Formalizing uncertainty in service request/offer description and matching

Tommaso Bolognesi[1]

CNR - Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo",
tommaso.bolognesi@isti.cnr.it,
WWW home page: http:/www1.isti.cnr.it/ bolognesi/

Abstract. This paper discusses the problem of describing and matching service requests and offers, that appears as central and characteristic in the context of the Service Oriented Computing (SOC) paradigm. Service descriptions, be they requests or offers, are seen as a mix of detailed, local state information, and fuzzy information, or speculation, about remote partners. Service fruition is represented, at a very high abstraction level, as an atomic interaction among two or more parties, each providing a partial view about a global state change. We propose here four formal solutions of increasing complexity; the first three of them are conveniently formalized in TLA+ and illustrated by simple examples. The use of a logic-based approach should favor the integration with current Semantic Web technologies, and with Description Logics, and at the same time it should help reducing the gap between formal and informal (natural language) descriptions of services. [1]

1 Introduction

One of the most characteristic and challenging features of the Service Oriented Computing (SOC) paradigm [6] is related to the idea of run-time, automatic or semi-automatic search, discovery and binding of services. The designer of a complex system, or, more appropriately, of a complex functionality, or *service*, is not expected to know in advance and in detail all the services that a given network infrastructure might offer, as sub-components, for achieving his or her goals; some of these services may not even exist at design time, while becoming available at run-time. Part of the job of software SOC-based system designers is therefore that of describing, at appropriate abstraction levels, service offers and requests, and devising successful criteria for discovering and matching them, without assuming a complete knowledge of the players, the rules, and the locations of the game. Of course SOC computing and its supporting technologies are not being created from scratch, both for the need to exploit existing (legacy) services and technologies, and because of the multiplicity and diversity of the

[1] Work partly supported by the SENSORIA European IST-Integrated Project - contract n. 016004

involved contributors on a global scale: it will be the result of a complex, *evolutionary* process based on competition, selection, agreement, standardization, similar to the one that has led to the WWW we have today.

In the software engineering of traditional distributed systems the distinction between the roles of system developers and users is quite clear, and so is the difference between the associated technical backgrounds. With SOC-computing such a distinction is blurred, and a wider range of actors with varying degrees of technical skills is likely to be involved. In particular, it appears nice and desirable to offer to non-technical people the possibility to play some autonomous role in the service offer/request game. One may imagine a scenario in which final user Bob, connected by a home PC or handheld device, assembles a relatively complex request and send it out in some 'infosphere', via an electronic portal whose address is the only explicit information available to him; in another scenario, employee Mary creates, publishes and advertises in an elecronic e-commerce mall the latest service offer from her company. Neither Bob nor Mary need to be familiar with programming or formal specification languages.

On the other hand the assessment and comparison of various approaches to the problems of service description, discovery, matching, does of course greatly benefit from some formal treatment, for supporting both the analysis and the implementation of the proposed solutions. In fact, the adoption of a formal specification language of sufficient simplicity and general applicability may help even in *thinking* about the rather intricate aspects of SOC, and in sorting out the various issues and needs that emerge in this new computing paradigm.

In light of the remarks above, the work presented here has been inspired by the following general requirements/objectives:

1. Keep compatible with the approaches and technologies of the Web Service and Semantic Web efforts [8]. A service involves the exchange of objects of which 'semantic' descriptions are available: they are resources with properties that can be declared and investigated, as in RDF [http://www.w3.org/RDF/]. On a long term, one might try to integrate some high level, logic based specification language (such as TLA [7]), for describing the dynamics of a service, and Description Logics [2], or some subset or derivation from it, for describing the properties of the objects manipulated by the service. (A combination of Description Logic and logic programs is indeed already proposed in [3].)
2. Recognize the ultimate similarity between service offers and service requests, by observing that either description may in general include both precise knowledge about a local environment and uncertain information about potential remote partners. We may therefore neutrally talk about service *descriptions* that match one another.
3. Provide a formalization of service (request/offer) descriptions that is simple enough to be easily converted into formulations in (restricted forms of) natural language, in view of usability by a wide range of possibly non technical people.
4. Provide a formalization of the service offer/request matching rule.

In Section 2 we provide a list of more specific choices that have guided our approach to the abstract, formal description of service offers/requests, and to the associated matching rules. In Section 3 we provide three formal solutions, using TLA+, and compare them; they open the way to a further, more elaborate formal solution, presented in Section 4. In Section 5 we compare our solutions with existing work, and identify items for further research.

