Deriving Specifications for Composite Web Services

George Baryannis
Department of Computer Science
University of Crete and
FORTH-ICS, Heraklion, Greece
Email: gmparg@csd.uoc.gr

Manuel Carro
School of Computer Science
Universidad Politécnica de Madrid (UPM) and
IMDEA Software Institute, Madrid, Spain
Email: manuel.carro@imdea.org

Dimitris Plexousakis
Department of Computer Science
University of Crete and
FORTH-ICS, Heraklion, Greece
Email: dp@csd.uoc.gr

***Abstract*—We address the problem of synthesizing specifications for composite Web services, starting from those of their component services. Unlike related work in programming languages, we assume the definition of the component services (i.e. their code) to be unavailable — at best, they are known by a specification which (safely) approximates their functional behavior. Within this scenario, we deduce general formula schemes to derive specifications for basic constructs such as sequential, parallel compositions and conditionals and provide details on how to handle the special cases of loops and asynchronous execution. The resulting specifications facilitate service verification and service evolution as well as auditing processes, promoting trust between the involved partners.**

***Keywords*-specification of service compositions, inference of specifications, service composition**

I. Introduction

Service composition enables service-based systems to be built using accepted engineering principles, such as (service) reusability and composability. Composite services provide value-added services that achieve functionality otherwise unattainable by atomic services. In order to achieve these goals, composite services should be made available to consumers in the same way as atomic services, abstracting away complex implementation details. This can be accomplished by providing formal specifications which present the minimum information required to understand the functionality offered, often by describing the inputs, outputs, preconditions and effects (known as IOPEs) of the composite service.

Formal specifications are indispensable in a variety of service-related activities. Similarly to the case of programming specifications, service specifications could be used as a basis to construct a service based on a set of requirements agreed upon by the parties involved, or to check that some existing specification meets a set of requirements. Furthermore, they can assist in auditing processes that check third party or legacy code conformance to specifications, promoting trust between digital society partners, since specification conformance is one step towards trustworthiness.

Specifications also play a major role in verification techniques, as well as in the evaluation of the results of service adaptation or service evolution. Verification involves checking whether a system satisfies a property given the particular property and a formal description (i.e. a specification) of the system. Composite specifications are also beneficial when one attempts to deduce whether a set of services can actually be composed in a meaningful way, by detecting inconsistencies between conditions while creating the specifications.

While existing service description frameworks attempt to describe service compositions (including those for Web services) using a variety of models ranging from orchestrations to choreographies to Finite State Machines, no attempt (to the best of our knowledge) has been made to handle the problem of automatically producing specifications for a composite service, based on the specifications of the participating services.

Traditional tasks involving specifications, especially in the field of programming languages, include generating code out of specifications (using languages such as VDM [1] or B [2]), and vice-versa or checking that a particular code conforms to a specification. The case handled in this paper is different: we aim at inferring specifications for a set of individual components, functioning as a whole (based on a composition schema), which are only known through their specifications.

Our approach involves characterizing the *semantics* of control structures assuming generic preconditions and post-conditions of the involved basic services. From them we synthesize the specification (or a safe approximation) of the composite service. When applied to particular services, the generic pre- and post-conditions are instantiated to the actual ones; this can be used to further simplify the logical expressions of the global specification.

The rest of this paper is organized as follows. Section II offers a motivating example that illustrates the issues behind creating a composite specification. Section III provides an analytical description of the derivation process for most fundamental control constructs while Section IV deals with the cases of loops and asynchronous interaction. Section V offers a brief description of related work and Section VI concludes and points out topics for future work.

II. Motivation

In this section, we present a motivating example to illustrate the need for service composition specification in a service evolution scenario, as well as the issues behind

