s_1, unknown^s(CS), !F_1.Car\_is\_ok, s_2)$,
  $(s_1, \top, !F_1.Car\_is\_damaged, s_3)$,
  $(s_3, \top, ?F_1.Take\_photo, s_4)$,
  $(s_4, unknown^s(CS), ?F_1.Problem\_analysis, s_5)$,
  $(s_5, unknown^s(CS), !F_1.Found\_light\_damage, s_6)$,
  $(s_5, unknown^s(CS), !F_1.Found\_severe\_damage, s_7)$,
  $(s_7, nok\text{-}severe^s(CS), ?F_1.Dispose\_car, s_8)$
- $\mathcal{S}^F = \{s_0, s_1, s_3, s_4, s_5, s_6, s_8\}$
- $\mathcal{F} = \emptyset$

The action table entries for the same fragment are:

- $\langle unknown^s(CS), !F_1.Car\_is\_ok, \{establishOK^e(CS)\}\rangle$
- $\langle unknown^s(CS), !F_1.Found\_light\_damage, \{establishNOK\text{-}Light^e(CS)\}\rangle$
- $\langle unknown^s(CS), !F_1.Found\_severe\_damage, \{establishNOK\text{-}Severe^e(CS)\}\rangle$
- $\langle nok\text{-}severe^s(CS), ?F_1.Dispose\_car, \{pull^e(CS)\}\rangle$

Transforming the object diagrams Given an object diagram $o = \langle L, L_0, \mathcal{E}, T\rangle$, we define an STS $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F}\rangle$, where $\mathcal{S} = L$, $\mathcal{S}^0 = L_0$, and all object configurations are accepting (i.e., $\mathcal{S}^F = \mathcal{S}$). Further, we label states with the corresponding propositions (i.e., $\forall s \in \mathcal{S} : \mathcal{F}(s) = \{s^{\mathsf{s}}(o)\}$).

To build the transitions in our new STS, we will use the action table created in Section 4.1.3. For every transition $(l, e, l') \in T$ we consider all the action entries $\langle b, a, \varepsilon\rangle$ in the action table, such that $e \in \varepsilon$. For each such action entry $\langle b, a, \varepsilon\rangle$, we add to $\mathcal{R}$ the transition (l, b, a, l').

For example, in the STS for *CS*, we use the original transition $(NOK\text{-}LIGHT, repair, OK)$ and the table entry $\langle nok\text{-}light^s(CS), ?F_3.Repair\_car, \{repair^e(CS)\}\rangle$ to create the new transition $(NOK\text{-}LIGHT, nok\text{-}light^s(CS), ?F_3.Repair\_car, OK)$.

Transforming the goals For each goal, we construct the STSs that correspond to the satisfiability of the goal. For every formula φ, we define a single output action e_φ which gets triggered when the formula is satisfied. We use these completion actions for composing the formulas. The preconditions on the activities will be carried over as guards also in the goal STSs. We shortly describe the STSs for each of the building blocks of goal formulas:

- the STS for $\top$ has one transition on the completion action;
- the STS for $s^{\mathsf{s}}(o)$ has one transition guarded with the corresponding proposition;
- the STS for $e^{\mathsf{e}}(o)$ waits for any activity that contains the event in its effects. For any action entry $\langle b, a, \varepsilon\rangle$ such that $e^{\mathsf{e}}(o) \in \varepsilon$, we add to our STS a transition on a guarded by b.
- the STS for $\varphi_1 \vee \varphi_2$ waits for any of e_{φ_1} and e_{φ_2}, while the STS for $\varphi_1 \wedge \varphi_2$ waits for both.

The STS for a goal formula $\varphi \Longrightarrow (\varphi_1 \succ \ldots \succ \varphi_n)$ is as follows. If the premise takes place (e_φ is reported), it moves to a non-accepting state and waits for any of $e_{\varphi_1}, \ldots, e_{\varphi_n}$ to be reported. The goal with preferences has the form: $\rho_c = (s_0, s_1, \ldots, s_n)$, where s_0 is the initial state of the STS, and $s_1, \ldots, s_n$ are the states reached with the transitions corresponding to $e_{\varphi_1}, \ldots, e_{\varphi_n}$.

Generating the composed flow We build the planning domain by taking the parallel product of all the STSs of fragment models, object diagrams and goals. The key idea here is to enforce actions to be applied together if they overlap and both of their preconditions hold.

We use a simplifying assumption when constructing the parallel product. The assumption is that a particular action always appears with the same guard. By construction, each action corresponds to exactly one appearance of a basic activity in a fragment model. The guard corresponds to the precondition of this activity. The relation between the two is maintained when transforming the object diagrams and the goals, since we always add the action together with the guard.

Slightly abusing the notation, we write $a\|a'$ to denote a new action which corresponds to actions a and a' being performed in parallel. We call $a\|a'$ a parallel action. Note that a and a' can themselves be parallel actions.

The overlap relation can now be extended to cover the parallel actions. Given two actions $a \equiv a_1\| \ldots \|a_n$ and $a' \equiv a'_1\| \ldots \|a'_m$, if for all $1 \leq i \leq n$ and $1 \leq j \leq m$, we have $\langle a_i, a_j\rangle \in Overlap$, then also $\langle a, a'\rangle \in Overlap$.

Definition 4.1.3 (Parallel Product)
Let $\Sigma_1 = \langle \mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{R}_1, \mathcal{S}_1^F, \mathcal{F}_1\rangle$ and $\Sigma_2 = \langle \mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{R}_2, \mathcal{S}_2^F, \mathcal{F}_2\rangle$ be two STSs. The parallel product $\Sigma_1\|\Sigma_2$ is defined as
$\langle \mathcal{S}_1 \times \mathcal{S}_2, \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{R}_1\|\mathcal{R}_2, \mathcal{S}_1^F \times \mathcal{S}_2^F, \mathcal{F}_1\|\mathcal{F}_2\rangle$
where $(\mathcal{F}_1\|\mathcal{F}_2)(s_1, s_2) = \mathcal{F}_1(s_1) \cup \mathcal{F}_2(s_2)$ and
for every state $s_1 \in \mathcal{S}_1$ and $s_2 \in \mathcal{S}_2$:

- *if $\langle s_1, b, a, s'_1\rangle \in \mathcal{R}_1$ and $\langle s_2, b, a, s'_2\rangle \in \mathcal{R}_2$, then*
  $\langle (s1, s2), b, a, (s'_1, s'_2)\rangle \in (\mathcal{R}_1\|\mathcal{R}_2)$
- *if $\langle s_1, b_1, a_1, s'_1\rangle \in \mathcal{R}_1$ and $\forall\langle s_2, b_2, a_2, s'_2\rangle \in \mathcal{R}_2$, $a_1 \neq a_2$ and $\langle a_1, a_2\rangle \notin Overlap$, then*
  $\langle (s1, s2), b_1, a_1, (s'_1, s_2)\rangle \in (\mathcal{R}_1\|\mathcal{R}_2)$
- *if $\langle s_2, b_2, a_2, s'_2\rangle \in \mathcal{R}_2$ and $\forall\langle s_1, b_1, a_1, s'_1\rangle \in \mathcal{R}_1$, $a_1 \neq a_2$ and $\langle a_1, a_2\rangle \notin Overlap$, then*
  $\langle (s1, s2), b_2, a_2, (s_1, s'_2)\rangle \in (\mathcal{R}_1\|\mathcal{R}_2)$
- *otherwise, if $\langle s_1, b_1, a_1, s'_1\rangle \in \mathcal{R}_1$, $\langle s_2, b_2, a_2, s'_2\rangle \in \mathcal{R}_2$ and $\langle a_1, a_2\rangle \in Overlap$, $(\mathcal{R}_1\|\mathcal{R}_2)$ contains:*
  - $\langle (s1, s2), b_1 \wedge b_2, a_1\|a_2, (s'_1, s'_2)\rangle$
  - $\langle (s1, s2), b_1 \wedge \neg b_2, a_1, (s'_1, s_2)\rangle$
  - $\langle (s1, s2), \neg b_1 \wedge b_2, a_2, (s_1, s'_2)\rangle$

We construct our planning domain Σ as the parallel product of fragment STSs $\Sigma_{F_1}, \ldots, \Sigma_{F_n}$, object STSs $\Sigma_{O_1}, \ldots, \Sigma_{O_m}$, and goal STSs $\Sigma_{C_1}, \ldots, \Sigma_{C_k}$:

$$\Sigma = \Sigma_{F_1}\| \ldots \|\Sigma_{F_n} \parallel \Sigma_{O_1}\| \ldots \|\Sigma_{O_m} \parallel \Sigma_{C_1}\| \ldots \|\Sigma_{C_k}$$

After construction, we can simplify Σ by removing all the transitions $\langle s, b, a, s'\rangle$ for which $s, \mathcal{F} \not\models b$. On the resulting STS, we then remove the guards as well as the labeling function. The simplified planning domain, which we denote with Σ_s is a tuple $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}', \mathcal{S}^F\rangle$, where $\mathcal{R}' \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{S}$ is the updated transition relation.