2 Further requirements for service description and manipulation

In addition to the general requirements identified above, we have adopted the following more specific design choices.

Atomicity. At the highest abstraction level, it seems appropriate, and possibly also convenient w.r.t. conversion to formulation in natural language, to describe a service offer or request simply as a change of state of (a part of) the world, that takes place in one atomic step. At this level a service could then be seen as a binary relation between global states.

Object-based architecture. Service requestors and providers ('agents') are objects – instances of classes – equipped with a local state – a set of local variables, or attributes – and a set of operations. Each agent shall have one synchronization operation for each type of service it offers or requests, beside other local operations that are inessential for our investigation. The exchanged data items are also represented by objects, whose properties are declared and/or inspected by the involved partners.

Multi-party synchronization and distributed state updating. The fruition of a service corresponds, in general, to a multi-party synchronization among operations of a set of distributed agents. Each agent participates in the interaction by executing one of its operations, that involves preconditions and postconditions on the local state of the agent.

Synchronization boards and objects. The synchronization occurs via a synchronization 'board', which is a place where each agent can virtually post a set of synchronization object descriptions; these objects may of course be related to the local state of the agent, via pre and post-conditions. Agents (and their operations) are aware of the existence of boards, but are uncertain about the precise nature of the needs of potential partners, that is, on the alternative sets of objects that may or must be made available on the board for the service to come to fruition. Thus in their synchronization operations they express partial views on these sets of objects.

Lamport's TLA, and its derived TLA+ specification language [7], provide a useful expressive tool for giving formal substance to the ideas above, and for exploring alternative solutions. In extreme synthesis, TLA+ is first order predicate logic and set theory enriched with some temporal logic operators and some convenient and fairly standard constructs for modularizing large specifications.

The above requirement on atomicity is conveniently supported by the notion of *action* in TLA+ (an action is a logic formula that includes unprimed as well as primed variables, where the latter denote changed values of state variables, as in several specification notations; and an action may or may not be satisfied by a pair of adjacent states, called a *step*.) And, although the TLA+ definition does not make explicit reference to object-oriented or object-based concepts, one can establish rather obvious analogies between a TLA+ 'module' and a class, a TLA+ module instance and an object, and between a TLA+ and an O-O operator.

The precise ways in which the features of TLA+ can represent the above ideas on multi-party synchronization and distributed state updating are introduced in the next section.

3 Service offer/request description and matching in TLA+

In this section we provide specifications in TLA+ that illustrate three increasingly complex formalizations of the design choices discussed above. Familiarity with TLA+ is beneficial, but the plain logic nature of this language should allow even the uninitiated reader to follow the examples, which are also commented in the text following each formal fragment. In illustrating our TLA+ solutions we shall refer to an example involving three agents.

3.1 Three agents exchanging files

Three agents interact for exchanging files. As discussed earlier, it its not important to assign to agents the precise and mutually exclusive roles of service provider or requestor. Each agent in general shall be able, with respect to some service, both to offer and to request data. For example, a 'provider' may offer an audio file, and request a payment, and a 'requestor' may offer a payment, and request an audio file. At the level of our formalization, agents are all of the same type.

For simplicity, each agent has only one local variable called S, which is a set of bit triples, and can interact with the other agents by requiring and offering bit triples. S can be seen as a simple model of a library of files, and a bit triple is then an extremely simple model for a file with three (ordered and unnamed) fields. The idea is that agents may interact, and let the service come to fruition, by expressing constraints on the requested and offered items, which refer to the fields (properties) of these *interaction* objects. In perspective, these characterizations of requested and offered objects should be based on some formalized representation of their semantics, that is, on ontologies, possibly using some form of description logic.

As another simplification, in the formalizations we shall only use one, default synchronization board, so that the agents do not need to mention its name explicitly.

3.2 TLA+ formalization with free, blind offers

In our first solution an agent expresses its view about a potential synchronization, for service fruition, by the following two constructs:

Get This is an *action* (in TLA+ sense) describing an object – a bit triple – or set of objects that the agent *must* obtain, and that must not be already included in the agent's local set S. Satisfying this request is mandatory: no synchronization can take place if the agents involved in the interaction are collectively unable to provide these objects. The typical postcondition for this action is the inclusion of these the objects into S.

Show This is a *state function*(again using the TLA+ terminology) used to specify a set of objects (bit triples) that an agent owns in its set S, and is willing to show at the (default) synchronization board to any potentially interested agent. Any other agent can indeed access (some of) these displayed objects, via the *Get* action.