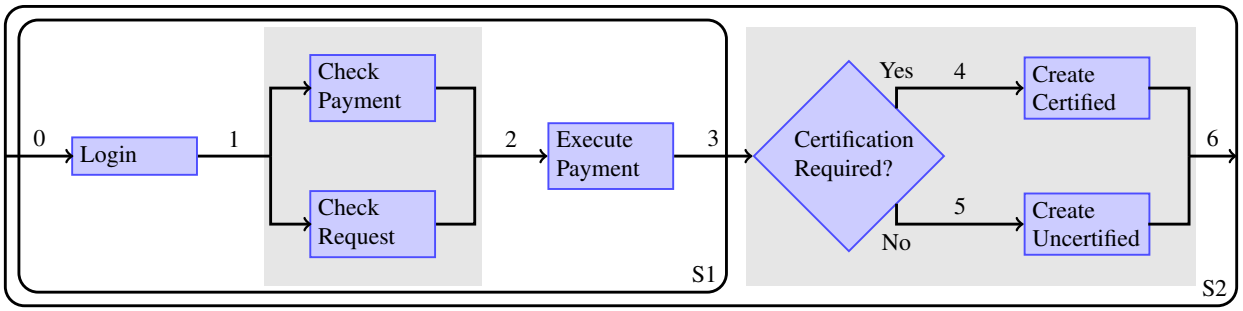

Figure 1. Composite process of the motivating example

deriving a composite specification, given the specifications of the participating services. The example is based on the E-Government case study of the European Network of Excellence S-Cube [3].

In this case study, citizens submit applications to request some government-related service, such as obtaining government-issued documents, as illustrated in Figure 1. Users log into the system and fill in forms regarding their request as well as payment details, which are then simultaneously processed before the payment process can begin. If users demand authentication for their documents, then a certification process is executed, resulting in the delivery of a certified document to the user. Otherwise, an uncertified document is delivered. For reasons that will be clarified later, we have labeled the states before and after particular points in the process. For instance, state 1 is the state after the completion of $Login$ and before beginning execution of services $CheckRequest$ and $CheckPayment$.

Let us assume that the individual tasks described in Figure 1 are implemented as Web services. Table I offers a possible specification of the services involved in the process, in terms of their preconditions and postconditions, expressed in first-order logic. s_i and s_o denote the states before and after execution of the particular service respectively. Suppose that, initially, we have a composite service S1 that is implemented according to a specification T1 in order to handle the document purchase process excluding certification, as shown in Figure 1. Then, it is decided that some documents should be certified with a digital signature, so the initial specification is augmented to T2. In order to meet the new requirements, service S1 needs to be evolved into a new composite service S2. We need to check if the evolved service S2 meets the new specification T2. A composite specification I(S2) can be derived based only on the information at hand (the orchestration definition of S2 and the specifications of the participating services, including that of S1) and check if I(S2) subsumes T2.

The composite specification should explicitly state all conditions that must be true before the execution of the whole composite service, as well as all conditions that are true after a successful execution. While we have preconditions

Service	**Preconditions**
Login	$Valid(user, s_i) \wedge \neg LoggedIn(user, s_i)$
CheckRequest	$FilledIn(request, s_i) \wedge LoggedIn(user, s_i)$
CheckPayment	$FilledIn(payForm, s_i) \wedge LoggedIn(user, s_i)$
ExecutePayment	$Valid(payForm, s_i)$
CreateCertified	$PayCompleted(doc, user, s_i)$
CreateUncertified	$PayCompleted(doc, user, s_i)$
Service	**Postconditions**
Login	$LoggedIn(user, s_o)$
CheckRequest	$Valid(request, s_o)$
CheckPayment	$Valid(payForm, s_o)$
ExecutePayment	$PayCompleted(doc, user, s_o)$
CreateCertified	$CertifCompleted(doc, user, s_o) \wedge$ $Delivered(certifDoc, s_o)$
CreateUncertified	$Delivered(doc, s_o)$

Table I
ATOMIC SERVICE SPECIFICATIONS

and postconditions for each participating service, there is no obvious way of deciding which part of them will be included in the composite specification.

We propose a derivation process to construct the composite specification using a bottom-up approach that is based on structural induction. The approach is applicable to any block-structured process, as well as graph-based ones, provided they can be transformed to block-structured equivalents [4]. The approach is based on the availability of the composition schema, which can be obtained, for instance, from the BPEL document of the composite service. In the following section, we formulate the derivation for some fundamental control constructs.

III. CALCULATING PRE- AND POST-CONDITIONS

Formal specifications have been extensively used in computer science in order to rigorously describe what a system should do and can also similarly be used to offer a formal presentation of what a Web service provides and under which circumstances. A traditional format for a specification contains the conditions that should be met prior to execution (called preconditions, which we will denote by P) and the conditions that result after a successful execution of the program (called postconditions or results, denoted by Q).

In contrast to program specifications where preconditions are usually the weakest possible ones (and postconditions the

strongest), in the case of services, P and Q can be expected to be safe approximations, e.g., P can be stronger than the weakest possible precondition for that particular service. P can therefore disallow invocations in cases where the actual code would work, but it would not allow invocations in a state not entailed by the weakest precondition (similarly for the postcondition Q). Note that if the approximation were done in the opposite direction, i.e., with P being weaker than the weakest precondition, executions allowed by P could be erroneous.