We then construct the planning goal ρ by combining the composition goals ρ_c:

$$\rho = \bigwedge_c \rho_c$$

Given the domain Σ_s and the planning goal ρ, we apply the technique presented in [22], which generates a controller Σ_c such that the controlled domain $\Sigma_c \triangleright \Sigma_s$ satisfies ρ. If such a controller exists, then $\Sigma_c \triangleright \Sigma_s$ corresponds to an arrangement of the fragments which is guaranteed to achieve the goals.

4.1.4 Scenario Implementation

To evaluate the feasibility of our approach, we have implemented our reference scenario. For this purpose, we have manually encoded the planning domain Σ and goal ρ, constructed according to the specifications presented in this section. For the encoding, we have used a customized SMV language [2], extended to allow the specification of goals with preferences.

Using this approach, we can generate the STS of the controlled domain $\Sigma_c \triangleright \Sigma_s$, which corresponds to our composed flow model. For this particular scenario the resulting STS has 29 states, the synthesizer using the STS of the fragment *Reserve terminal location* two times in the composition, and parallelizing the overlapping actions

$F_3.Get\_directions\_to\_terminal\_location$ and
$F_4.Get\_directions\_from\_ship\_to\_terminal\_location$,

respectively

$F_3.Get\_directions\_to\_terminal\_location$ and
$F_5.Get\_directions\_from\_treatment\_to\_terminal\_location$.

Fig. 4.5 shows the translation of the composed flow back to APFL. This translation is conceptually simple, and is performed based on action names. From the construction of STSs, each action name is unique and corresponds to one appearance of an activity in a fragment model. Each action can therefore be mapped back to its corresponding activity. The only exception occurs in the case of parallel actions, where we need to introduce new activities, which result from merging the original overlapping activities.

4.1.5 Future Work

We presented an approach for composing flow fragment models according to goals. We started by viewing the available knowledge from a three-layer perspective. The first layer is the domain knowledge, represented using object diagrams. The second layer is the stable and abstract part of the flow definition, represented using goals, while the third layer is the context-dependent, concrete part, represented using flow fragments. We showed how to relate goals and fragment models to the domain knowledge. Finally, we presented a mechanism for composing fragment by encoding object diagrams, goals and fragments into an AI planning problem.

In our future work, we intend to address several open issues. First, we plan to include in our flow fragments the remaining modeling element introduced with process fragments: the region. A second issue is to extend the composition mechanism to take into account also the context information (context events, context constraints). As a last issue, until now we have been dealing only with control-flow requirements. We therefore plan to add also data to our flow fragment models.

4.2 Vertical Adaptation by on-demand flow creation

4.2.1 Problem Description

The aim of vertical adaptation is to refine the abstract activities of a flow model to obtain a concrete flow that can be executed [4]. This refinement/realization can be achieved in different ways. It can be achieved by binding the abstract activity to a concrete activity (web service invocation, human interaction). It can also be achieved by substituting the abstract activity with a concrete flow that can be either predefined or constructed at run-time by composing concrete activities or fragments.

A characteristic of vertical adaptation is that, although new activities are introduced in the flow, the structure of the original flow model remains unchanged and only the flow composition changes. In

[2] http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf

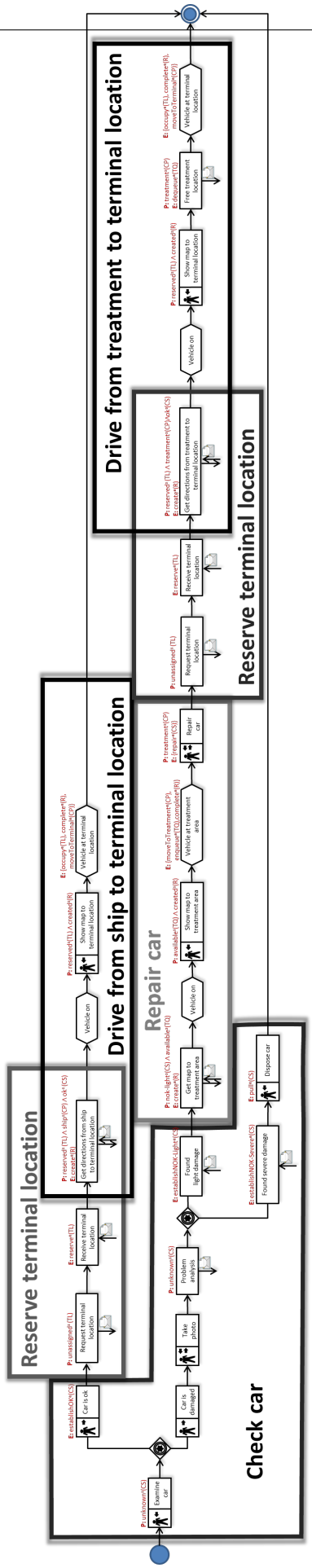


Figure 4.5: Goal-based fragment composition result.

contrast, when the structure of the flow model is affected, we say that the adaptation strategy is horizontal. When vertical adaptation is performed at the instance-level, the refinement of the abstract activity occurs only for one flow instance, while the instance is running. When performed at the level of the flow model, the refinement will be predefined. If only the control flow is considered, the problem of vertical adaptation at the instance-level and at the flow model level is largely the same. The main difference is that at the level of the flow model, the lack of execution information may introduce more uncertainty. This in turn may reduce the number of solutions. However, the mechanism used by vertical adaptation at both levels is the same. Therefore, in the following we will concentrate on vertical adaptation at the instance-level.

In order to solve the problem of vertical adaptation at instance-level, we make the following assumptions. Similarly to the fragment models in goal-based fragment composition, we consider that the flow instance is related to a set of object diagrams. In particular, the activities in the flow instance are annotated with preconditions and effects. Further, we consider that the execution of the flow instance triggers transitions on the associated object diagrams.

Figure 4.6 presents schematically the inputs and outputs of the vertical adaptation problem. The inputs are: the object diagrams associated to the flow instance together with information about their current states (1), the goal of the abstract activity currently being refined (2), and a set of annotated flow fragment models (3).

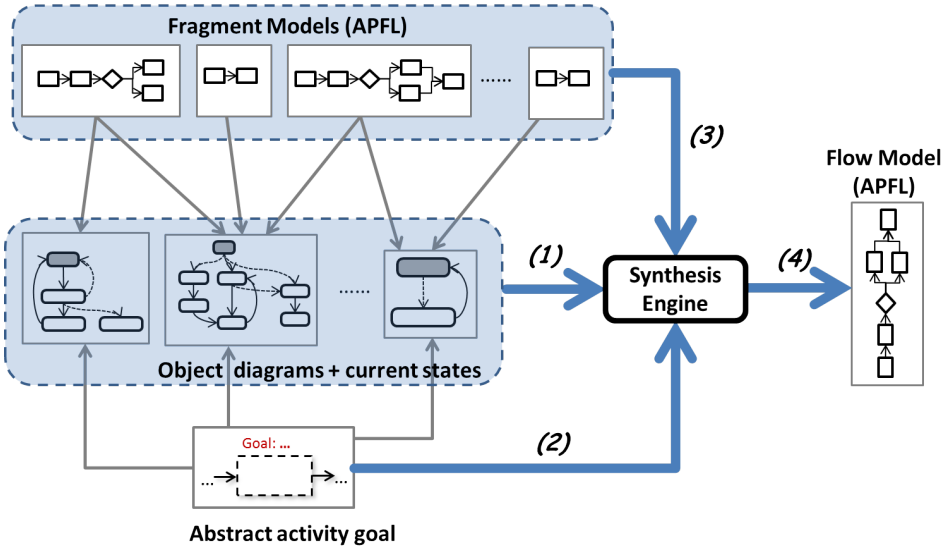


Figure 4.6: Vertical adaptation problem.