In calling this technical solution 'free, blind offers', 'free' refers to the fact that it is not possible to express any dependency of the offered objects on the requested ones, and 'blind' refers to the fact that an agent showing some values is not aware of which one of them, if any, is indeed used by which partner; thus, this information cannot contribute to the agent's postcondition.

The following TLA+ module, called *AAgents*, describe three agents that use the above two constructs in various ways.

MODULE *AAgents*

EXTENDS *BitTriples*

MODULE *AgentA1*

VARIABLE S

$$
\begin{array}{l}
\mathit{Init} \triangleq \; S = \mathit{BitTriples} \\
\mathit{Get} \triangleq \; S' = S \\
\mathit{Show} \triangleq \; \{y \in S : y[3] = 1\}
\end{array}
$$

MODULE *AgentA2*

VARIABLE S

$$
\begin{array}{l}
\mathit{Init} \triangleq \; S = \{0\} \times \mathit{Bits} \times \mathit{Bits} \\
\mathit{Get}(x) \triangleq \\
\qquad \land\, x \in \mathit{BitTriples} \\
\qquad \land\, x[1] = 1 \\
\qquad \land\, \neg x \in S \\
\qquad \land\, S' = S \cup \{x\} \\
\mathit{Show} \triangleq \; \{y \in S : y[2] = 1 \lor y[3] = 1\}
\end{array}
$$

MODULE *AgentA3*

VARIABLE S

$$
\begin{array}{l}
\mathit{Init} \triangleq \; S = \{\langle 0,\, 0,\, 0\rangle\} \\
\mathit{Get}(x1,\, x2) \triangleq \\
\qquad \land\, x1 \in \mathit{BitTriples} \land x2 \in \mathit{BitTriples} \\
\qquad \land\, \neg x1 \in S \land \neg x2 \in S \land x1 \neq x2 \\
\qquad \land\, x1[1] = 0 \land x2[1] = 1 \\
\qquad \land\, S' = S \cup \{x1,\, x2\} \\
\mathit{Show} \triangleq \; \{y \in S : y[1] = 0\}
\end{array}
$$

VARIABLE $S1,\, S2,\, S3$

$$
\begin{array}{l}
\mathit{AgA}1 \triangleq \text{INSTANCE } \mathit{AgentA}1 \text{ WITH } S \leftarrow S1 \\
\mathit{AgA}2 \triangleq \text{INSTANCE } \mathit{AgentA}2 \text{ WITH } S \leftarrow S2 \\
\mathit{AgA}3 \triangleq \text{INSTANCE } \mathit{AgentA}3 \text{ WITH } S \leftarrow S3
\end{array}
$$

$$
\begin{array}{l}
\mathit{Init} \triangleq \; \mathit{AgA}1!\mathit{Init} \land \mathit{AgA}2!\mathit{Init} \land \mathit{AgA}3!\mathit{Init} \\
\mathit{Offer} \triangleq \; \mathit{AgA}1!\mathit{Show} \cup \mathit{AgA}2!\mathit{Show} \cup \mathit{AgA}3!\mathit{Show} \\
\mathit{Next} \triangleq \; \exists\, x,\, x1,\, x2 \in \mathit{BitTriples} : \\
\qquad \mathit{AgA}1!\mathit{Get} \land \mathit{AgA}2!\mathit{Get}(x) \land \mathit{AgA}3!\mathit{Get}(x1,\, x2) \\
\qquad \land\, \{x,\, x1,\, x2\} \subseteq \mathit{Offer}
\end{array}
$$

$$
\mathit{Spec} \triangleq \; \mathit{Init} \land \Box[\mathit{Next}]_{\langle S1,\, S2,\, S3\rangle}
$$

The module imports, via the TLA+ EXTENDS contruct, an elementary module called *BitTriples*, defining this simple data type, and not shown in the specification; then, it encapsulates three module definitions, for the three agents *AgentA1*, *AgentA2*, *AgentA3*.

In the module for the first agent, the local state S is initalized by the *Init* predicate to be the whole space of bit triples. This agent does not require any object, and the *Get* action is only used for expressing the invariance of the local state S. The need to explicitly indicate unchanged state components is a feature of TLA+ (the so called 'frame' problem) that we do not need to discuss here. With the *Show* state function this agent identifies the set of values it is willing to offer, which is the subset of the bit triples it owns with last bit equal to 1.