A. Specification Semantics

A FOL semantics for a service specification with regard to its preconditions and postconditions is:

$$\forall x \cdot (P(x, s_i) \Rightarrow \exists y \cdot Q(x, y, s_o))$$

$P(x, s_i)$ and $Q(x, y, s_o)$ are the (approximations of) preconditions and postconditions, respectively, using predicates, where x and y are vector variables that represent accordingly the input fed to the service and the returned output. s_i and s_o are fixed for a given composition schema and denote execution points. The reason for using such state identifiers as additional arguments to the predicates is to differentiate the truth value of predicates based on when they are evaluated, without having to carry around a usually cumbersome notion of state of the world. This allows us to express fluency in predicate values in a lean way. We choose FOL instead of other formalisms such as the situation or fluent calculi, to employ a widely known formalism with equal expressive power, as well as base our proofs in automated theorem provers such as Prover9 [5], which are mature enough to provide high performance in practice.

Given similar specifications for the services participating in a composition, we want to construct a specification for the composite service c, which essentially involves calculating a set P_c of preconditions and a set Q_c of postconditions such that the following holds:

$$\forall x \cdot (P_c(x, s_i) \Rightarrow \exists y \cdot Q_c(x, y, s_o))$$

where $P_c(x, s_i)$ and $Q_c(x, y, s_o)$ are built using the preconditions and postconditions of the component services.

We insist that the derived specifications maintain the approximation that we mentioned earlier: preconditions for c derived from preconditions that are not the weakest themselves should be stronger than (or at least as strong as) the weakest possible precondition for the composition. We will return to this issue in Section V. In the rest of this section we will show how to calculate preconditions and postconditions for fundamental control constructs [6]. In all cases, and without loss of generailty, we consider compositions of two services.

B. Sequence

We denote sequential invocation by $A(x, z); B(z, y)$, where all variables z that constitute the input of service B are produced as an output of service A. This includes

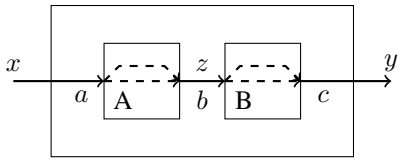

Figure 2. Sequential composition of services A and B. Information routed inside A and B is explicitly represented.

variables which are input for B and which do not result from the execution of A, but come directly from sources external to the sequence. For the purposes of the specification we consider them to be *routed* untouched through A. a, b, and c respectively denote the state before the execution of A, after A and before B, and after the execution of B (Fig. 2). The semantics of the sequential composition would be:

$$\begin{aligned} \forall x \exists z \cdot ((P_A(x, a) \Rightarrow Q_A(x, z, b)) \wedge \\ (P_B(z, b) \Rightarrow \exists y \cdot Q_B(z, y, c))) \end{aligned} \quad (1)$$

From Formula (1) we can deduce:

$$\begin{aligned} \forall x \exists z \cdot (P_A(x, a) \wedge P_B(z, b) \Rightarrow \\ \exists y \cdot (Q_A(x, z, b) \wedge Q_B(z, y, c))) \end{aligned} \quad (2)$$

However, Formula (2) exposes internal variable z to the precondition. This is not desirable, since preconditions should be externally checkable and depend only on the input data to the composition. We can use the postcondition of A to eliminate this shortcoming:

$$\begin{aligned} \forall x \exists z \cdot (P_A(x, a) \wedge Q_A(x, z, b) \wedge P_B(z, b) \Rightarrow \\ \exists y \cdot (Q_A(x, z, b) \wedge Q_B(z, y, c))) \end{aligned} \quad (3)$$

In Formula (3) the precondition can be checked exclusively based on x.[1]

The derived specification shows what conditions must be met before executing the sequence $A; B$ and which conditions will hold after a successful execution. However, it does not state clearly which conditions must hold for the composition to be *valid*. For a sequential composition to be valid there should be at least one case where it is applicable: the precondition of the first service should hold and the precondition of the second one should be true when applied to the result of the first service. Expressed in FOL, this validity condition is as follows:

$$\exists x, z, y \cdot (P_A(x) \wedge Q_A(x, z, b) \Rightarrow P_B(z, b)) \quad (4)$$

Note that there is a close connection with Hoare's triples [7] (more on that in Section V): the notion of an non-valid precondition (an empty domain for the composition) would correspond to inferring a *false* precondition for a piece of code in Hoare's logic.