The mechanism of vertical adaptation consists of two basic steps. In the first step, we generate the refinement as a new flow model (4). For this purpose, we identify the fragment or composition of fragments that fulfills the goal associated to the abstract activity. In the second step, we replace the abstract activity with the newly generated flow model.

4.2.2 Solution

The problem of vertical adaptation when considering only the control flow can easily be restated as a problem of fragment composition. First, we create a set of new object diagrams corresponding to the object diagrams associated to the flow instance. Each new diagram will have as initial state the current state. Note that there can also be multiple possible current states, due to the fact that we allow nondeterministic events in our object diagrams. We can then restate the goal and revise all fragment models using the new object diagrams. Then, any solution to the fragment composition problem considering the new object diagrams, new goals, and new fragment models corresponds to a refinement for the abstract activity.

The inputs to the problem of vertical adaptation are:

- a set of object diagrams $\{o_1, \ldots, o_n\}, o_i = \langle L, L_0, \mathcal{E}, T\rangle$;
- for each object diagram $o \in \{o_1, \ldots, o_n\}$, the set of current configurations L_c;
- an abstract activity defined in terms of the goal G that it should achieve;
- a set of annotated flow fragment models FM available for selection/composition.

The problem can therefore be rewritten as a fragment composition problem as follows. For each object diagram $o \in \{o_1, \ldots, o_n\}$, where $o = \langle L, L_0, \mathcal{E}, T\rangle$ and o has as current configurations the set L_c, we create a new object $o' = \langle L, L_c, \mathcal{E}, T\rangle$.

We then construct the new goal formula G' by replacing in G all occurrences of the diagrams $\{o_1, \ldots, o_n\}$ with the corresponding new diagrams $\{o'_1, \ldots, o'_n\}$. Similarly, we construct a new set of flow fragment models FM' annotated with preconditions and effects that correspond to the new diagrams.

The new object diagrams $\{o'_1, \ldots, o'_n\}$, the new goal G', and the new set of flow fragment models FM' constitute the inputs to the goal-based fragment composition problem.

The output of composition will be a new flow model F that is guaranteed to satisfy the goal G'. F is annotated with preconditions and effects that correspond to $\{o'_1, \ldots, o'_n\}$. In order to obtain the refinement for our original goal G, we must simply revert the preconditions and effects of F to point to our original object diagrams $\{o_1, \ldots, o_n\}$.

Finally, the vertical adaptation mechanism will replace the abstract activity with the newly generated flow model.

4.2.3 Scenario Implementation

To demonstrate vertical adaptation, we consider an example from our Car Logistics scenario and assume that the abstract activity "Reach Terminal" is ready to be executed by the flow engine (Figure 4.7).

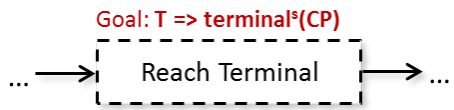


Figure 4.7: Abstract activity "Reach Terminal".

The object diagrams related to the flow instance are displayed in Figure 4.8, with the current configurations highlighted. For the car position (CP) and the terminal location (TL), the current configuration is the initial one. Only for the car status (CS) the current configuration is different.

We consider also that the set of flow fragment models available for vertical adaptation contains the two fragments presented in Figure 4.9.

Given these inputs, the vertical adaptation returns the fragment model in Figure 4.10.

4.2.4 Future Work

As already mentioned, in our work we deal with vertical adaptation only with respect to the control flow. Therefore, one important future direction is to add data to our models. Here, we envision two main challenges. First, we will need to ensure that the data required by the activity refinement can be obtained from the flow instance data. Secondly, we will also need to ensure that the data required in the remaining, not yet executed part of the flow instance can be obtained from the original flow instance data and the activity refinement data.

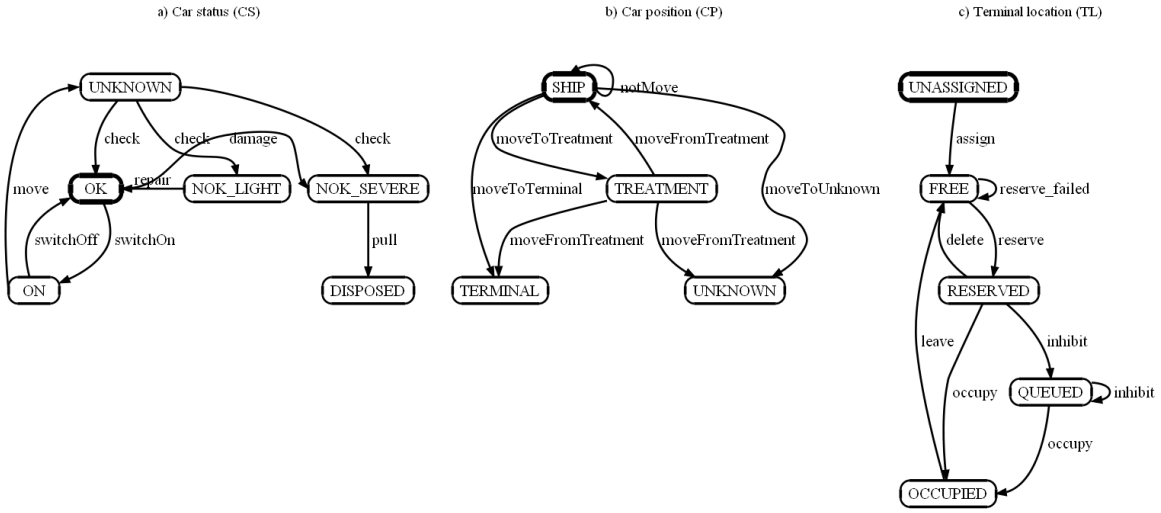


Figure 4.8: Object diagrams with highlighted current states

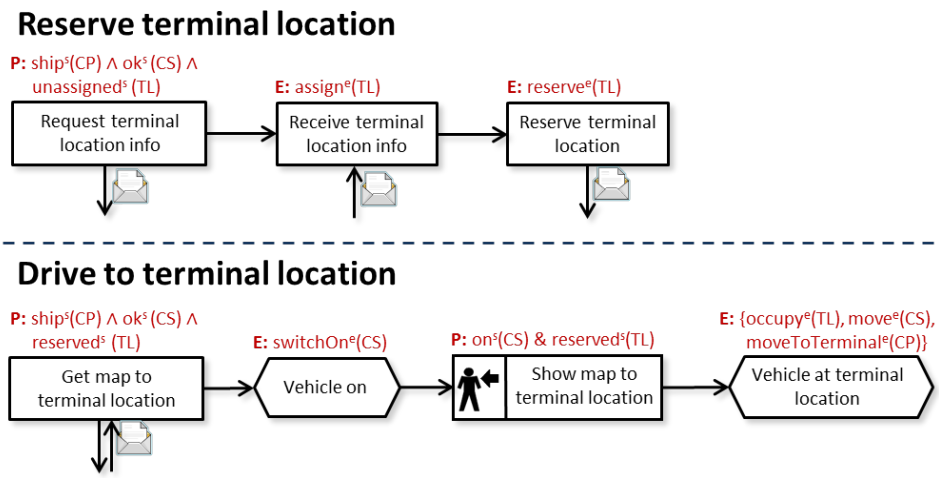


Figure 4.9: Reach Terminal Fragments

We also plan to devise mechanisms for selecting among different refinements based on higher-level QoS preferences. For this purpose, we will add QoS information to the flow fragment models available for composition. The QoS values for each fragment will also be linked to the context.

Finally, we will implement and evaluate the performance of vertical adaptation using the Car Logistics scenario. We plan to measure the behavior of vertical adaptation when scaling the number of available flow fragments. When using also QoS information, we plan to evaluate the benefit of vertical adaptation, that is, the performance of the adapted flow instances according to the KPIs of the application.

