Formalizing Collaboration Goal Sequences for Service Choreography

Humberto Nicolás Castejón and Rolv Bræk

NTNU, Department of Telematics, N-7491 Trondheim, Norway
{humberto.castejon, rolv.braek}@item.ntnu.no

Abstract. Methods for service specification should be simple and intuitive. At the same time they should be precise and allow early validation and detection of inconsistencies. UML 2.0 collaborations enable a systematic and structured way to provide overview of distributed services, and decompose cross-cutting service behaviour into features and interfaces by means of collaboration-uses. To fully take advantage of the possibilities thus opened, a way to compose (i.e. choreograph) the joint collaboration behaviour is needed. So-called collaboration goal sequences have been introduced for this purpose. They describe the behavioural composition of collaboration-uses (modeling interface behaviour and features) within a composite collaboration. In this paper we propose a formal semantics for collaboration goal sequences by means of hierarchical coloured Petri-nets (HCPNs). We then show how tools available for HCPNs can be used to automatically analyse goal sequences in order to detect implied scenarios.

1 Introduction

Many authors have identified the cross-cutting nature of distributed services (e.g. [8,5]) i.e. that services in the general case, involve several collaborating components playing different roles, that each may participate in several services. For service engineering, this implies a need to specify services in terms of their roles and cross-cutting service behaviour, then to specify the detailed behaviour of each service role and, finally, to compose the behaviour of service roles into complete, coordinated and correct component behaviours. UML 2.0 collaborations [7] provide language concepts and mechanisms that partially support this and are therefore very promising from a service engineering point of view. Being both structural and behavioural classifiers in UML 2.0, collaborations can be used to define a service as a structure of roles with associated cross-cutting behaviour defined using e.g. sequence diagrams. Detailed role behaviour can be defined using e.g. state machines. UML collaborations can be bound to specific contexts (e.g. larger collaborations) by means of collaboration-uses. This feature enables a compositional and incremental specification of services.

As an example consider a simple transport service (inspired by a case study from [12]) in which one vehicle transports one passenger at a time between two terminals. Figure 1a depicts this service as a UML 2.0 collaboration. This collaboration identifies three roles, namely P (Passenger), T (Terminal) and V

E. Najm et al. (Eds.): FORTE 2006, LNCS 4229, pp. 275–291, 2006.
© IFIP International Federation for Information Processing 2006

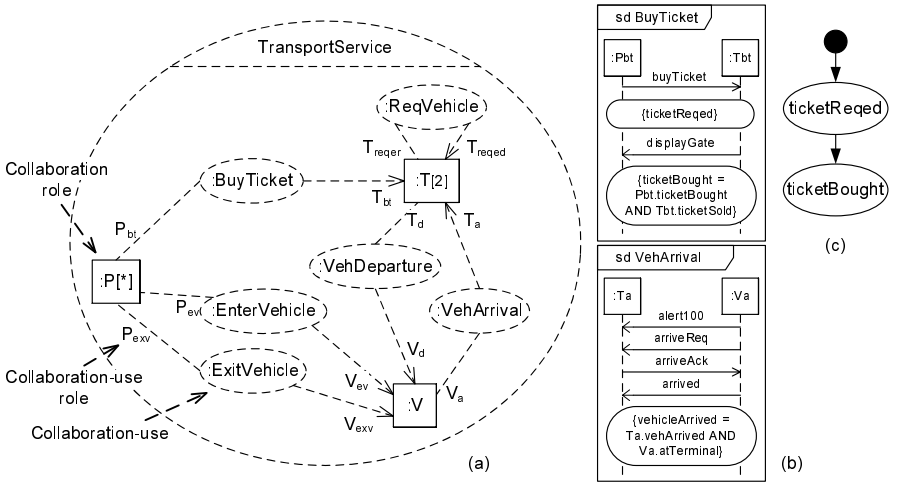

Fig. 1. (a) Transport service as a UML 2.0 collaboration; (b) Sequence diagrams for *BuyTicket* and *VehArrival* sub-collaborations; (c) Service-goal tree for *BuyTicket*

(Vehicle); as well as seven sub-collaborations representing interfaces and features of the service. These sub-collaborations are specified as UML collaboration-uses, whose roles are bound to the *TransportService*'s roles (e.g. *BuyTicket*'s role T_{bt} is bound to *TransportService*'s role T). Bound roles are classified as either *initiating* (i.e. takes the initiative to start the collaboration) or *offered* (i.e. accepts the initiative), indicated by an arrow head with offered roles. For the sake of clarity, in the following we will refer to P, T and V as service-roles, and to $T_{\mathrm{bt}}, T_{\mathrm{d}}$ and the like as sub-roles (of T, P or V). The *TransportService*'s sub-collaborations have been identified from the following service requirements. In order to travel, a passenger must buy a ticket at one of the terminals (collaboration-use *BuyTicket*). When this happens, if the vehicle is waiting at the terminal, the departure gate is indicated, and the passenger can enter the vehicle (*EnterVehicle*). The terminal then dispatches the vehicle (*VehDeparture*) and, after arriving at the second terminal (*VehArrival*), the passenger disembarks (*ExitVehicle*). If the vehicle is not at the terminal where the passenger buys the ticket, that terminal requests the vehicle from the other terminal (*ReqVehicle*), which dispatches the vehicle towards the requesting terminal. When the vehicle arrives, the departure gate is displayed and the service continues as explained before. In order to support validation and composition, service-goals [9] are associated with each of the identified sub-collaborations. These goals are expressed in terms of predicates over properties of the collaborations. Two types of service-goals can be described: event-goals, denoting desired events; and state-goals, which are properties of global collaboration states that we wish to reach, and which entail combinations of role goals. The ordered sequence of goals for an individual collaboration can be described with a *service-goal tree*, which is a directed graph with an initial node, zero or more intermediary nodes representing event-goals, and one or more

leaf nodes representing state-goals. Figure 1c shows the service-goal tree for *BuyTicket*, with an event-goal (i.e. *ticketReqed*) and a state-goal (i.e. *ticketBought*). Goal trees describe the behaviour of elementary collaborations at a high-level of abstraction, since the interactions are not detailed. These interactions can be specified in sequence diagrams annotated with goal information (by means of continuations), such as the ones presented for *BuyTicket* and *VehArrival* in Fig. 1b.

What remains in Fig. 1 is to specify the overall cross-cutting behaviour of the *TransportService* collaboration, that is, how its sub-collaborations interact. This kind of behaviour will be distributed among the collaboration roles and is traditionally referred to as a *choreography* in SOA. *Collaboration goal sequences* have been proposed by Sanders [9,11], and extended in [2], to describe the choreography of collaborations. They capture the liveness aspects of composite service collaborations by describing the execution order of their sub-collaborations, and by showing the interactions between these sub-collaborations in terms of goal achievement (hence the name collaboration *goal* sequences). While *service-goal trees* describe the sequence of goals for individual collaborations, *collaboration goal sequences* specify the sequence of goals for their composition. The information provided by the goal trees and the goal sequence should therefore be consistent. In the following we will assume this is the case.