[1]More details on this work, including Prover9 proofs for all derivation steps can be found online at http://www.csd.uoc.gr/ ~gmparg/specs

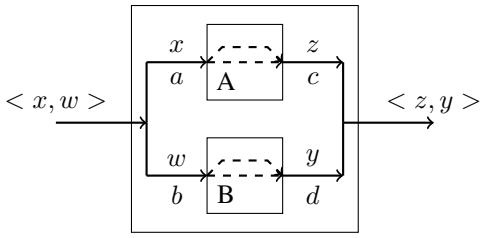

Figure 3. Parallel composition of services A and B

C. Parallel Composition

Parallel composition follows the pattern shown in Fig. 3, but there are different variations that depend mainly on what is considered successful completion:

- AND-Split/AND-Join ($A(x,z) \wedge B(w,y)$): there are two (or more) diverging branches of activities that are executed concurrently, which eventually converge after all activities have completed successfully.
- OR-Split/OR-Join ($A(x,z) \vee B(w,y)$): not all of the diverging branches are necessarily activated. Instead, a mechanism selects one or more of them to be executed each time. Also, at the merging stage there is no need for synchronization between the converging branches.
- XOR-Split/XOR-Join ($A(x,z) \oplus B(w,y)$): only one of the diverging branches is allowed to be executed and is expected to provide results at the end.

For the composite service of Fig. 3, if we consider the AND-Split/AND-Join case, the following holds:

$$\forall x \cdot (P_A(x,a) \Rightarrow \exists z \cdot Q_A(x,z,c)) \wedge \\ \forall w \cdot (P_B(w,b) \Rightarrow \exists y \cdot Q_B(w,y,d)) \quad (5)$$

Note that it is possible for states a and b (and for states c and d as well) to be equivalent, but we leave equations in their general form. In a similar way to the sequential case, we can deduce from Formula (5) the following:

$$\forall x \forall w \cdot (P_A(x,a) \wedge P_B(w,b) \Rightarrow \\ \exists z, y \cdot (Q_A(x,z,c) \wedge Q_B(w,y,d))) \quad (6)$$

In this case, there is no need for further steps, as all input and output variables should be externally visible. Following similar steps, we can derive the specifications shown in Table II for the cases of OR-Split/OR-Join and XOR-Split/XOR-Join. As far as the validity condition is concerned, for all parallel composition patterns, we only need to ensure that there is a case where the preconditions of both services are true:

$$\exists x, w \cdot (P_A(x,a) \wedge P_B(w,b)) \quad (7)$$

D. Conditional Constructs

Conditional constructs, such as if-then-else or switch statements, evaluate a condition in order to decide which branch will be executed. Similarly to the XOR-Split/XOR-Join pattern, only one of the branches is selected, based on the truth value of the condition.

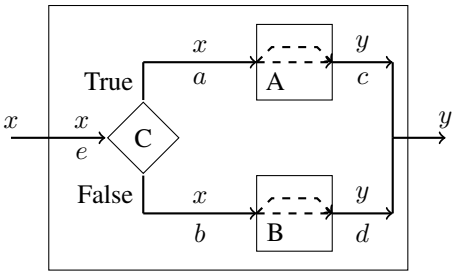

Figure 4. Conditional composition of services A and B

In an if-then-else composition of the form *IF C(x) THEN A(x, y) ELSE B(x, y)*, as seen in Figure 4, if the condition C is true, A is executed; if it is false, B is executed. Input variable x refers to either of the two services since the branches are exclusive and the same is true for output variable y. x also contains the terms that are involved in the condition C. Hence, the following should hold:

$$\forall x \exists y \cdot \\ ([(C(x,e) \wedge P_A(x,a)) \vee (\neg C(x,e) \wedge P_B(x,b))] \Rightarrow \\ ([(C(x,e) \wedge Q_A(x,y,c)) \vee (\neg C(x,e) \wedge Q_B(x,y,d))])) \quad (8)$$

Determining whether a conditional composition is valid depends on finding a case where the precondition derived above is valid, resulting in the following validity check:

$$\exists x \cdot ((C(x,e) \wedge P_A(x,a)) \vee (\neg C(x,e) \wedge P_B(x,b))) \quad (9)$$