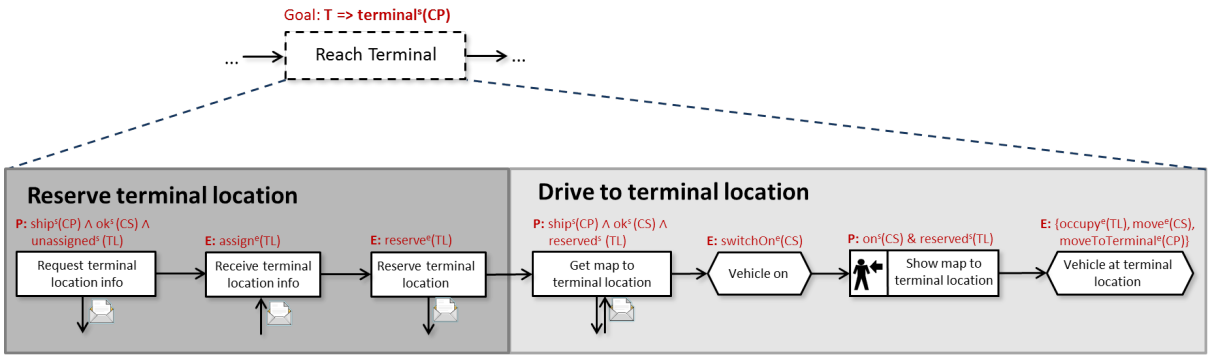


Figure 4.10: Result of vertical adaptation

4.3 Horizontal Adaptation by context-aware replanning

4.3.1 Problem Description

A flow model is composed of activities that an application has to perform to achieve a particular goal. Thus, it implements a process (i.e., to unload a car from a ship and move it to the terminal area) by describing activities as well as their context constraints (i.e., the car can not be damaged during the process execution). If during the flow execution some context constraint is violated we should guarantee that the flow execution is not stopped and the goal is still achieved. Horizontal adaptation is an adaptation mechanism that comes into play when a context constraint is violated and the flow stops its execution. It affects the structure of the flow model trying to adjust it to reach the initial goal.

Respect to the approach presented in the Deliverable D5.1 [3], where the adaptation logic was hard-coded into the application, here we are interested at adapting the application to unexpected changes. In the next sections we define our solution that consider the problem to change the flow structure, to solve a constraint violation, as a planning problem executed in a precise context. This means that instead of generating a plan in a static way (i.e., at design time as in Section 3.4) we generate plans during the flow execution considering the runtime state of the application. The result plan helps us to continue the flow execution trying to come back to a stable state of the application where the initial goal can be achieved. We first define, in a formal way, the main ingredients needed to have flow-based adaptable systems, then we define the architecture that implements horizontal adaptation outlining the main principles of its operation.

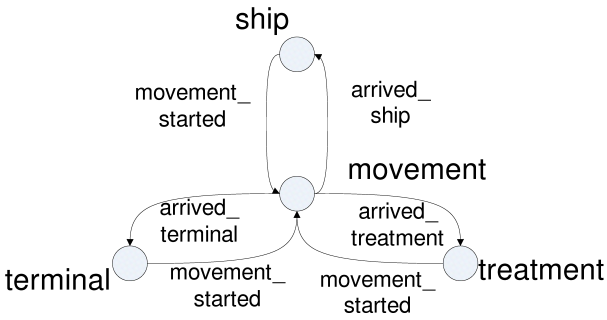
4.3.2 Solution

In this section the solution for horizontal adaptation based on planning techniques is presented. For this purpose, we (1) give a set of formal definitions for the elements of the formal framework and (2) describe the architecture of the formal framework and the way it operates.

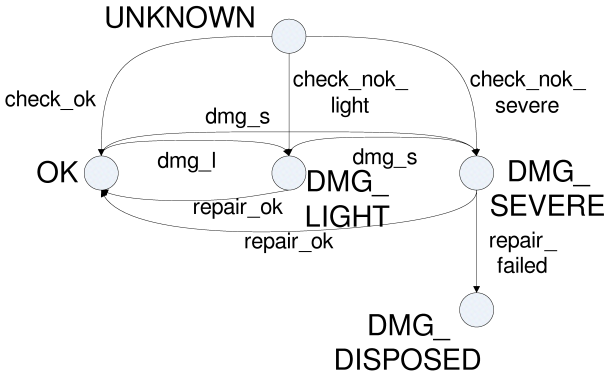
The formal framework is basically needed to describe the flow to be executed and, if needed, adapted and an environment in which this flow runs.

Aspect

Aspect represents some important characteristic of the environment that can change with the time. We model the evolution of an aspect with an aspect diagram, which defines all possible states of an aspect and transitions between them. Each transition is labeled with one or a few aspect-specific events. The state of the aspect can be changed as a result of fragment execution or due to contextual changes. As such, we can consider flows as a means of changing different aspects of the system. The concept of an aspect diagram is closed to the one of the domain object (see Definition 3.4.7) for service composition. The main difference is that the object diagram is a complete representation of some real object, so that



(a) Car Position



(b) Car Status

Figure 4.11: Aspect diagrams for the car position and car status.

any service always relates to exactly one object. For the *simplicity of modelling* in horizontal adaptation tasks, it is important to be able to consider different aspects of an object separately. In this case, a fragment, which is the analogue of service, can relate to a few aspects at a time.

Definition 4.3.1 *(Aspect Diagram) Aspect diagram a is a tuple $\langle L, L_0, E, T\rangle$, where:*

- *L is a set of states and $L_0 \subseteq L$ is a set of initial states;*
- *E is a set of aspect-specific events;*
- *$T \subseteq L \times (E \times L)^+$ is a transition relation allowing for non-deterministic transitions;*

We remark that domains normally need more than one aspect diagram in order to be described completely. In this case, we deal with the *system of aspect diagrams A*, whose state is a product of all possible states of aspect diagrams included.

Example. In Fig. 4.11, two simple aspect diagrams for the car position and car status are shown. As in the formal definition, they have a number of states and transitions labeled with events. In aspect diagram for the car status, the one of the transitions is non-deterministic (the result of initial car check is unpredictable).

Activity

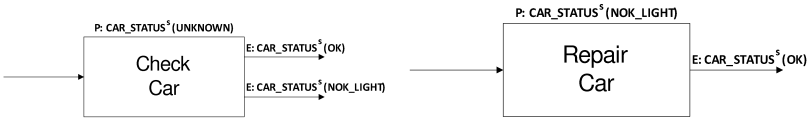
Each activity represents an atomic operation that is intended to make the particular aspects of the domain evolve. As opposed to transitions in STS, activities are not immediate and last for some time, so that during the activity execution other contextual changes may occur. Each activity has a precondition, an

initial effect and one or a few final effects. A precondition specifies all possible states of the system of aspect diagrams of the domain, in which the execution of the activity is possible. The initial effect shows how the start of the execution affect changes the system of aspects. The final effects show all possible consequences of the activity execution. Each effect is expressed in terms of aspect-specific events (it is a set of events that has to be published when the activity is initiated or finalized respectively). An effect can also be empty. More formally:

Definition 4.3.2 *(Activity) Activity act is a tuple* $\langle p, d^I, D^O \rangle$*, where:*

- $p \subseteq L(a_1) \times L(a_2) \times \ldots \times L(a_n), a_i \in A$ *is a precondition;*
- d^I *is an initial effect and* D^O *is a set of final effects, such that* $\forall e \in \{d^I, D^O\} : e \subseteq \bigcup_i E(a_i), a \in A$*;*

Example In Fig. 4.12, two activities are presented. One of them has only one possible effect, while the other has two (again, the result of checking the car is unpredictable). Both activities have preconditions (placed above the boxes).



(a) check car (b) repair car

Figure 4.12: Activities for checking and repairing the car.

Fragment

Fragment is a flow of activities that implements a higher level functionality compared to a single activity. A scope inside a fragment is a subset of activities and states of this fragment. A few scopes can be defined for one fragment. A constraint over the states of aspect diagrams can be defined for each scope. The execution can go on inside a scope unless the constraint is violated.