In the synchronization behavior of *AgentA2* the *Get* action has parameter x, which identifies the requested object. The body of this action identifies the type of x (*BitTriples*), it restricts the choice by stating a property of it – the first component must be '1' – it explicitly excludes that this object be already present in the local state S, and it describes this inclusion as a postcondition. A set of objects is offered also by *AgentA2*, characterized by some properties of its elements.

In *AgentA3* the get action is only a bit more complex, and involves two objects, $x1$ and $x2$.

The composition of the three agents is expressed in the subsequent part of module *AAgents*. Three instances of the above modules are created, called $\mathit{AgA}1$, $\mathit{AgA}2$ and $\mathit{AgA}3$, by the INSTANCE construct, which also renames the local

variable S of each agent as, respectively, $S1$, $S2$ and $S3$. The global initialization is defined as the logical conjunction of the individual initialization conditions. State variable *Offer* is defined as the union of the sets of objects individually made available by the agents via their *Show* state functions. The pre- and post-conditions for the service interaction are expressed by action *Next*, which includes the conjunction of the invidual *Get* actions. A crucial conjunct in the body of the *Next* action states that the set of objects cumulatively required by the agents, namely $\{x, x1, x2\}$, must be included in the set of those collectively shown by the agents. Note that in TLA+ (and in the associated TLC model checker) extra copies of the same element in a set are disregarded, so that, for example, this inclusion is recognized as true:

$$\{0, 0, 1, 2, 2, 2\} \subseteq \{0, 1, 1, 2, 3\}$$

The definition of the module is completed by the usual temporal formula conventionally named *Spec*, involving predicates *Init* and *Next*, and the 'always' temporal operator, denoted by the box symbol '$\Box$'. A behavior is a sequence of states, that are assignments to the variables $S1$, $S2$ and $S3$. A behavior satisfies the *Spec* when its first state satisfies predicate *Init* and every step (i.e., pair of adjacent states) satisfies predicate *Next*, or keeps all three variables unchanged (*stuttering* step).

3.3 TLA+ formalization with constrained, blind offers

Our second formalization of the service specification and matching rule is a bit more general, in the sense that the objects an agent offers may depend on those it requests – within the same transaction: offers are 'constrained', as opposed to 'free'. This feature allows an agent to be more selective and precise about the items it offers. An yet, similar to the previous solution, an agent has no means to know which elements of *yy* are indeed exploited by its partners, and therefore it cannot modify its own state based on this information: offers are still 'blind'.

One way to express the above feature in TLA+ is to equip every interacting agent with a single action – a predicate that we may call *GetShow* – which is parameterized both by the requested and by the offered objects, so that the relation between the two sets can be expressed in the body of the predicate.

MODULE *BAgents*

EXTENDS *BitTriples*
VARIABLES $S1$, $S2$, $S3$

MODULE *AgentB1*

VARIABLE S
$Init \triangleq S = \{0\} \times Bits \times Bits$

$GetShow(x, yy) \triangleq$
$\quad \wedge x \in BitTriples$
$\quad \wedge x[1] = 1$

$\land \neg x \in S$
$\land S' = S \cup \{x\}$
$\land yy = \{y \in S : y[2] = x[2] \land y[3] = x[3]\}$

MODULE *AgentB2*

VARIABLE S

$Init \triangleq S = Bits \times \{0\} \times Bits$

$Get(x) \triangleq$
$\land x \in BitTriples$
$\land x[2] = 1$
$\land \neg x \in S$
$\land S' = S \cup \{x\}$

MODULE *AgentB3*

VARIABLE S

$Init \triangleq S = Bits \times Bits \times \{0\}$

$Show(yy) \triangleq yy = \{y \in S : y[1] = 1\}$

$AgB1 \triangleq$ INSTANCE $AgentB1$ WITH $S \leftarrow S1$
$AgB2 \triangleq$ INSTANCE $AgentB2$ WITH $S \leftarrow S2$
$AgB3 \triangleq$ INSTANCE $AgentB3$ WITH $S \leftarrow S3$

$Init \triangleq AgB1!Init \land AgB2!Init \land AgB3!Init$

$Next \triangleq \exists\, x,\, x1 \in BitTriples,\, yy,\, yy1 \in$ SUBSET $BitTriples :$
$AgB1!GetShow(x,\, yy) \land AgB2!Get(x1) \land AgB3!Show(yy1)$
$\land \{x,\, x1\} \subseteq (yy \cup yy1)$

$Spec \triangleq Init \land \Box[Next]_{\langle S1,\, S2,\, S3\rangle}$

For example, in module *AgentB1* above, the offered set of objects, yy, depends on the values of the second and third elements of the x object, with yy and x being exchanged within the same atomic interaction. Action *GetShow* is designed to handle these two object, that flow in opposite directions.