In this paper, we present the formal syntax of goal sequences and provide semantics to them by means of hierarchical coloured Petri-nets (HCPNs) [4] (see Sect. 2). We also show how a general purpose tool for HCPNs (i.e. CPN Tools [3]) can be used to analyse goal sequences for the detection of implied scenarios (see Sect. 3). These scenarios are a direct consequence of concurrency and correspond to service behaviour that has not been explicitly described in the specification of the service, but that will be present in any implementation of it [1]. The proposed detection approach avoids a global analysis of the service specification, limiting thus the effect of the state-explosion problem. We end with related work and some discussion in Sects. 4 and 5.

2 Collaboration Goal Sequences

Collaboration goal sequences complement UML collaborations for the specification of services by describing the execution dependencies that exist between the sub-collaborations (i.e. features) of the service. As an example, Fig. 2 depicts the goal sequence for the *TransportService* collaboration. The actual meaning of this diagram will become clear in the following, when we explain the syntax and semantics of goal sequences.

2.1 Syntax for Goal Sequences

The goal sequences presented here are inspired by UML activity diagrams. Conceptually, they show an ordering of service phases for a service collaboration C. Each of these phases corresponds to an activity (i.e. round-cornered rectangle)

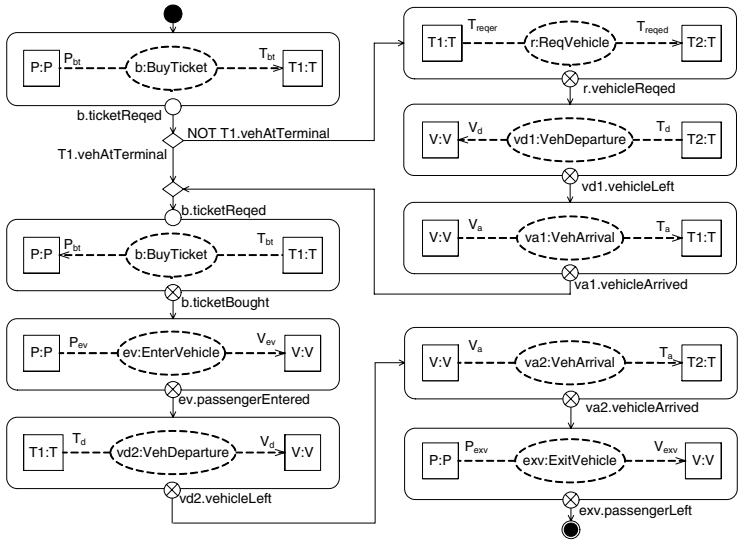

Fig. 2. Goal sequence for the *TransportService* collaboration

in the goal sequence. In each phase or activity, a specific sub-collaboration of C is active (so-called activity's *active collaboration*). This is represented by adorning the activity with a collaboration-use, whose roles are bound to instances of C's roles. For example, in Fig. 2, the *BuyTicket* collaboration is active in the first activity. This is expressed by adorning that activity with a *b:BuyTicket* collaboration-use, whose roles (i.e. P_{bt} and T_{bt}) are bound to instances of *TransportService*'s roles (i.e. *P:P* and *T1:T*). The arrow in the binding identifies the offered role. In a goal sequence, the same sub-collaboration may be active in several activities. In some cases these activities represent different phases of that sub-collaboration, while in other cases they represent different occurrences of the sub-collaboration. In the former cases activities are annotated with the same collaboration-use, such as the two first activities to the left in Fig. 2. They represent different phases of *BuyTicket* (i.e. before and after requesting the ticket) and are therefore annotated with the same collaboration-use (i.e. *b:BuyTicket*). In the latter cases, activities are annotated with distinct collaboration-uses, as for instance *va1:VehArrival* and *va2:VehArrival* in Fig. 2.

Each activity has one or more exit-points, and may or may not have one entry-point. Both entry- and exit-points represent execution points at which an activity's active collaboration interact with other collaborations. They are labeled with predicates describing goals of the active collaboration. Exit-points can be of two different types. An empty-circle (◯) is used for *suspension* exit-points. They are annotated with event-goals, and correspond to execution points of an active collaboration where the latter can be (or must be) suspended for another collaboration to be started (or resumed). In Fig. 2 a suspension exit-point is used in the first activity. The activity's active collaboration (i.e. *b:BuyTicket*)

will therefore be suspended when the *ticketRequed* event-goal of *BuyTicket* holds. A crossed-circle ($\otimes$) is used for *end-of-execution* exit-points. They are annotated with state-goals, and represent the end of execution of an active collaboration. Entry-points are drawn as empty circles and annotated with event-goals. They represent the execution point at which a previously suspended active collaboration is to be resumed. When an activity does not have an entry-point, its active collaboration starts execution from its initial state.

Edges (i.e. directed connections between activities) and control-flow nodes (i.e. decision, merge, fork, join, initial and final nodes) are respectively used to allow and coordinate the flow of control among activities. An activity can only have one incoming edge, so multiple incoming edges must be AND- or OR-joined.

According to the concrete syntax just described, the formal syntax of goal sequences can be defined as:

Definition 1 (collaboration goal sequence). *A collaboration goal sequence, for a collaboration C, is a tuple $GS = (N, E, g_{\mathrm{d}}, \ m_{\mathrm{exp-a}}, R_{\mathrm{GS}}, AC, m_{\mathrm{a-ac}}, m_{\mathrm{enp-a}}, l_{\mathrm{ep-pred}}, exp_{\mathrm{type}})$ where*

(i) *N is a set of nodes. It is partitioned into an initial node (n_0) and sub-sets of activities (N_{A}), entry-points (N_{EnP}), exit-points (N_{ExP}), control flow nodes (N_{FLOW}) and final nodes (N_{FI}). In turn, N_{FLOW} is partitioned into decision (N_{D}), merge (N_{M}), fork (N_{F}) and join (N_{J}) nodes.*

(ii) *$E \subseteq (N_{\mathrm{ExP}} \cup N_{\mathrm{FLOW}} \cup \{n_0\}) \times (N_{\mathrm{A}} \cup N_{\mathrm{EnP}} \cup N_{\mathrm{FI}} \cup N_{\mathrm{FLOW}})$ is a set of directed edges between nodes.*

(iii) *g_{d} is a guard function for decision nodes' outgoing edges. It is defined from $\{(s,t) \in E \mid s \epsilon N_{\mathrm{D}}\}$ into boolean expressions.*

(iv) *$R_{\mathrm{GS}} = \{(id, type) : type \in R_{\mathrm{C}}\}$ is a set of role instances, with R_{C} being the set of roles of collaboration C.*