Definition 4.3.3 *(Fragment) Fragment* f *is a tuple* $\langle S, S_0, Act, R, SC, C \rangle$*, where:*

- S *is a set of states and* $S_0 \subseteq S$ *is a set of initial states;*
- Act *is a set of activities;*
- $R \subseteq S \times act \times (D^O(act) \times S)^+, act \in Act$ *is a transition relation;*
- $\forall sc \in SC = \langle \mathcal{S}, \mathcal{A} \rangle : \mathcal{S} \subseteq S, \mathcal{A} \subseteq act$ *is a scope;*
- $C : sc \Rightarrow c, sc \in SC, c \subseteq L(a_1) \times L(a_2) \times \ldots \times L(a_n), a_i \in A$ *is a constraint relation that specifies constraints for the fragment scopes;*

The model of a *flow* coincides with the model of a fragment. A flow π is correct regarding fragment f if $\forall (s_i, act_i, \{\ldots, (d^O_m, s_j), \ldots\}), (s_j, act_j, \{\ldots\}) \in R(\pi) : \exists (s_k, act_i, \{\ldots, (d^O_m, s_l), \ldots\}), (s_l, act_j, \{\ldots\}) \in R(f)$.

In other words, the correct flow has to follow the activity order defined in the fragment. Similarly, a flow is correct regarding a set of fragments if it respect the activity order of all fragments.

Example. In Fig. 4.13, a fragment for repairing the car is shown. It contains only one activity and also adds a scope that contains this activity. The constraint for the scope requires that while the car is being repaired it should stay at the treatment area and cannot change its position.

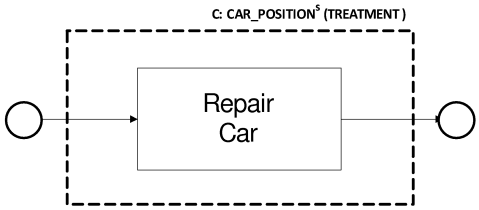


Figure 4.13: Fragment for car repairing process.

Adaptive System

An adaptive system ξ is the representation of an environment in which the flow can be executed. It contains a set of aspect diagrams and fragments. A set of configurations of an adaptive system is defined as $S(f_1) \times \ldots \times S(f_m) \times L(a_1) \times \ldots \times L(a_n), f_i \in F, a_j \in A$ (we also distinguish purely fragment and aspect configurations defined as $S(f_1) \times \ldots \times S(f_m)$ and $L(a_n), f_i \in F$ respectively). A set of initial and final configurations are specified.

Definition 4.3.4 *(Adaptive System) Adaptive system ξ is a tuple $\langle A, F, \mathcal{I}, \mathcal{G}, \mathcal{C} \rangle$, where:*

- *A is a set of aspect diagrams;*
- *F is a set of fragments defined over A;*
- *$\mathcal{I}, \mathcal{G}, \subseteq S(f_1) \times \ldots \times S(f_m) \times L(a_1) \times \ldots \times L(a_n), f_i \in F, a_j \in A$ are sets of initial and final configurations;*
- *$\mathcal{C}$ is a set of possible context changes (context events) such that $\forall c \in \mathcal{C} : c \subseteq \bigcup_i E(a_i)$;*

Main Flow

The main flow M for adaptive system ξ is a flow that is correct for the system of fragments $F(\xi)$ and, when executed, *may* leads the system from any of its initial configurations $\mathcal{I}(\xi)$ to one of its final configurations $\mathcal{G}(\xi)$. An execution of the main flow that leads the system to one of the goal states is called successful. All system configurations through which the system passes during the successful executions are called *stable configurations* $\mathcal{S}(\xi, M)$. The semantics of stable configurations is such that as soon as the adaptive system ξ is in one of its configurations $\mathcal{S}(\xi, M)$ the main flow M can be used to "move" towards goal states. On the contrary, if the adaptive system is not in one of its stable configurations we should run the adaptation mechanism to either bring the system back to one of stable states or to bring it to one of the goal states directly. The main flow represents the "ideal" plan for changing the system, where no context events occur and all activities finish with desirable effects. In this regards, successful execution of the main flow, which is, as a rule, the most probable one, do not need any adaptation activities.

Adaptive System Configuration

Adaptive system configuration represents the current configuration of the adaptive system, more precisely the current configurations of all its fragments and aspect diagrams.

Definition 4.3.5 *(Adaptive System configuration) Adaptive system configuration $\mathcal{R}(\xi)$ is a tuple $\langle \mathcal{P}, \mathcal{F} \rangle$, where:*

- *$\mathcal{P} \in L(a_1) \times \ldots \times L(a_n), a_i \in A(\xi)$ is the aspect configuration relation;*
- *$\mathcal{F} \in S(f_1) \times \ldots \times S(f_n), f_i \in F(\xi)$ is the fragment configuration relation;*

4.3.3 Architecture

In this section, we describe the architecture of the system that implements horizontal adaptation of a flow and outline the main principles of its operation. The architecture itself can be seen in Fig. 4.14. The figure is based on the definitions and symbols introduces in the previous section. The architecture contains three main elements: Orchestrator, Adaptor and Environment (split into Flow Engine and Context Manager).

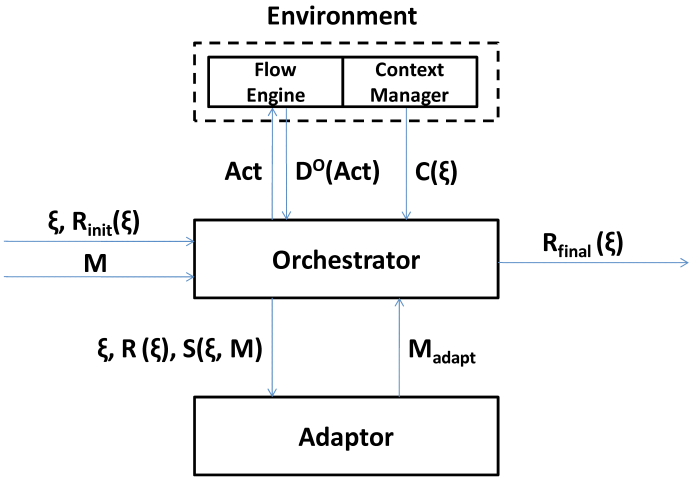


Figure 4.14: Horizontal Adaptation Architecture.

Orchestrator

The Orchestrator coordinates the work of all other elements. In particular, it:

- stores and updates the runtime configuration of the model;
- detects the situations in which the adaptation is required and triggers the adaptation;
- integrates the adaptation flow with the main flow;
- handles context changes;
- triggers the execution of the flow;

As an input the Orchestrator takes main flow M to be executed, adaptive system ξ describing the environment in which the main flow is to be executed, and the initial configuration of the adaptive system $\mathcal{R}_{init}(\xi)$.

The execution of any flow π implies the following steps:

1. The Orchestrator calculates the set of stable configurations $\mathcal{S}(\xi, \pi)$. In fact, stable configurations are those configurations of the adaptive system from which the successful execution is still possible. The Orchestrator proceeds to Step 2;
2. If the current configuration $\mathcal{R}(\xi)$ belongs to a set of stable configurations and is not terminal for π, then the next activity to be executed (act) is fetched. If the constraints for act are not violated, then the activity is sent to the Flow Engine and proceeds to Step 3. Otherwise ($\mathcal{R}(\xi)$ is not stable or is terminal or constraints for act are violated), the execution is stopped;
3. The system listens to events from the environment. If an effect $d^O \in D^O(act)$ comes from the Flow engine, then activity is considered as completed and Orchestrator proceeds to Step 4. If a

context change $c \in \mathcal{C}$ happens, the current configuration of ξ is updated immediately. If after the update the constraints of act become violated the Orchestrator proceeds to Step 5. Otherwise the Orchestrator repeats Step 3;

4. The activity act is considered as completed and the state of the corresponding fragment and the states of the aspect diagrams are updated with respect to d^O. The Orchestrator proceeds to Step 6;

5. The activity act is considered as failed and the state of the fragment remains the same. The Orchestrator proceeds to Step 6;

6. For each fragment $f \in F(\xi)$, if for the current state of f the constraints are violated, the fragment "rolls back" to the nearest previous state where the constraints are not violated. The Orchestrator proceeds to Step 2;

At high level of abstraction, the Orchestrator operates as follows:

1. the main flow M is executed. If the adaptive system is finally in one of its final configurations $\mathcal{G}$, then the overall execution is finished and the current configuration is returned; If the current configuration is unstable (which may happen if some fail or context change has occurred), the Orchestrator proceeds to Step 2;

2. The Orchestrator sends to the Adaptor a request for adaptation, that includes the adaptive system ξ, the current configuration $\mathcal{R}(\xi)$ and the set of stable configurations $\mathcal{S}(\xi, M)$. In response, the Adaptor returns adaptation flow M_{adapt}. The orchestrator proceeds to Step 3;

3. The Orchestrator executes M_{adapt}. If the final configuration is unstable, the Orchestrator proceeds to Step 2, otherwise to Step 1;

Adaptor

Adaptor is a tool that, given an adaptive system in an unstable configuration, generates an adaptation flow that is intended to bring the system either to one of its stable configurations or to one of its goal configurations. This part of the architecture implements the horizontal adaptation logics (using AI planning techniques).