AgentB2 and *AgentB3* illustrate the separate use of operators *Get* and *Show*.

In action *Next* of the outer module *BAgents*, existential quantification is used both for the requested objects and for the offered object sets, while in the previous case it only referred to the former. Then, in the scope of this quantification, the pre- and post-conditions of the three agents are conjoined, and the condition is expressed that the requested objects be included in the union of the offered object sets, as done for the *AAgents*.

For notational uniformity we have used a parametric *Show*, with parameter yy, but when an agent only needs to 'show' objects, as it happens with *AgentB3*,

the previously adopted formal solution of using a simple state function is also possible.

Note that, in spite of the the 'blindness' limitation, the availability of both variables x and yy in the body of the *GetShow* predicate gives us at the possibility, whenever meaningful, to express a dependency also between the set yy, *as a whole*, and the next state S'.

3.4 TLA+ formalization with constrained offers

Our third formalization is an extension of the second one and gives more control on the 'offered' objects, that can *individually* affect and be affected by the local state of their provider. Offered objects are now handled by *Put* actions, that are similar to *Get* actions, and may involve post-conditions. In the example below, all objects offered by the three agents, denoted by the y variables, contribute by subtraction to the updating of the respective local states. We use one synchronization action for each agent (*Put*, *GetPut*, *GetGetPut* respectively), which simultaneously handles the requested and offered objects for that agent and their effects on the local state.

MODULE *CAgents*

EXTENDS *BitTriples*
VARIABLES $S1$, $S2$, $S3$

MODULE *AgentC*1

VARIABLE S
$Init \triangleq S = \{0\} \times Bits \times Bits$
$Put(y) \triangleq$
$\quad \wedge\, y \in S$
$\quad \wedge\, S' = S \setminus \{y\}$

MODULE *AgentC*2

VARIABLE S
$Init \triangleq S = Bits \times \{0\} \times Bits$
$GetPut(x,\, y) \triangleq$
$\quad \wedge\, x \in Bits \times \{1\} \times Bits$
$\quad \wedge\, \neg x \in S$
$\quad \wedge\, y \in S$
$\quad \wedge\, y[1] = 0$
$\quad \wedge\, S' = (S \cup \{x\}) \setminus \{y\}$

MODULE *AgentC*3

VARIABLE S
$Init \triangleq S = Bits \times Bits \times \{0\}$
$GetGetPut(x1,\, x2,\, y) \triangleq$
$\quad \wedge\, x1 \in BitTriples \wedge x2 \in BitTriples$
$\quad \wedge\, x1 \neq x2$

$\land \neg x1 \in S \land \neg x2 \in S$
$\land\ y \in S$
$\land\ y[1] = 1$
$\land\ S' = (S \cup \{x1,\ x2\}) \setminus \{y\}$

$AgC1 \triangleq$ INSTANCE $AgentC1$ WITH $S \leftarrow S1$
$AgC2 \triangleq$ INSTANCE $AgentC2$ WITH $S \leftarrow S2$
$AgC3 \triangleq$ INSTANCE $AgentC3$ WITH $S \leftarrow S3$

$Init \triangleq AgC1!Init \land AgC2!Init \land AgC3!Init$

$Next \triangleq \exists\, g1,\ g2,\ g3,\ \ p1,\ p2,\ p3 \in BitTriples :$
$\land\ AgC1!Put(p1)$
$\land\ AgC2!GetPut(g1,\ p2)$
$\land\ AgC3!GetGetPut(g2,\ g3,\ p3)$
$\land\ \{g1,\ g2,\ g3\} \subseteq \{p1,\ p2,\ p3\}$

$Spec \triangleq Init \land \Box[Next]_{\langle S1,\ S2,\ S3\rangle}$

Quantification now refers to individual objects, both for requested and for offered items. Furthermore, the usual inclusion between the two sets is enforced; this means that, similar to the previous two cases, the overall set of objects that eventually satisfy all the requests may be strictly smaller than the overall set of offered objects. Thus in a sense this solution might still be considered as a 'blind' one, because an agent still has no complete knowledge about which subset of its offers were eventually used by other agents. However, while in the previous cases this lack of information suggested us exclude the possibility to express any influence of the individual offered objects on the local state, in the third solution we have adopted the somewhat opposite extreme of letting every individual offered object, if desired, affect the local state in its own way, regardless of whether or not the offer was used by some partner.