(v) *$AC = \{(id, type, B)\}$ is a set of* active collaborations*, that is, a collaboration-use representing a specific occurrence of one of C's sub-collaborations. For each $ac \in AC$, id is the name of the collaboration-use; $type$ is the name of the collaboration that actually describes the collaboration-use (i.e. one of C's sub-collaborations); and $B \subseteq R_{\mathrm{type}} \times R_{\mathrm{GS}}$ is a set of role bindings, where R_{type} is the set of roles of the sub-collaboration named $type$.*

(vi) *$m_{\mathrm{a-ac}} : N_{\mathrm{A}} \rightarrow AC \times CL$ is a non-injective function that maps active collaborations to activities and classifies the active collaboration's roles as* initiating *or* offered *roles within the context of the mapping (i.e. for the given activity). More formally, CL is a set of binary relations, such that if $m_{\mathrm{a-ac}}(n_{\mathrm{a}}) = (ac, cl)$, then $cl = \{(r, typ) : r$ is a role of the collaboration with name $ac.type$ and $typ \in \{INIT, OFF\}\}$.*

(vii) *$m_{\mathrm{enp-a}} : N_{\mathrm{EnP}} \rightarrow N_{\mathrm{A}}$ and $m_{\mathrm{exp-a}} : N_{\mathrm{ExP}} \rightarrow N_{\mathrm{A}}$ are functions mapping entry- and exit-points to activities.*

(viii) *$l_{\mathrm{ep-pred}} : (N_{\mathrm{EnP}} \cup N_{\mathrm{ExP}}) \rightarrow P$ is an injective function labeling each entry and exit-point of an activity with a state predicate of the activity's active collaboration.*

(ix) *$exp_{\mathrm{type}} : N_{\mathrm{ExP}} \rightarrow \{END, SUSPENSION\}$ is a function that classifies exit-points either as* end-of-execution *or as* suspension *ones.*

2.2 Semantics for Goal Sequences

Goal sequences are given a token-game semantics. Intuitively, when an activity receives an input token, its active collaboration is enabled. If the token is directly received from an edge (i.e. not via an entry-point), the active collaboration can begin execution from its initial state. Otherwise, if the token is received through an entry-point, the active collaboration can resume execution from the state represented by the event-goal labeling the entry-point. The active collaboration in reality begins or resumes its execution when one of its roles takes the appropriate initiative. Thereafter, it evolves until an interaction point with other collaborations is eventually reached. That is, the active collaboration runs until the predicate of one of its activity's exit-points holds. When this happens, the control token is passed on to the next activity or control node. According to this semantics, the intended behaviour of the *TransportService* collaboration, as specified by its goal sequence (Fig. 2), closely reflects the requirements. Initially the *BuyTicket* collaboration is started and thereafter suspended after the ticket is requested. At that point, a check is performed to determine if the vehicle is at the terminal (i.e. at *T1*). If the result is positive, *BuyTicket* is finished and *EnterVehicle* is enabled, followed by *VehDeparture*, *VehArrival* and *ExitVehicle*. If the vehicle was not at *T1*, this role initiates *ReqVehicle* to request the vehicle from *T2*. *VehDeparture* is then executed, followed by *VehArrival*, which allows *BuyTicket* to be resumed. Thereafter the service progresses as explained before.

Formal semantics for goal sequences is provided by mapping them into hierarchical coloured Petri-nets (HCPNs). The selection of HCPNs as the semantic domain has been mainly motivated by two facts. First, Petri-nets in general, and HCPNs in particular, have been extensively studied, and quite a number of quality tools exist that support and automate their analysis. Second, the mapping of goal sequences into HCPNs is rather intuitive, as will become clear later on. Due to space limitations we will assume that the reader is familiar with traditional Petri-nets and will only give a short introduction to (H)CPNs.

Coloured Petri-nets (CPNs) [4] extend traditional Petri-nets by associating a *colour* or data type with each token. In this way, tokens are distinguishable from each other, unlike in traditional Petri-nets. Places has also an associated data type (or *colour domain*) determining the kind of tokens they can contain. Transitions can modify the type and value of their output tokens. In addition, they can have an associated guard stating conditions over its input tokens, that must be satisfied for the transition to become enabled.

Definition 2 (CPN). *A non-hierarchical CPN is a tuple* $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ *[4] where* Σ *is a finite set of non-empty* types, P *is a finite set of* places, T *is a finite set of* transitions, A *is a finite set of* arcs, $N : A \rightarrow (P \times T) \cup (T \times P)$ *is a* node *function,* $C : P \rightarrow \Sigma$ *is a* colour *function,* G *is a* guard *function mapping boolean guards to transitions,* E *is an* arc expression *function labeling arcs, and* I *is an* initialisation *function for places.*

In a hierarchical CPN it is possible to define *substitution transitions*, which can be decomposed into so-called *subpages* (i.e. subnets). Each subpage has a

number of places called *port places*, through which the subpage communicates with its surroundings. The relationship between a substitution transition and its subpage is specified by describing a *port assignment*, which couples the port places of the subpage with the surrounding places, or so-called *socket places*, of the substitution transition. Port and socket places can be classified as input (i.e. accept tokens), output (i.e. deliver tokens) or I/O (i.e. both accept and deliver tokens) places.

Definition 3 (HCPN). *A hierarchical CPN is a tuple* $HCPN = (S, SN, SA, PN, PT, PA, FS, FT, PP)$ *where* S *is a finite set of* pages *(i.e. subnets),* SN *is a set of* substitution transitions, $SA : SN \rightarrow S$ *is a* page assignment *function,* PN *is a set of* port nodes, $PT : PN \rightarrow \{in,out,i/o, general\}$ *is a port type function,* PA *is a port assignment function mapping, for a given substitution transition, its sockets with its subnet's ports,* FS *is a finite set of* fusion sets, FT *is a* fusion type *function, and* PP *is a multi-set of* prime pages *[4].*

Informal Mapping. The main idea behind the mapping of goal sequences to HCPNs is to map the collaboration-uses of a goal sequence to substitution transitions, and decompose them into subnets describing the behaviour of those collaboration-uses.