The operation of the Adaptor consists in transforming the adaptation problem for the adaptive system ξ, in current configuration $\mathcal{R}(\xi)$ with stable configurations $\mathcal{S}(\xi, M)$ into a planning problem, which is then solve using the aforementioned planning techniques. In particular, the adaptive system ξ is transformed into a planning domain, current configuration $\mathcal{R}(\xi)$ into the initial states over the domain and stable configurations $\mathcal{S}(\xi, M)$ into the goal states over the domain. The plan obtained is translated into the adaptation flow M_{adapt}. The both translations are quite straightforward, so that we do not give further details here.

It is important to notice that the adaptor creates an "optimistic" plan in a sense that it assumes that all activities produce desirable outcomes and no context changes occur during execution. Such an approach makes sense in case all abnormal behaviors are quite improbable (which is true in out application domain). Once these failures happen during the execution, another round of adaptation takes place.

Environment

Environment consists of two main parts. *Flow Engine* executes activities of the flow in the environment and reports the result of the execution (if any). *Context manager* checks the status of the environment and publishes context events if needed. Context Manager guarantees that the state of the adaptive runtime system store in orchestrator is synchronized with the environment. If during the execution of the activity, the Context Manager reports context changes that violate the constraints of the activity being executed, the further execution is skipped.

4.3.4 Scenario Implementation

In order to see the architecture for horizontal adaptation (Fig. 4.14) in action, we use a simplified version of the Car Logistics scenario presented in Chapter 2.

For that purpose, we define two aspect diagrams (Fig. 4.11) describing the car position and status respectively. Particularly, the car can be in one of the four places (SHIP, TREATMENT, TERMINAL and on the move (MOVEMENT) from one of three aforementioned places to another). The status of the car can be UNKNOWN (upon the car arrival), OK (if the car is not damaged), DMG_LIGHT (light damage), DMG_SEVERE (severe damage) and DMG_DISPOSED (the car cannot be repaired). Light damage means that the car can move but must be repaired as soon as possible, while severe damage prevents the car from moving. Being damaged, the car can still be repaired, unless it is unrepairable (DMG_DISPOSED).

We also define the following set of activities (some of them are show in Fig. 4.12):

- **CarCheck** checks the current status of the car. It can be executed only if the current status of the car is unknown and the car is not on the move. The outcome reports the current state of the car (leads car status from UNKNOWN to OK, DMG_LIGHT or DMG_SEVERE);
- **MoveToShip, MoveToTreatment, MoveToTerminal** move the car from the current location to a corresponding destination. The preconditions for all three require that the car is not damaged (OK). **MoveToTreatment** can also be executed if the car is slightly damaged (DMG_LIGHT). We also specify that the car can be moved to the terminal only from the ship;
- **PullToTreatment** is executed to move the severely damaged car (DMG_SEVERE) to the treatment area from any location, including when the car is broken during the movement (MOVEMENT);
- **CarRepair** is used to repair the damaged car. The precondition requires that the car is not ok and it is at the treatment area. As a result of the execution the car is either repaired (OK) or claimed to be unrepairable (DMG_DISPOSED);

On the basis of the given activities we define a number of fragments and a main flow. For this example, we use very simple fragments containing only one activity per fragment (as it is shown in Fig. 4.13). For this simple fragments we additionally defined fragment scopes and constraints (for example, during the car adjustment the car cannot be moved and must stay at the treatment area). We also specify the main flow as it is shown in Fig. 4.15. Finally, the set of contextual events $\mathcal{C}$ contains only two events corresponding to aspect events *dmg_l* (the car is slightly damaged) and *dmg_s* (the car is severely damaged). The goal of the execution is to move the car undamaged (OK) to the terminal (TERMINAL).

Below we give one of the possible executions of the adaptive system containing all the element described above and show how the adaptation process works.

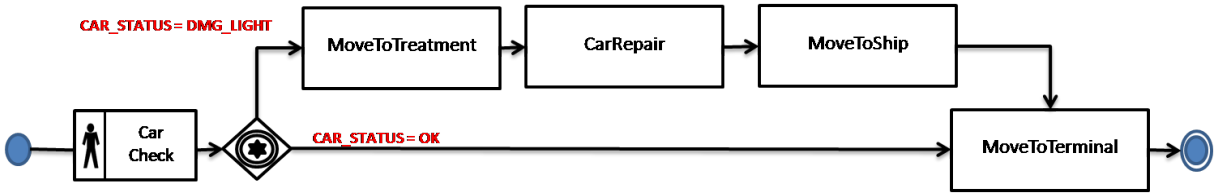


Figure 4.15: Main Flow.

In the initial state, the car is on the ship (SHIP) and its status is unknown (UNKNOWN). The Flow Engine starts the execution of the main flow, of Figure 4.15, executing the **CarCheck** activity. As a result, it turns out that the car has a light damage (car status is DMG_LIGHT). Although, light damages is a sort of failure on the way to the car successful delivery, the adaptation is not needed since the case of light damage is already considered and embedded into the main flow (in other words, the aspect

configuration $CarPosition = SHIP\&CarStatus = DMG\_LIGHT$ belongs to the stable states $\mathcal{S}(\xi, M)$). According to the main flow, activity **MoveToTreatment** is subsequently executed (as we have mentioned, the car can move by itself even if it has light damages). While moving to the treatment, the car is severely damaged and stops (it actually remains in location MOVEMENT). The current configuration of the adaptive system is not a stable state (the flow engine does not know how to proceed), that is why the adaptation is required. The main flow is suspended and all necessary information are passed to the Adaptor to find the adaptation strategy for the current case. The Adaptor returns he adaptation flow M_{adapt} (Fig. 4.16) that suggests to pull the car to the treatment area and repair it. If executed without failures and in the lack of contextual events, the adaptation flow brings the car to one of stable states, from which the execution of the main flow will be possible. Indeed, after the car is pulled to the treatment and successfully repaired (activity **CarRepair** finishes with event *repair_ok*), the car is not damaged anymore (OK) and is located at the treatment area (TREATMENT). From here, the main flow can be executed (it is important that we execute the main flow from a different point that the one where we previously stopped; we do not store the status of the main flow). More precisely, the activities **MoveToShip** and **MoveToTerminal** are executed, which brings the adaptive system to the goal configuration.

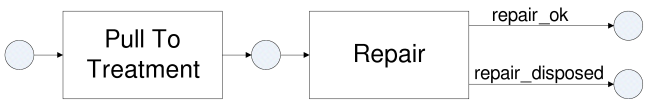


Figure 4.16: Adaptation flow.

Finally, Figure 4.17 shows how the main flow has been adapted, it contains all the activities that have been really executed by the Flow Engine.

Figure 4.17: Activities executed by the Flow Engine.

4.3.5 Future Work

In this section we have presented how, using a planning technique, we are able to perform horizontal adaptations during a flow execution. We have provided a formal framework with its implementation (defining a precise architecture). We have started also to execute it with our running scenario. For the next period we plan to evaluate its execution performance using the same scenario but with more complex flows and adaptation examples. We plan to measure its behavior scaling the number of available activities, flow fragments, but also application aspects (i.e., aspect diagrams). Using also QoS information, we want to evaluate the benefit of the adaptation calculating the performance of the adapted flow instances according to some Key Performance Indicators (KPIs).

Chapter 5

Evolution

5.1 Problem Description

In the previous chapters we have seen how to adapt flows using vertical and horizontal adaptation mechanisms. Over time, these two mechanisms are applied on single instances when an abstract activity need to be refined or when a context constraint is violated (i.e., a car is damaged). Flow evolution, respect to flow adaptation, affects a flow model, thus modifying the characteristics of all the flow instances that will be based on that model; and it is based on the analysis of histories of flow instance execution and adaptations. The set of adapted flow instances together with the information concerning their success will be used as training cases for evolution mechanisms in order to progressively improve flow prototypes that may then be used if a new flow shall be instantiated.