Are there any intermediate scenarios between these two extremes, that we can express in TLA+?

The next example illustrates the use of mutually exclusive offers. Agent *CAgentXor* below puts forward its offer through a single variable y, but this variable can be bound to two mutually exclusive objects, which correspond to two different post-conditions for state variable S. Mutual exclusion is achieved via the first element of bit triple y.

MODULE *CAgentXor*

EXTENDS *BitTriples*

VARIABLE S
$Init \triangleq S = Bits \times \{0\} \times Bits$
$GetPut(x,\ y) \triangleq$

$$
\begin{array}{l}
\land\ x \in \mathit{Bits} \times \{1\} \times \mathit{Bits} \\
\land\ \neg x \in S \\
\land\ y \in S \\
\land\ y[3] = x[3] \\
\land\ (\quad \lor\ (\quad \land\ y[1] = 0 \\
\qquad\qquad\quad\ \ \land\ S' = (S \cup \{x\}) \setminus \{y\}) \\
\qquad\quad \lor\ (\quad \land\ y[1] = 1 \\
\qquad\qquad\quad\ \ \land\ S' = (S \cup \{x\})))
\end{array}
$$

However this is only a partial solution to the problem of blind offers. The good feature is that the agent is now able to specify a set of potential offers and associated local state changes, while at the same time letting only one of them be offered and affect, correspondingly, the local state. The bad feature is that the agent still does not know whether that one offer was actually used by some partner, or was simply redundant. Note that this problem is not solved by replacing the inclusion of requests into offers by the equality of these two sets, in the body of the *Next* predicate of the outer TLA+ module expressing the composition of agents, since duplicate offers from different agents would still be possible: any duplicate would be redundant.

A solution to this problem is presented in the next section.

4 A generalized atomic service synchronization rule

The limitation of the service synchronization policies described in the previous section is that an agent is not aware of which ones of its offers have been *essential* for the success of the synchronization, and which ones have been redundant and disregarded by the other partners. Thus, the agent cannot reflect this information in its postcondition.

In our solution to this problem, a service-oriented operation for an agent shall still involve two sets of objects, the requested and the offered ones, and, as before, all requests must be satisfied while not all offers must be taken. However, we want now to be precise in the computation of an agent's postcondition, and let it reflect the exact subset of essential offers, as mentioned above. The first step for doing this is to associate individual postconditions to individual offers, so that the postcondition for an agent's operation can then be defined as the selective composition of only those postconditions that are associated to the contributing offers. Since composing postconditions referring to the same state variable is not a trivial task, we shall restrict to very simple postconditions, namely plain insertion and deletion of elements in sets.

The structure of the body of a synchronization operation shall be as follows:

GetAll $\{x1, ..., xn\} : G(S, x1, ..., xn, S')$
PutSome $\{y1 : P_{y1}(S, y1), ..., ym : P_{ym}(S, ym)\}$

The set of synchronization variables $\{x1, ..., xn\}$ identifies the requested objects. All requests must be satisfied for the interaction to take place. G is a precondition that relates the requested objects with one another and with the current value of the local state S.

The set of synchronization variables $\{y1, ..., ym\}$ identifies the offered objects. Not all offers must be consumed. In this case we associate a different precondition – the P's – to each offered object. A default conjunct of all these preconditions is that the y object be owned by the agent that offers it: $y \in S$ (relaxation of this condition is left for further investigation).

Let $Op = \{Op_i, \ldots, Op_k\}$ be a set of operations, each belonging to a different agent. Each operation basically expresses a partial view about the set of synchronization objects that enable the atomic interaction, thus service fruition. All these operations can successfully engage in a synchronization when it is possible to find a set of objects that matches the view expressed by each operation.

Let $Var = Var_1 \cup \ldots \cup Var_k$ be the set of synchronization variables, consisting of the union of the variable sets (Var_i) from each operation, that we assume to be disjoint. We view tuples as sets, for taking their union.

Let $VarGet_i$ and $VarPut_i$ denote, respectively, the tuples of variables (say $\{x1, ..., xn\}$ and $\{y1, ..., ym\}$) used for denoting the requests (*GetAll*) and offers (*PutSome*) of operation i, and let *VarGet* and *VarPut* denote the corresponding unions over operations.