Given a goal sequence describing the behaviour of a collaboration C (composed of a set of sub-collaborations), we map each collaboration-use of the goal sequence into a substitution transition. This means that several activities may be mapped into the same substitution transition, if they are annotated with the same collaboration-use (e.g. the two activities annotated with *b:BuyTicket* in Fig. 2 are mapped to the same substitution transition). The mapping of activities and their collaboration-uses is illustrated in Fig. 3b. Note that entry-points, as well as *suspension* exit-points of an activity are mapped into I/O socket places of the corresponding substitution transition, while *end-of-execution* exit-points are mapped into output socket places. Therefore, socket places represent event- and state-goals (i.e. the goals labeling the entry- and exit-points). In addition, an input socket is added, representing the starting point of the collaboration, as well as an *id* I/O socket, which is used to uniquely identify the specific collaboration-use the substitution transition represents. The colours used for socket places are `CTRL_ST` and `CTRL_STxDEP`, which are two custom defined data-types. `CTRL_ST` represents C's global state, and is composed of the individual states of C's sub-collaborations. `CTRL_STxDEP` is a Cartesian product of `CTRL_ST` and `DEP`. The latter is an enumeration with two values: `depUnres` (for dependency unresolved) and `depResolved` (for dependency resolved). The `CTRL_STxDEP` type is used to cope with suspend-resume dependencies, which require sub-collaborations to give away the control token while in the middle of execution (i.e. at *suspension* exit-points in the goal sequence). To enforce this behaviour, all tokens leaving I/O socket places (except the *id* one) must be marked with `depUnres`, while all arriving tokens must be marked with `depResolved`.

The initial node, as well as the final and merge nodes of the goal sequence are mapped into places, while join and fork nodes are mapped into normal

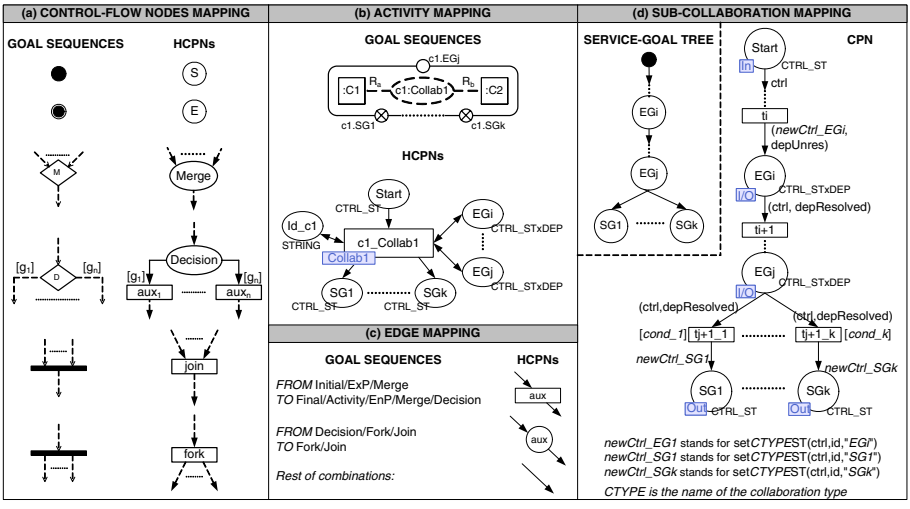

Fig. 3. Mapping of goal sequence elements to HCPN elements

transitions. The mapping of a decision node yields a place interconnected to as many transitions as the node has outgoing edges. The guards of these edges are then assigned to the transitions. Edges become net arcs, possibly with auxiliary transitions or places so as to respect the bipartite nature of Petri nets. All these mappings are summarized in Figs. 3a and 3c.

The translation of activities, edges and control-flow nodes, we have just explained, yields the main net of the final HCPN. For the mapping to be complete, we need to describe the decomposition of substitution transitions into subnets. These subnets will describe the behaviour of the goal sequence's collaboration-uses that the substitution transitions represent. As the collaboration-uses of the goal sequence (e.g. *va1:VehArrival* in Fig. 2) are occurrences of the sub-collaborations of C (e.g. *VehArrival* in Fig. 1a), the subnets will describe the behaviour of those sub-collaborations. Several substitution transitions may be assigned the same subnet, if they represent collaboration-uses of the same type (i.e. occurrences of the same sub-collaboration of C).

We are not interested on subnets describing detailed behaviour, but rather aim at high-level, abstract behaviour descriptions. Service goal trees ($SGTs$) provide such descriptions, so we use them as input for the mapping of sub-collaborations into subnets (see Fig. 3d). The SGT nodes are translated into net places, and the SGT arcs into net arcs plus an auxiliary transition. Places are characterized as port places: the *Start* place becomes an input port, places representing event-goals (EG) become I/O (i.e. bidirectional) ports, and those representing state-goals (SG) become output ports. Then, when coupling the subnet's ports with the sockets of a substitution transition, those ports and sockets representing the same goal are interconnected. The *Start* place, as well as those places representing state-goals are typed with the `CTRL_ST` colour, while the `CTRL_STxDEP`

colour is used for places representing event-goals. Custom defined functions are used to modify the state of the collaboration (represented by `CTRL_ST`) as the control token travels form the *Start* input port to the output port(s). At each point in time the value of the token reflects the place the token has reached, thus reflecting the event-/state-goal that has been achieved. In addition, all tokens arriving at an I/O port are marked with `depUnres`, while all tokens leaving an I/O port are marked with `depResolved`. This ensures that the control token leaves the net at I/O ports, in order to satisfy suspend-resume dependencies.

All transitions of the subnet will be unguarded, except possibly those leading to output ports (i.e. places representing state-goals). If several transitions lead to different output ports from the same place, as illustrated in Fig. 3d, guards may be imposed on those transitions if a deterministic choice is wanted. These guards would determine the conditions to achieve each of the goals. They can be constructed from the information provided by the goal sequence, since the latter describes the relationships between sub-collaboration goals (i.e. it tells us the goal that a sub-collaboration must achieve in order for another sub-collaboration to achieve its own goal). The process to determine these guards is explained in the next section.

Figure 4a partially shows the HCPN resulting from the mapping of the *TransportService*'s goal sequence. Each one of the collaboration-uses in Fig. 2 has been mapped to a substitution transition. Note that the two activities referring to *b:BuyTicket* correspond now to a single substitution transition (i.e. *b_BuyTicket*). This substitution transition has one I/O socket (i.e. *b_ticketReqed*) representing both the suspension exit-point of the first activity and the entry-point of the second activity to the left in Fig. 2. Figure 4b depicts the subnet describing the behaviour of *BuyTicket*. This is the subnet assigned to the *b_BuyTicket* substitution transition, and closely resembles the service-goal tree in Fig. 1c.

Formal Semantics. For the mapping of goal sequences, we define two semantic functions: $[\![\_]\!]_{\text{CPN}}$, which maps elementary sub-collaborations into non-hierarchical CPNs; and $[\![\_]\!]_{\text{HCPN}}$, which maps collaboration goal sequences into HCPNs. $[\![\_]\!]_{\text{CPN}}$ takes a service-goal tree and a collaboration goal sequence, and returns a *CPN* representing the collaboration whose goals are described by the service-goal tree. A service-goal tree is defined as:

Definition 4 (Service-goal tree). *A service-goal tree is a directed graph* $SGT = (cId, GN, GA)$ *where: cId is the name of the collaboration whose goals SGT describes;* $GN = \{start\} \cup EG \cup SG$ *is a set of nodes, with start being the initial node, EG being a set of intermediary nodes representing event-goals, and SG being a set of final nodes representing sate-goals; and* $GA \subseteq N \times N$ *is a set of directed arcs between nodes, such that* $\forall (s,t) \in GA : [s \notin SG \wedge t \neq start]$.