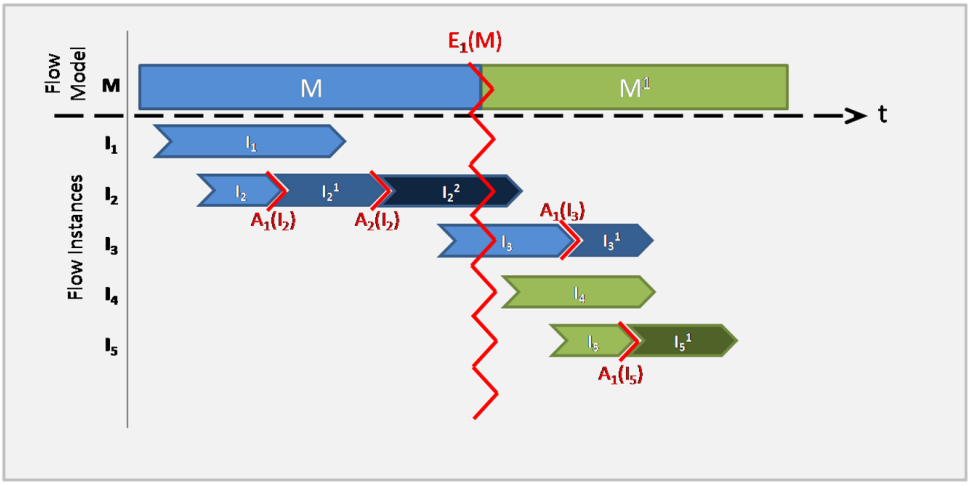


Figure 5.1: Flow Adaptation and Evolution

Figure 5.1 captures the core concepts on adaptation and evolution described so far. It focuses on a specific flow model M and a set of flow instances, $I1$ to $I5$, all instantiated on model M and its evolution M^1. As can be noticed, there are flow instances that conclude their execution without any adaptation (i.e. $I1$ and $I4$), while others need to be adapted one or more times (i.e. $I2$, $I3$, and $I5$). M^1 is used to instantiate the flow executions starting after the evolution $E1(M)$, that is $I4$ and $I5$. All the instances already running, based on M, are not affected by evolution $E1(M)$, even when they occur in . The problem that we start to analyze in this chapter is how to use historical data on adaptation executions to generalize the solutions evolving an initial flow model.

In the following sections we describe first a lifecycle that we need to execute, secondly we describe the architecture that we need to implement to concretize the solution proposed.

5.2 Solution

In this section the initial solution for the evolution problem is presented. Respect to the adaptation problems, here we do not provide a formal framework and its implementation but only an high-level and general solution, starting to understand what should be the main elements to consider. The goal of the next deliverable will be the complete implementation and evaluation of an evolution framework that uses these preliminary ideas. For this purpose, we propose a general life-cycle to manage evolution and a possible extension of the architecture presented in Figure 4.14 (see Figure 5.3) adding components needed to analyze the system executions and adaptations and propose flow evolutions. The life-cycle shown in Figure 5.2 highlights the phases needed to analyze the flow instance executions and adaptations in order to introduce permanent and , usually, important adjustments to a flow model. In the following sections we explain in detail each of them. After this life-cycle we show how the architecture introduced to implement horizontal adaptation should be extended to manage also evolution.

5.2.1 Evolution Lifecycle

Figure 5.1 shows the life-cycle to manage evolution of flow-based pervasive systems. At runtime each flow instance of a designed flow model M is executed and probably adapted to react at some constraint violations. Respective instances execution and adaptation are logged in an execution log. Based on the information from logs, we can analyze it (i.e., *Execution Log Analysis*) and find a way to evolve the flow model in a new one M^1 (*Evolution Enactment*). The analysis phase is responsible of (1) evaluate the quality of execution of its flow instances and relative adaptations and (2) deciding the need for evolution for a certain flow model. While step (2) is performed at the level of the flow model, that is, evaluating all the flow instances related to that model, step (1) requires a finer grain of evaluation. As a matter of fact, the aim of step (1) is to identify good flow adaptations that can be used to define a new version of the flow model.

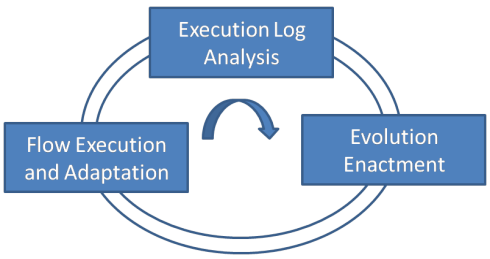


Figure 5.2: Flow Evolution LifeCycle.

Analyzing flow adaptations of a precise instance requires the definition of a set of metrics to evaluate their goodness. For examples quality metrics like the following should be included and should be used to decide the need for evolution.

- (1) Standard qualitative metrics for flow evaluation such as frequency of usage, completion time, failure/success rate, etc..;
- (2) Ad-hoc qualitative metrics targeted to the specific domain (e.g. percentage of damaged/disposed cars in the Car Logistic domain);
- (3) Adaptable flow specific metrics, such as average number of adaptations per flow model, occurrence of a specific flow adaptation, average time required by each kind of adaptation, etc..

For each flow model, is also important to classify its adaptations, define a precise ranking, and to select which of them should be embedded within the new flow model. For example we can rank

adaptations that are good wrt the qualitative metrics defined in (1) and (2) or flow adaptation which are frequent/time consuming wrt the metrics in (3).

Once we have detected the need to evolve a flow model and identified the flow adaptations that, thanks to their goodness, should be embedded within the new model, we should simply plug them within the original flow. To do this we need to have modeling tools that to allow to easily embed adaptation variants within a flow model. The WS-BPEL extensions proposed in [18] come in our help and can be easily used to change a flow model inserting the selected adaptations. These extensions have been defined to provide a way to design built-in adaptations directly in a flow model.

5.2.2 Evolution Architecture

After the description of the evolution life-cycle, in this section we describe the architecture of the system that implements evolution of a flow model. It is an extension of the architecture proposed for horizontal adaptation and can be seen in Figure 5.1. The new component added respect the horizontal problem is called *Evolution Manager* and is composed of two subcomponents: the *Execution Log* and the *Execution Analyzer*. Starting from the flow model definition M all its instances are created and executed.

Since that each flow is attached to a precise entity (i.e., a car) of a pervasive system, and since that we should manage more entities at the same execution time, we have several instances running of the same flow model M. Each instance can be adapted any time to react at a context constraint violation. In order to understand how the flow model has been executed during its life, we must be able to memorize all the actions that effects the state of the system in a chronological order. This means that during each flow instance execution, in the *Execution Log* component we should be able to record data on all changes, that have been applied. These information contain each activity initialization, each activity outcome, each contextual events and each adaptation, that change the state of the system.

The analysis phase, using a precise set of metrics, and the set of historical data, (1) identifies the set of most common execution pattern and (2) ranks them using precise metrics. When the final set of patterns has been created they can be embedded, using the built-in adaptation constructs defined in [18], in the flow model M generating the new flow model M^1.