Let $G_i(S_i, \ldots, S_i')$ be the predicate appearing under the *GetAll* keyword in the body of operator $Op_i, i = 1, \ldots, k$. The dots in parentheses stand for the variables denoting the requested objects.

Let $P_y(S_j, y, S_j')$ be a predicate appearing under the *PutSome* keyword in the body of some operator Op_j, where $y \in VarPut$. We assume that all such variables are different across the operations. It is notationally convenient to use variable y itself as the index of this predicate.

Let $F : \{S_1, \ldots, S_k\} \rightarrow Sets$ be the state valuation function that describes the current global system state by associating a value – a set – to each local state variable name S_i of agent Ag_i.

Synchronization rule The operations $Op = \{Op_i, \ldots, Op_k\}$ can successfully engage in an atomic service exchange if there exists

- a nonempty set of objects $O = \{O_1, \ldots, O_h\}$, called *synchronization objects*,
- a valuation function $f : Var \rightarrow O$, and
- a state valuation function $F' : \{S_1', \ldots, S_k'\} \rightarrow Sets$ that associates values to *primed* state variable symbols,

such that:

1. f is surjective (that is, each synchronization object is hit by at least one variable);
2. $f|_{VarGet}$ is total (that is, each request is satisfied);
3. $f|_{VarPut}$ is partial (that is, not all offers are necessarily taken);
4. $f|_{Var_i}$ is injective, for $i = 1, \ldots, k$ (that is, f is locally injective, and each agent describes a set of distinct synchronization objects);

5. $\forall O_i \in O : f^{-1}(O_i) \cap \mathit{VarPut} \neq \emptyset$ (that is, each object is hit by at least one variable denoting an offer);
6. $\forall i \in \{1, \ldots, k\} : \{F, f, F'\} \models G_i(S, \ldots, S')$ (that is, the current and next state values, as espressed by F and F', and the values of the synchronization objects, as expressed by f, satisfy all predicates associated with object requests);
7. $\forall y \in \mathit{VarPut} : [f(y)$ is defined $\Rightarrow \{F, f, F'\} \models P_y(S_j, y, S'_j)]$ (that is, if function f associates an object to y, then, similar to the previous requirement, the predicate associated with this variable must be satisfied). □

The first five requirements characterize function f. In particular, requirement 5 is meant to guarantee that no synchronization object is created 'artificially' without any agent being able to actually provide, via the *PutSome* construct, a fully defined instance of it. The last two requirements put further constraints on f, but also define the next global state, which is readily obtained by the assignment

$$S_i \leftarrow F'(S'_i), i = 1, \ldots, k.$$

According to the above synchronization rule the next global state depends on a composition of predicates including *all* the G_i's, and *some* of the P_y's. No conflict about the computation of the next state could arise if we only considered the G_i's, since each of these predicates handles a separate portion of the global state, namely a set S_i. However, conflicts could arise when the P_y's come into play, since they express additional requirements on those state variables, and it may also happen that some of them share the same local state variable S_i's. The specifier does not know at design time which bundle of P_y predicates is going to be 'activated' each time the service is delivered; thus, in writing these predicates one should conceive postconditions in isolation, one for each offer variable y, but in such a way that any potential grouping of these postconditions would be feasible and meaningful.

Consider for example the two definitions of offer predicates:

$$P_{y1}(y1) =_{def} y1 \in S_j \wedge S'_j = S_j \setminus \{y1\}$$

$$P_{y2}(y2) =_{def} y2 \in S_j \wedge S'_j = S_j$$

that refer to the same state variable S_j. Note that, in line with the previous TLA+ specifications, we have omitted the state arguments from the headers, since these variables are global within the body of an agent and to its operations. The two predicates express conflicting views about the next state value S'_j, and, if taken literally, cannot be simultaneously satisfied.

However, conflicts can be eliminated if (i) we insist that all the offer predicates P_y's be only of the two types of the example above, namely, they either preserve the state set or remove one element from it, and (ii) we consider all removal operations in parallel, while disregarding the individual conjuncts involving the value S'_j in the bodies of the predicates, be they conservative or

subtractive. Recall that, by condition 4 of the synchronization rule, all the synchronization objects expressed by an operation are distinct, hence adding and removing these objects to and from the local set are operations that can be equivalently performed in any order or in parallel.