According to the mapping explained in the previous section, and given $SGT = (cId, GN, GA)$ and $GS = (N, E, g_{\text{d}}, m_{\text{exp-a}}, R_{\text{GS}}, AC, m_{\text{a-ac}}, m_{\text{enp-a}}, l_{\text{ep-pred}}, exp_{\text{type}})$, we define $[\![SGT, GS]\!]_{\text{CPN}} = (\Sigma, P, T, A, N, C, G, E, I)$, where:

$$\Sigma = \{CTRL\_ST,\ CTRL\_ST{\times}DEP,\ STRING\}$$

$$P = GN \cup \{Id\} \qquad T = \{t_{ga} : ga \in GA\}$$

$$A = \{source\text{TO}t_{ga}, t_{ga}\text{TO}target : ga = (source, target) \in GA\}$$
$$\cup \{Id\text{TO}t_{ga}, t_{ga}\text{TO}Id : Id \in P, ga \in GA\}$$

$$N(a) = (source, target),\ \text{if } a \text{ is in the form } source\text{TO}target$$

$$C(p) = \begin{cases} CTRL\_ST, & \text{if } p \in SG \cup \{start\} \\ CTRL\_ST{\times}DEP, & \text{if } p \in EG \\ STRING, & \text{if } p = Id \end{cases}$$

$$E(a) = \begin{cases} ctrl, & \text{if } (a = s\_t_{ga}) \wedge (s = start) \\ (set\text{CTYPEn}ST(ctrl,id,\text{"tgn"}), depUnres), & \text{if } (a = t_{ga}\_t) \wedge (t \in EG) \\ (ctrl, depResolved), & \text{if } (a = s\_t_{ga}) \wedge (s \in EG) \\ set\text{CTYPEn}ST(ctrl,id,\text{"t"}), & \text{if } (a = t_{ga}\_t) \wedge (t \in SG) \\ iId, & \text{if } [(a = s\_t_{ga}) \wedge (s = Id)] \vee [(a = t_{ga}\_t) \wedge (t = Id)] \end{cases}$$

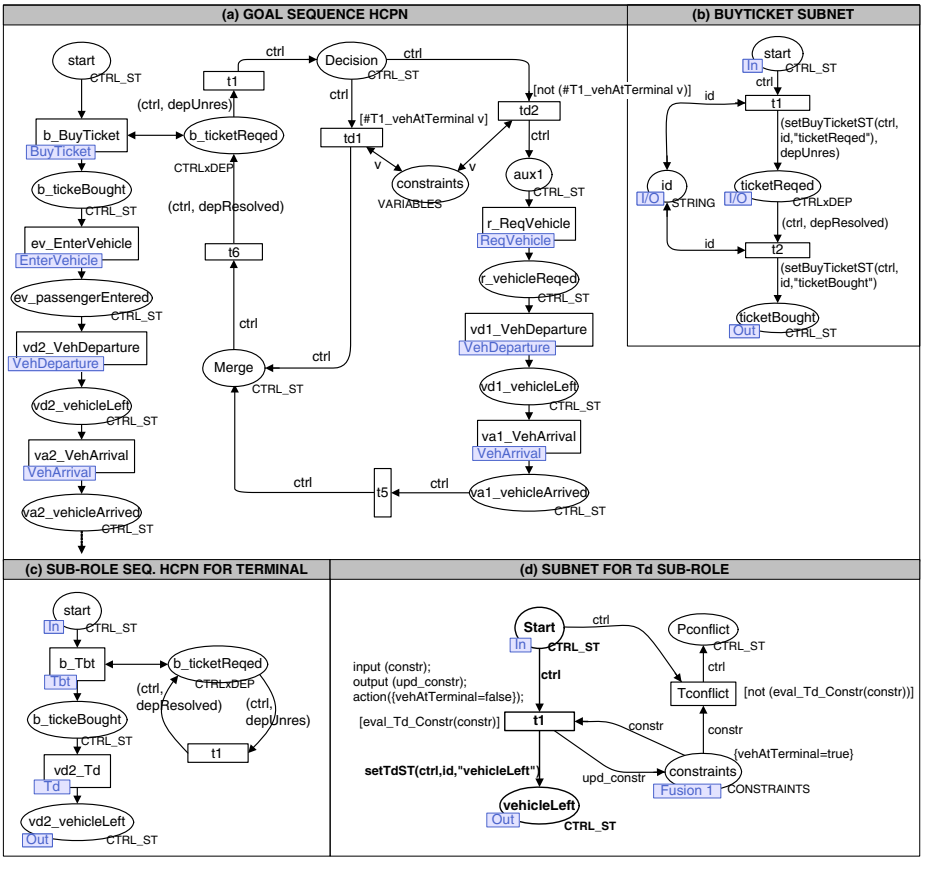

Fig. 4. Nets for the *TransportService* case study

No initial marking (I) is defined for the resulting CPN. The guard function G assigns *true* to all transitions of the CPN, except possibly to those leading to places $p \in SG$. To describe how the guards are assigned to those transitions, let us consider the example net in Fig. 3d. To determine the set of conditions *cond_i*, we just need to search for entry-points in the goal sequence that are labeled with EG_j. For each entry-point, if its associated activity A has several exit-points, then the conditions are set to *true* (i.e. representing a non-deterministic choice). Otherwise, if the state-goal labeling A's exit-point is SG_n, then *cond_n* is set to the value of the goal labeling the exit-point of the activity immediately preceding A. Note that *cond_n* may actually be a boolean expression of goals, if several activities lead to A through a control flow node.