Table II shows the derived preconditions and postconditions for the constructs that we examined in this Section. We can now return to the motivating example and apply the results for the patterns of sequence, AND-Split/AND-Join and conditional execution in order to derive the following complete specification for the composite service:

$$\forall request, payForm, doc, user, \exists certifDoc \cdot \\ (Valid(user,0) \wedge \neg LoggedIn(user,0) \\ \wedge FilledIn(request,1) \wedge FilledIn(payForm,1) \\ \wedge LoggedIn(user,1) \wedge Valid(payForm,2) \wedge \\ [(ReqCertif(doc,user,3) \wedge \\ PayCompleted(doc,user,4)) \vee \\ (\neg ReqCertif(doc,user,3) \wedge \\ PayCompleted(doc,user,5))] \\ \Rightarrow (LoggedIn(user,1) \wedge Valid(request,2) \\ \wedge Valid(payForm,2) \wedge PayCompleted(doc,user,3) \\ \wedge [(ReqCertif(doc,user,3) \wedge \\ CertifCompleted(doc,user,6) \\ \wedge Delivered(certifDoc,user,6)) \vee \\ (\neg ReqCertif(doc,user,3) \wedge Delivered(doc,user,6))])) \quad (10)$$

Construct	Precondition
Sequence	$P_A(x,a) \wedge Q_A(x,z,b) \wedge P_B(z,b)$
AND-Split/Join	$P_A(x,a) \wedge P_B(w,b)$
OR-Split/Join	$P_A(x,a) \wedge P_B(w,b)$
XOR-Split/Join	$P_A(x,a) \wedge P_B(w,b)$
Conditional	$(C(u,e) \wedge P_A(x,a)) \vee (\neg C(u,e) \wedge P_B(x,b))$
Construct	**Postcondition**
Sequence	$Q_A(x,z,b) \wedge Q_B(z,y,c)$
AND-Split/Join	$Q_A(x,z,c) \wedge Q_B(w,y,d)$
OR-Split/Join	$Q_A(x,z,c) \vee Q_B(w,y,d)$
XOR-Split/Join	$Q_A(x,z,c) \oplus Q_B(w,y,d)$
Conditional	$(C(u,e) \wedge Q_A(x,y,c)) \vee$ $(\neg C(u,e) \wedge Q_B(x,y,d))$

Table II
DERIVED PRECONDITIONS AND POSTCONDITIONS

Note that $LoggedIn(user, 1)$ and $Valid(payForm, 2)$ appear on both sides of the implication, and therefore can be removed from the right hand side without changing the meaning of the formula. This is an example of specification simplification, which becomes more crucial as compositions become larger and more complicated. Simplification actions range from dealing with duplicate predicates to applying known equivalences, to tasks that depend on specific knowledge on the particular composite service.

IV. SPECIAL CASES

A. Loops

The loop structure was excluded from the discussion in Section III. Loops allow for the repeated execution of a task or a process until a condition (the loop guard) ceases to hold. This poses a significant challenge as there is no *a priori* knowledge of how many iterations will be performed, thus the state identifiers that we used in all other constructs to differentiate predicate evaluations are rendered inapplicable.

Without knowledge of its precondition and postcondition, a possible way to specify a loop is either based on an upper limit on the number of iterations or through its invariant. The former relies on limits that can't be expected to be always available and results in recursive specifications which are difficult to work with. A loop invariant I is a statement that is true before and after each iteration of the loop, thus it stays unaffected by the loop execution. By definition, the loop invariant is a loop precondition ($I \Rightarrow P$). Moreover, a loop postcondition can be derived through the following implication: $I \wedge \neg C \Rightarrow Q$, where C is the loop guard, the condition that must be true for the iteration to continue [8].

Several issues are raised in the discussion of using invariants to generate loop specifications. We once again have to consider the special characteristics of services, meaning we can expect that a generated invariant is an approximation, as are preconditions and postconditions. The correct direction of the approximation needs to be determined. It can be deduced that we need a stronger approximation of the invariant in order to derive a useful precondition, and a weaker approximation in order to derive a useful postcondition.

Another issue concerns the invariant generation process itself. While, in general, invariant generation is based on a set of commands (the loop program), in our case we only have an approximate specification of the commands of the loop, so the generation process must be based on this information. Essentially, the invariant generator must take into account the preconditions of the looped commands, so that the resulting invariant at least implies these preconditions, as mentioned at the beginning of this section. Furia and Meyer [9] provide a concise summary on the different methods that have been proposed in literature to generate loop invariants. Of the works mentioned, only static methods such as abstract interpretation and constraint-based techniques, that do not depend on executing the program and do not rely on existing program annotations can be applied in our case since we actually need the invariant as a means to specify the loop and not the other way round.