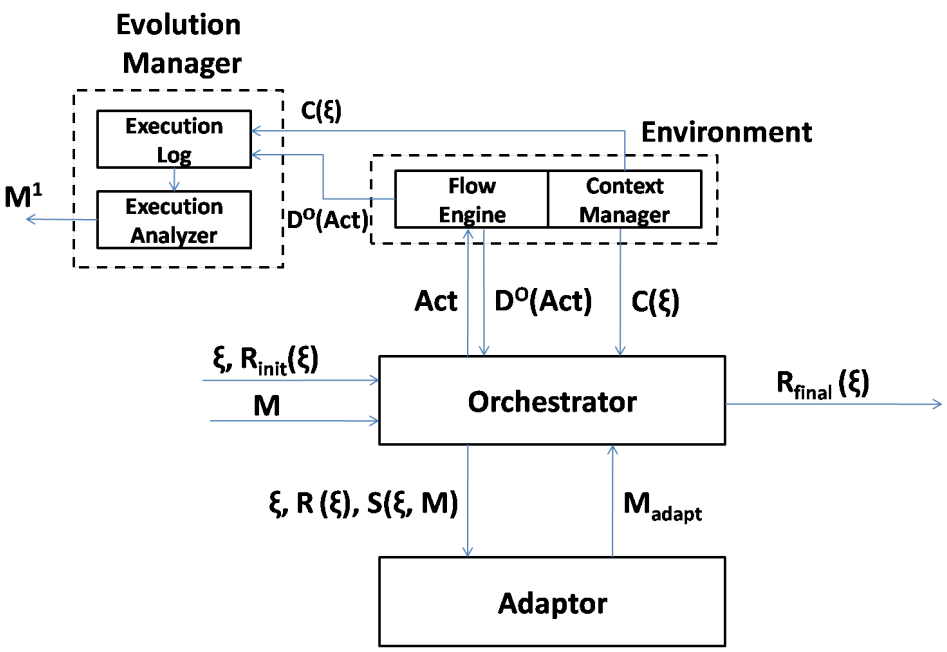


Figure 5.3: Evolution Architecture.

5.3 Future Work

In this chapter we have started to analyze the evolution problem giving an idea on what are the main elements that we need to consider any time that we want to evolve a flow. We have not provided concrete techniques but starting from this preliminary analysis our plan is to extend and implement the architecture depicted in Figure 5.3. This means that we need first to identify techniques to log execution histories and analyze them [24,26]. Moreover we need also to understand how to discover common execution patterns and generalize them in a flow model evolution [17]. All the solutions proposed will be evaluated using our reference scenario.

Chapter 6

Conclusion

In this document we showed theories and mechanisms to adapt and evolve flow-based pervasive systems. In particular, we have proposed formal frameworks (i.e., based on Planning Techniques) to support different runtime adaptations (vertical and horizontal). Moreover we have started to analyze the problem of flow evolution trying to identify the main phases of an evolution life-cycle and the main information that we need to log during flow executions and adaptations. To understand better the solutions proposed we have used a common scenario. As future work, we have in mind to consider the following open issues:

- **Validation of the Adaptation Mechanisms:** we will evaluate the performance of vertical and horizontal adaptation mechanisms proposed in this deliverable using the Car Logistics scenario introduced in Section 2.1. We plan to measure the behavior of them scaling the number of available flow fragments. Using also QoS information, we plan also to evaluate the benefit of these adaptations calculating the performance of the adapted flow instances according to some Key Performance Indicators (KPIs) of the application.

- **Implementation and Validation of the Evolution Architecture:** In this deliverable the Evolution problem has been started. We have identified what are the main phases to execute to evolve a flow model after a certain number of adaptations. We plan to implement completely the architecture proposed in Figure 5.1 and to validate this mechanism using the same scenario used for adaptation.

- **Implementation and Validation of an Overall Adaptation Process:** In this deliverable we have presented adaptations that are executed by replacing or rearranging tasks in a flow such that the flow's goals can still be fulfilled despite the fact that various failures and problems have occurred. Beyond that, a number of other aspects need to be controlled while a flow is running such as: the user interfaces, flow security, and flow distribution. The adaptation strategies for these different aspects of the flow execution are stored and executed by the components in control of these aspects, such: security manager, distribution manager and so forth. Having each component adapt its own concerns independently of the rest of the system, makes the adaptation much more robust since the adaptation knowledge and strategies are decentralized. This in turn makes the components loosely-coupled and the system more robust in face of component failures. However, this also introduces a problem of coordinating these different independent adaptations to avoid conflicts, oscillations and race conditions among their adaptation actions and goals. In the next deliverable we want to propose an approach for coordinating a set of autonomous components that can adapt different execution aspects of a common pervasive system. This is important to allow the components to retain their autonomy and authority over the execution aspects that they are overseeing, while undesirable effects such as conflicts, oscillations and race conditions are avoided.

Bibliography

[1] *OASIS: Web Services Business Process Execution Language Version 2.0, Organization for the Advancement of Structured Information Standards*, 2007.

[2] D3.1: Basic flow-model and language for Adaptable Pervasive Flows. ALLOW Project Deliverable, November 2008.

[3] D5.1: Theory and mechanisms for horizontal flow adaptation by built-in context-awareness. ALLOW Project Deliverable, November 2008.

[4] D5.2: Theory and mechanisms for vertical flow adaptation by flow composition. ALLOW Project Deliverable, August 2009.

[5] D3.2: Declarative language for goals, constraints, adaptation and evolution. ALLOW Project Deliverable, January 2010.

[6] Piergiorgio Bertoli, Raman Kazhamiakin, Massimo Paolucci, Marco Pistore, Heorhi Raik, and Matthias Wagner. Control flow requirements for automated service composition. In *Proceedings of the 2009 IEEE International Conference on Web Services*, ICWS '09, pages 17–24, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of web services via planning in asynchronous domains. *Journal of Artificial Intelligence*, 174(3-4):316–361, 2009.

[8] F. Bose and K. Windt. Autonomously controlled storage allocation on an automobile terminal. In K. Windt (eds.) in M. Hulsmann, editor, *Understanding Autonomous Cooperation and Control in Logistics - The impact in Management, Information and Communication and Material Flow.*, pages 351–363. Springer, Berlin, Germany, 2007.

[9] Antonio Bucchiarone, Cinzia Cappiello, Elisabetta Di Nitto, Raman Kazhamiakin, Valentina Mazza, and Marco Pistore. Design for adaptation of service-based applications: Main issues and requirements. In *ICSOC/ServiceWave Workshops*, pages 467–476, 2009.

[10] Antonio Bucchiarone, Alberto Lluch-Lafuente, Annapaola Marconi, and Marco Pistore. A formalisation of adaptable pervasive flows. In *WS-FM*, pages 61–75, 2009.

[11] Hanna Eberle, Tobias Unger, and Frank Leymann. Process fragments. In *OTM Conferences (1)*, pages 398–405, 2009.

[12] E. Allen Emerson. Temporal and modal logic. In *HANDBOOK OF THEORETICAL COMPUTER SCIENCE*, pages 995–1072. Elsevier, 1995.

[13] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, Amsterdam, 2004.

[14] Klaus Herrmann, Kurt Rothermel, Gerd Kortuem, and Naranker Dulay. Adaptable pervasive flows - an emerging technology for pervasive adaptation. In *SASO Workshops*, pages 108–113, 2008.

[15] Antonio Bucchiarone Ivan Lanese and Fabrizio Montesi. A framework for rule-based dynamic adaptation. In *Proceedings of TGC 2010, 5th International Symposium on Trustworty Global Computing,*, pages 284–300. Springer Verlag, February 2010.

[16] Ugo Dal Lago, Marco Pistore, and Paolo Traverso. Planning with a language for extended goals. In *AAAI/IAAI*, pages 447–454, 2002.

[17] Chen Li, Manfred Reichert, and Andreas Wombacher. Discovering reference models by mining process variants using a heuristic approach. In *Proceedings of the 7th International Conference on Business Process Management*, BPM '09, pages 344–362, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Annapaola Marconi, Marco Pistore, Adina Sirbu, Hanna Eberle, Frank Leymann, and Tobias Unger. Enabling adaptation of pervasive flows: Built-in contextual adaptation. In *ICSOC/ServiceWave*, pages 445–454, 2009.

[19] Annapaola Marconi, Marco Pistore, and Paolo Traverso. Specifying data-flow requirements for the automated composition of web services. In *SEFM*, pages 147–156, 2006.

[20] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.

[21] Marco Pistore and Paolo Traverso. Planning as model checking for extended goals in non-deterministic domains. In *IJCAI*, pages 479–486, 2001.

[22] Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. Automated synthesis of composite bpel4ws web services. In *ICWS*, pages 293–301, 2005.

[23] Dmitry Shaparau, Marco Pistore, and Paolo Traverso. Contingent planning with goal preferences. In *AAAI*, 2006.

[24] Wil M. P. van der Aalst, Boudewijn F. van Dongen, Joachim Herbst, Laura Maruster, Guido Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.

[25] Wil M. P. van der Aalst and Kees M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[26] W.M.P. van der Aalst, A.J.M.M. Weijter, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16:2004, 2003.