As a convention, in the following we will use the notation $T.E$, meaning element E of tuple T, in order to access the elements of a tuple. We can now define $[\![\_]\!]_{\mathrm{HCPN}}$, which takes a goal sequence $GS = (N, exp_{type}, m_{enp-a}, m_{exp-a}, R_{\mathrm{GS}}, AC, m_{a-ac}, l_{ep-pred}, E, g_d)$ and a set of service-goal trees $\{SGT_{ac} = (cId, GN, GA), SGT_{ac}.cId = ac.type, ac \in GS.AC\}$ (i.e. one for each sub-collaboration referred to by GS), and returns a $HCPN = (S, SN, SA, PN, PT, PA, FS, FT, PP)$. We start by introducing the set of subnets (S_{AC}), the set of transitions due to the mapping of decision nodes and edges (T_{D} and T_{edges}), the set of places due to the mapping of arcs (P_{edges}), the set of arcs connecting *id* places to substitution transitions (A_{Id}), and the set of arcs connecting the *constraint* place to transitions generated by decision nodes (A_{constr}) as:

$$
\begin{aligned}
S_{\mathrm{AC}} &= \{[\![(SGT_{ac}, GS)]\!]_{CPN} : SGT_{ac}.cId = ac.type, ac \in AC\} \\
T_{\mathrm{D}} &= \{t_d : d \in N_{\mathrm{D}}, (d, \_) \in E\} \\
T_{\mathrm{edges}} &= \{t_{(es,et)} : (es, et) \in E, es \in \{n_0\} \cup N_{\mathrm{M}} \cup N_{\mathrm{ExP}}, \\
&\qquad et \in N_{\mathrm{FI}} \cup N_{\mathrm{A}} \cup N_{\mathrm{EnP}} \cup N_{\mathrm{M}} \cup N_{\mathrm{D}}\} \\
P_{\mathrm{edges}} &= \{p_{(es,et)} : es \in T_D \cup N_F \cup N_J, et \in N_F \cup N_J, (es, et) \in E\} \\
A_{\mathrm{Id}} &= \{ac.id\_Id \mathrm{TO} ac.id\_ac.type, ac.id\_ac.type \mathrm{TO} ac.id\_Id : ac \in AC\} \\
A_{\mathrm{constr}} &= \{constr \mathrm{TO} t_d, t_d \mathrm{TO} constr : constr \in P, t_d \in T_D\}
\end{aligned}
$$

Now we define the main net (s_{main}) describing the interconnection of substitution transitions (representing collaboration uses of the goal sequence). This net is a $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ described as:

$$
\begin{aligned}
\Sigma &= \{CTRL\_ST, CTRL\_ST \times DEP, STRING, VARIABLES\} \\
P &= \{n_0\} \cup N_{FI} \cup N_D \cup N_M \cup \{constr\} \cup \{ac.id\_Id : ac \in AC\} \\
&\quad \cup \{ac.id\_p_{s_{ac}} : p_{s_{ac}} \in P_{s_{ac}}, s_{ac} \in S_{AC}\} \cup P_{\mathrm{edges}} \\
T &= \{ac.id\_ac.type : ac \in AC\} \cup T_D \cup T_{\mathrm{edges}} \\
A &= A_{Id} \cup A_{constr} \cup \{es \mathrm{TO} t_{(es,et)}, t_{(es,et)} \mathrm{TO} et : t_{(es,et)} \in T_{\mathrm{edges}}\} \\
&\quad \cup \{es \mathrm{TO} p_{(es,et)}, p_{(es,et)} \mathrm{TO} et : p_{(es,et)} \in P_{\mathrm{edges}}\} \\
&\quad \cup \{es \mathrm{TO} et : (es, et) \in E, t_{(es,et)} \notin T_{\mathrm{edges}}, p_{(es,et)} \notin P_{\mathrm{edges}}\}
\end{aligned}
$$

$$N(a) = (source, target),\ \text{if } a \text{ is in the form } source\text{TO}target$$

$$C(p) = \begin{cases} CTRL\_ST,\ \text{if } p \in \{n_0\} \cup N_{FI} \cup N_D \cup N_M \cup P_{arcs} \\ STRING,\ \text{if } p \text{ is in the form } ac.id\_Id \\ VARIABLES,\ \text{if } p = constr \\ C(p'),\ \text{if } p \text{ is a socket connected to port } p\text{'} \end{cases}$$

$$E(a) = \begin{cases} (ctrl, depUnres),\ \text{if } a{=}source\text{TO}target \\ \quad \text{and } source \text{ is a socket connected to an i/o port} \\ (ctrl, depResolved),\ \text{if } a{=}source\text{TO}target \\ \quad \text{and } target \text{ is a socket connected to an i/o port} \\ varbl,\ \text{if } a \in A_{constr} \\ id,\ \text{if } a \in A_{Id} \\ ctrl,\ \text{otherwise} \end{cases}$$

$$G(t) = \begin{cases} g_D(e),\ \text{if } t = t_d \in T_D, e = (d, \_) \in E, d \in N_D \\ true,\ \text{otherwise} \end{cases}$$

The initialisation function (I) of s_{main} assigns to the starting place (i.e. $p = n_0$) a token of type `CTRL_ST`. This token describes the initial state of the composite collaboration described by GS, where all state predicates representing the goals of the collaboration are set to *false*.

Finally, we define $[\![GS, \{SGT_{ac}\}]\!]_{\text{HCPN}} = (S, SN, SA, PN, PT, PA, FS, FT, PP)$, where:

$$S = \{s_{\text{main}}\} \cup S_{\text{AC}} \qquad SN = \{ac.id\_ac.type : ac \in AC\}$$

$$PN = \bigcup_{s_{ac} \in S_{AC}} P_{s_{ac}} \qquad FS = \emptyset \qquad PP = \{s_{\text{main}}\}$$

$$SA(t) = s_{ac},\ \text{if } t = ac.id\_ac.type, ac \in AC, s_{ac} \in S_{\text{AC}}$$

$$PT(p) = \begin{cases} in,\ \text{if } p = start \\ out,\ \text{if } p \in GT_{ac}.SG \\ i/o,\ \text{if } p \in GT_{ac}.EG \cup \{Id\} \end{cases}, \forall p \in P_{s_{ac}}, \forall s_{ac} = [\![(SGT_{ac}, GS)]\!]_{\text{CPN}} \in S_{\text{AC}}$$

$$PA(t) = (ac.id\_p, p), \forall p \in PN, \forall ac.id\_p \in P_{s_{main}}$$

3 Detection of Implied Scenarios

A goal sequence describes the intended behaviour of a service from a global perspective, and can be used to synthesize state-machines for the service-roles. The actual service behaviour is performed by the components playing those roles. Since components only have a local view of the service, unexpected interactions may arise. These are the so-called implied scenarios [1], which correspond to service behaviour that has not been explicitly specified, but follows implicitly, and will be present in any implementation of the service. An implied scenario may capture some overlooked positive behaviour, but it may also represent undesired behaviour. Detecting implied scenarios is therefore important.

In the context of the collaboration-based service specification approach treated here, an implied scenario may arise due to the existence of multiple initiatives,

from the service-roles, to engage in sub-collaborations. In the collaboration goal sequence these initiatives are ordered in some desired sequence. However, this ordering may not be guaranteed at runtime due to the independence between the initiatives of different service-roles. Therefore, all possible orderings should be analyzed in order to determine if undesired behaviours may arise. Fortunately, this can be done without performing a global analysis of the service collaboration. It suffices to analyse, separately, the sub-role sequences that each service-role may execute. These sub-role sequences can be obtained from the collaboration goal sequence. For example, the following sub-role sequence: $V_{\text{ev}} \rightarrow V_{\text{d}} \rightarrow V_{\text{a}} \rightarrow V_{\text{exv}}$; can be extracted from the goal sequence in Fig. 2 for the V service-role.