5 Conclusions

In this paper we have investigated some formalizations of service offer/request descriptions, and the associated matching problem. Our solutions are rather abstract, and are based on predicate logic. We expect this choice to be helpful in:

- sorting out some of the key concepts in the SOC (Service-Oriented Computing) paradigm, which, despite the abundance of proposed technologies (e.g. those related with the Semantic web), could still benefit, in our opinion, from some brainstorming activity at the level of fundamental concepts;
- identifying useful informal description techniques, based on natural language, that be usable by many of the actors that play some role in SOC, including those with no skills in programming and formal specification;
- facilitating the integration with description logics, which is at the basis of many current, ontology-based proposals for the Semantic Web.

A primary concern in our atomic service synchronization rules is to try and combine precise information about the local state with fuzzy information about potential remote offers and requests. In viewing things under this light we have been largely inspired by the work of Hausmann, Heckel and Lohmann [4], who propose a combination of UML and graph transformation rules for expressing service requests and offers. A transformation rule consists of two object diagrams, representing a partial view of a portion of the 'world' respectively before and after service fruition. The service provider rule may be different from the service requestor rule, reflecting their partial mutual knowledge. Formal conditions are then identified for two rules to successfully match. One feature of these transformation rules is that only effects that are observable as structural changes (i.e. deleted or created objects or links) can be expressed. The use of logical formulae, as illustrated by our TLA+ specifications, can clearly offer higher expressiveness, and we believe that the integration of the two approaches could be profitably investigated.

Although we are concerned with Service Oriented Computing, we may view our atomic service exchanges simply as generic interaction mechanisms, and assess them under this light. In this respect, our proposal appears much more in line with a shared-event policy (also called 'rendez-vous', or 'hand-shake'), than with a shared-variable policy, since the synchronization objects described by the interacting parties are not persistent, but exist only within the scope of an instantaneous interaction.

We can therefore appropriately compare our interaction mechanisms with those adopted by process algebras such as CSP [5] and LOTOS [1], which in particular support multi-party synchronization.

For example, LOTOS does support, in its own way, the expression of a mix of precise and fuzzy information about the components of an interaction. Assume three agents (LOTOS processes P, Q and R) want to synchronize at some board (a LOTOS 'gate'), say *GateA*. This is expressed by the parallel composition of the three processes, with synchronization at *GateA*:

```
P |[GateA, ...]| Q |[GateA, ...]| R
```

In the expressions that characterize the individual behavior of these processes one would then write some *action prefix* referring to the shared gate, each giving a partial view of the synchronization event. In the example below, the interaction event has two parameters, and each process expresses a different view about them. Each action prefix thus expresses two *value offers*, that must be understood as parallel, not sequential, each following a punctuation mark ('?' or '!'). The question mark identifies an open offer, while the exclamation mark offers a single, precise value. The matching rule is purely positional, and variable names are irrelevant for the synchronization (but are used by each process for capturing the values established by the synchronization): synchronization is possible when, position by position, the conjunction of the constraints individually expressed by the processes admit a solution.

```
process P := ... GateA ? x: Nat  ! 35; ...
process Q := ... GateA ? x: Nat  ! H+K [x < 10]; ...
process R := ... GateA ! 8       ? y: Nat [IsEven(y)]; ...
```

In the example above the first parameter of the interaction can only have value 8, while no solution exists for the second parameter, regardless of the value of expression 'H+K', since the only offer from P is an odd number, while R only admits even numbers. As a consequence, the whole interaction is unfeasible.

The analogy between this mechanism and the synchronization policy described in this paper is clear, with synchronization objects in the latter corresponding to interaction parameters in the former. But a crucial difference is that in LOTOS the number of synchronization objects must be established precisely at specification time, while a major goal of our proposal has been exactly to introduce more flexibility in the expression of interaction object sets.

How complex can a service be, if we insist in describing it by a single atomic interaction as done in this paper? And is it always possible to decompose a service of higher complexity into a (natural) sequence of steps that can be individually handled by our technique?

References

1. Ed Brinksma and Tommaso Bolognesi. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.
2. Ian Horrocks Franz Baader and Ulrike Sattler. Description logic. In R. Studer S. Staab, editor, *Handbook on Ontologies*, chapter 1. Springer, 2004.
3. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic, 2003.

4. J.H. Hausmann, R. Heckel, and M. Lohmann. Model-based development of web service descriptions: Enabling a precise matching concept. *International Journal of Web Services Research*, 2(2):67–84, 2005.
5. C. A. R. (Tony) Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
6. Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 09(1):75–81, 2005.
7. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
8. S. McIlraith, T. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems (Special Issue on the Semantic Web)*, 16(2), 2001.