Separate sub-role sequences are extracted for each (instance of a) service-role (e.g. *T1:T*, *T2:T*, ...). This can be done by invoking the *VISIT* algorithm (see Algorithm 1), with $i = 0$ and $n = n_0$, for each service-role (*rType*), and for each instance of that role (*rIns*). This algorithm traverses the goal sequence's graph (*GSG*) with a depth-first search method, looking for occurrences of *rIns*. While traversing the *GSG* forwards, the algorithm creates a role-sequence graph (*RSG*) that includes only those activities (and their associated entry-/exit-points) *related* to *rIns*. *RSGs* have the same syntax and semantics as *GSGs*. If a fork node is found, the algorithm adds it to the *RSG* and continues the search through one of the fork's outgoing edges. When a decision node is found, one of its outgoing edges is also chosen to continue the search, but the decision node is not added to the *RSG* (since at runtime only one of the branches can be executed). Instead, different *RSGs* will be generated for each of the decision's branches (e.g. in our case study two *RSGs* are generated for *T1:T*, one for *vehAtTerminal* and other for *NOT vehAtTerminal*). In order to know the decision node's branch a *RSG* corresponds to, the branch's guard is saved in a dedicated table (*decisions*). Once a final node is found, a sub-role sequence has been obtained. From there, a copy of the *RSG* is done and the algorithm begins the backtracking phase. During this phase the previously added nodes are removed from the *RSG* until a decision or fork node with unvisited edges is found. If this happens, one of the unvisited edges is selected and the *GSG* is again traversed forwards (so new nodes are added to the *RSG*). Otherwise, if the initial node is reached during backtracking, the extraction process ends. Note that if fork (resp. join) nodes where found while traversing the *GSG*, the generated *RGSs* describe a path through only one of the outgoing (resp. incoming) edges of these nodes. The individual *RGSs* sharing fork (resp. join) nodes must therefore be merged at the end. To help in this process, each time a fork (resp. join) node is found, information about the traversed edge is saved in a dedicated table (*forks*; resp. *joins*). Note also that loops are traversed only once (i.e. only one iteration is performed). This is achieved by annotating in a table (*visited*) the number of times each node is visited. With this restriction we avoid infinite role sequences, while we ensure that all possible non-repetitive sequences of sub-roles are considered.

Once the sub-role sequences have been obtained, their analysis can start. For each service-role, its sub-role sequences are first analysed individually, and thereafter their interactions are studied. In the individual analysis, we look for any

Algorithm 1. VISIT(GSG,n,$rIns$,$RSG[rIns,i]$)

```
// All variables except adjNodes and nextN are global
// All elements of the visited array are initialized to 0 before first call to VISIT
visited[n]++; adjNodes[n] = GETADJACENTNODES(n, GSG)
while adjNodes[n] ≠ ∅ do
  nextN = adjNodes[n].pop()
  if visited[nextN] < 2 then
    if ((n ∈ N_EnP) ∨ (n ∈ N_ExP) ∨ (n ∈ N_A)) ∧ RELATED(n, rIns) then
      ADDTOGRAPH(n, RSG[rIns, i])
    else if (n ∈ N_F) ∨ (n ∈ N_J) then
      ADDTOGRAPH(n, RSG[rIns, i]) and update forks[rIns, i]/joins[rIns, i]
    else if n ∈ N_D then update decisions[rIns, i]
    else if n = n_0 then ADDTOGRAPH(n, RSG[rIns, i])
    end if
    VISIT(GSG, nextN, rIns, RSG[rIns, i])
end while //There are no (more) adjacent nodes
visited[n] = visited[n] − 1
if n ∈ N_FI then //Final node
  ADDTOGRAPH(n, RSG[rIns, i]); RSG[rIns, i + 1] = RSG[rIns, i]; i++
end if
REMOVEFROMGRAPH(n, RSG[rIns, i]) //Backtracking
```

set of two or more consecutive offered sub-roles (i.e. offered sub-roles connected by edges and/or join/fork nodes) that the sequence may contain. Consecutive offered sub-roles may represent a conflict, if they are played in collaborations with different parties, and these collaborations maintain some kind of dependency (e.g. one of them should not finish before the other does). In that case, the dependency might be violated, since the initiatives to start the collaborations are taken by different parties. In the *TransportService* example this happens for the V service-role. According to their sub-role sequences, V_{ev} is to be played in *EnterVehicle* before V_d in *VehDeparture* (see Fig. 2). However there is no way for T, which takes the initiative in *VehDeparture*, to know if *Passenger (P)* has taken the initiative to start *EnterVehicle*, and when this has finished (i.e. the condition *ev.passEntered* is not visible for *Terminal*). Thus T may request V to play V_d before P has requested it to play V_{ev}.

After the individual analysis, we study how the sub-role sequences of a single service-role interact with each other, if executed concurrently. Intuitively, we first constrain the execution of sub-roles by imposing pre- and post-conditions, and then build the cross-product of the sub-role sequences to detect constraint conflicts. For that purpose, sub-role sequences are semantically mapped into HCPNs. This mapping follows the same guidelines as the goal sequence mapping detailed in Sect. 2.2, the only difference being substitution transitions labeled with sub-role names, rather than with active collaboration names. As an example, consider Fig. 4c, which depicts the HCPN for the sub-role sequence obtained when the *TransportService*'s goal sequence is projected onto *T1:T* and *T1.vehAtTerminal* is *true*. Figure 4d presents the subnet representing role T_d (part in boldface).

The execution constraints (i.e pre- and post-conditions) to be imposed on sub-roles follow from the requirements and the service domain. For example, in our case study we can further restrict the execution of role T_d (from *VehDeparture*) by setting *VehAtTerminal* and *NOT VehAtTerminal* as part of T_d's pre- and post-condition, respectively. In our HCPN model constraints are represented as boolean tokens that reside in a place shared by all the sub-role sequence nets. Since HCPNs do not allow guards to be imposed on substitution transitions (which, remember, represent sub-roles), the pre-condition for the execution of a sub-role is instead specified as a guard on the first transition of the subnet describing the sub-role behaviour. If the guard is satisfied, the transition fires and it updates the value of the constraints according to the post-condition. This is illustrated in Fig. 4d for the T_d sub-role, where the result of calling function *eval_Td_Constr(constr)* has been imposed as guard of transition *t1*. This function processes the value of the *constr* token, which represents the constraints, and returns *true* if *VehAtTerminal* is *true*. The value of *VehAtTerminal* is updated when *t1* fires, by its code segment. Note that in addition to the *constraints* place, a *Tconflict* transition and a *Pconflict* place have been added to the subnet of T_d. Note also that *Tconflict* can only be fired when *t1* can not, that is, when *VehAtTerminal* is *false*. In such a case, *Tconflict* "steals" the tokens from the *Start* and *constraints* places forcing a dead-marking to be reached. This behaviour reflects our desire of a (potential) conflict to be reported if a sub-role cannot be immediately executed when it receives the control token, because its pre-condition is not satisfied.

At the end, the sub-role sequences are composed in parallel and the reachability graph of the resulting net is constructed and analysed in search of dead-markings, which would represent potential conflicts. In order to test our analysis method, we used CPN Tools [3] to analyse an extended version of the *Transport-Service* (with a control center for mediation between the terminals). A reachability graph with 37 nodes and 58 arcs was generated for the analysis of the sub-role sequences of the *Terminal* (T) service-role. This analysis revealed two implied scenarios: a passenger may miss the vehicle after buying the ticket, if the vehicle is dispatched following a request from the control center; or the vehicle may depart with the passenger before a control center's request has been completely processed. A reachability graph of similar size was generated for the *Vehicle* (V) service-role. As a comparison, the detection method by Uchitel et al. [12], which is of exponential complexity with the number of service-roles, needs to build a safety property for the same case study of 4414 states, if heuristics are used. Although no formal conclusions can be obtained from this comparison, we believe the results show the potential of our approach.

4 Related Work

Service-oriented specification has been addressed in several works. Rößler et al. [8] suggested collaboration based design with a tighter integration between interaction and state diagram models, and created a specific language, CoSDL, to

define collaborations. CoSDL is inspired by SDL, so it fails at providing the cross-cutting service composition offered by UML collaborations and goal sequences. Krüger et al. [5] propose an approach to service engineering that has many commonalities with our own. They consider, as we do, services as collaborations between roles played by components, and use a combination of Use Cases and an extended MSC language to describe them. Liveness is expressed by means of the operators provided by their MSC language, while service structure and role binding are described with, so-called, role and deployment domain models. In our approach UML collaboration diagrams are used to provide a unified way of describing service structure and role bindings, and to provide a framework for expressing liveness with goal sequences. Goal sequences provide interesting opportunities for analysis, as we have discussed.

The concept of implied scenarios was first introduced by Alur et al. in [1], where they presented an algorithm to detect this kind of scenarios from MSC specifications. This work was later extended by Uchitel et al. [12], who proposed an approach for the incremental specification (using both MSCs and HMSCs) of systems, driven by the detection of implied scenarios. The main drawback of Uchitel et al.'s work is, however, the state explosion problem (although they limit it by applying heuristics). Munccini has proposed an approach for the detection of implied scenarios based on the analysis of HMSCs [6]. His work builds over a previous work of Uchitel et al., and avoids the state explosion problem. Our method also limits the state explosion problem and it is applicable to UML collaboration-based specifications, while Munccini's approach applies to HMSC-based specifications.

5 Discussion and Conclusions

UML 2.0 collaborations provide very useful structuring mechanisms for specifying cross-cutting service behaviours. They enable: (a) an attractive structured overview; (b) structural decomposition into features, by means of collaboration-uses; (c) re-usability; and (d) definition of semantic interfaces for dynamic discovery, binding and compatibility checks [10]. Still, a proper way to describe the choreography or joint behaviour of the sub-collaborations of a composite collaboration is needed. Collaboration goal sequences can be used to fill this gap. They help to understand and document the relationships and execution dependencies between sub-collaborations, in terms of their goals. Moreover, they can be analysed in order to detect inconsistencies and implied scenarios at an early stage of service specification.

Formal semantics for goal sequences based on hierarchical coloured Petri-nets has been presented here that allows their automated analysis using general purpose tools available for HCPNs. The detection of implied scenarios is done in two phases. First, sub-role sequences are extracted from the goal sequence and individually analysed. Then the cross-product of the sub-role sequences of each service-role is built to examine how they interact. The proposed analysis suffers little from the state explosion problem since the sub-role sequences of each

service-role are analysed separately, so the complexity is linear with the number of service-roles. In addition, the analysis is done at a high-level of abstraction (i.e. with role sequences and not message sequences). The proposed implied scenario detection approach demonstrates, in addition, that we have much to gain from the explicit description of features dependencies, and from the analysis and understanding of concurrency on interfaces.

Although we can use HCPN-tools for the analysis of goal sequences, their mapping into HCPNs is still performed manually. Thus, a short-term objective is to provide tool support for the mapping, so the whole process can be automatized. Another interesting issue we plan to work on is how to address the elimination of the implied scenarios. One possibility might be to specify negative goal sequences (as the the negative scenarios in [12]).

Acknowledgements

We would like to thank Gregor von Bochmann, Cyril Carrez and the anonymous reviewers for their valuable comments on this work.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. In: 22nd Intl. Conf. on Software Engineering (ICSE'00). (2000) 304–313
2. Castejón, H.N., Bræk, R.: A collaboration-based approach to service specification and detection of implied scenarios. In: ICSE's 5th Intl. Workshop on Scenarios and State Machines: models, algorithms and tools (SCESM'06), ACM Press (2006)
3. CPN Group: CPN Tools Manual. Technical report, Univ. of Aarhus, Denmark (2005) available at http://wiki.daimi.au.dk/cpntools/cpntools.wiki.
4. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1. Springer-Verlag (1997)
5. Krüger, I.H., Gupta, D., Mathew, R., Moorthy, P., Phillips, W., Rittmann, S., Ahluwalia, J.: Towards a process and tool-chain for service-oriented automotive software engineering. In: ICSE'04 Workshop on Software Engineering for Automotive Systems (SEAS). (2004)
6. Muccini, H.: Detecting implied scenarios analyzing non-local branching choices. In: 6th Intl. Conf. of Fundamental Approaches to Software Engineering (FASE'03). LNCS 2621. (2003) 372–386
7. Object Management Group: UML 2.0 Superstructure Specification. (2005)
8. Rößler, F., Geppert, B., Gotzhein, R.: Collaboration-based design of SDL systems. In: 10th SDL Forum. LNCS 2078 (2001) 72–89
9. Sanders, R.T., Bræk, R.: Modeling peer-to-peer service goals in UML. In: 2nd IEEE Intl. Conf. on Software Engineering and Formal Methods (SEFM'04). (2004)
10. Sanders, R.T., Bræk, R., von Bochmann, G., Amyot, D.: Service discovery and component reuse with semantic interfaces. In: 12th SDL Forum. LNCS 3530 (2005)
11. Sanders, R.T., Castejón, H.N., Kraemer, F.A., Bræk, R.: Using UML 2.0 collaborations for compositional service specification. In: ACM/IEEE 8th Intl. Conf. on Model Driven Engineering Languages and Systems (MoDELS). LNCS 3713 (2005)
12. Uchitel, S., Kramer, J., Magee, J.: Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. ACM TOSEM **13** (2004)