B. Specifying Asynchronous Services

So far, we have made the implicit assumption that all service executions are synchronous: a service receives a request, the client waits for the service to handle the request and the service returns a response. However, it is very common in Service-Oriented Computing to employ services that interact in an asynchronous manner: the client invokes the service but does not block waiting for the response, Instead, the composition can carry on tasks independenttly from the invoked service. This affects the evaluation of postconditions because in the asynchronous case the response is received in a state which may differ from that in which the invocation was performed. Our proposal is to evaluate the postcondition for the asynchronous call in the same environment in which the invocation was made by referring to the values of the original variables at the point in which the invocation was made.

V. RELATED WORK

Formal specifications have been used in computer science to describe what a system should do. Hoare [7] introduced the well-known triple notation $P\{S\}Q$ which expresses that if preconditions P are met before initiating execution of program S, then when the execution completes postconditions Q will be true. Dijkstra [10] expanded on this by focusing on necessary and sufficient (called *weakest*) preconditions, that guarantee the desired result. The notation he introduced, $wp(S, Q)$ denotes the weakest precondition for program S, which is "the set of all states such that execution of S begun in anyone of them is guaranteed to terminate in a finite amount of time in a state satisfying Q" [8].

Our work attempts to bring the style of Dijkstra's derivation of program specifications to the field of Service-Oriented Computing, but differing in some relevant points: Dijkstra's derivation process is driven by the program im-

plementation, specifications are not approximations, and all interactions are synchronous.

Despite these differences, it is important to make sure that our approach does not contradict Dijkstra's. In other words, we need to ensure that our approach does not infer a precondition that is weaker than the one calculated by the wp operator. Let us reason by contradiction and assume that the precondition P produced by our approach for a sequential composition $A; B$ is **not** stronger or as strong as the weakest precondition P_{wp}: $\neg(P \Rightarrow P_{wp})$ — or, equivalently, $P \wedge \neg P_{wp}$. Then $P_A \wedge P_B \wedge \neg P_{wp}$ must also be true. From the discussion at the beginning of Section III, we know that the approximated preconditions P_A and P_B are stronger than or equivalent to the corresponding weakest preconditions P_{wp_A} and P_{wp_B}, i.e., $P_A \Rightarrow P_{wp_A}$ and $P_B \Rightarrow P_{wp_B}$. With that into account, we need $P_{wp_A} \wedge P_{wp_B} \wedge \neg P_{wp}$ to be true.

The result is contradictory since we want at the same time the precondition of a composite service to be false and the preconditions of the services it contains to be true. Hence our assumption was incorrect, meaning that our approach will never produce a precondition that is weakest than the one derived by the wp operator. Similarly, we can prove that our result holds for all control constructs handled in this work, since in almost all of them, P contains the conjunction of P_A and P_B. The only exception is the conditional case, however the contradiction we have proven still holds, since in any case either P_A or P_B will have to be true.

Another work related to specification derivation is that of Ghezzi et al. [11], [12] which focuses on methods for specification recovery. The authors propose a method to infer algebraic specifications given the related class and its methods and with no access to the source code. This work relies on the run-time behavior of a component in order to derive its specification, which is different from our approach, which relies on the specifications of components and the control flow between them.

VI. Conclusions and Future Work

In this paper, we proposed an approach for inferring composite service specifications given the specifications of the services participating in the composition and the composition schema, by constructing the specification using structural induction based on derivation rules defined for most fundamental control constructs. The nature of the proposed approach facilitates a possible implementation: structural induction lends itself to be written as a recursive algorithm. Hence, it would be straightforward to create an automated process that takes a set of service specifications and a composition schema and produces the specification for the composite service of the schema.

Future work includes implementing the proposed approach and evaluating it for compositions of varying complexity. Concerning specification simplification, we plan to look into the work of Douglas Smith [13] and determine whether the actions he proposes may be applied in our case. Finally, it would be interesting to explore whether the resulting specifications suffer from the frame problem and related issues as examined in [14].

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube). Manuel Carro has also been partially funded by Spanish MICINN project TIN-2008-05624 *DOVES* and CAM project S2009TIC-1465 *PROMETIDOS*.

References

[1] C. B. Jones, *Systematic Software Development Using VDM*, 2nd ed., ser. International Series in Computer Science, 1991.

[2] J. R. Abrial, *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[3] E. D. Nitto, V. Mazza, and A. Mocci, "CD-IA-2.2.2: Collection of industrial best practices, scenarios and business cases," S-Cube Network of Excellence, Tech. Rep., May 2009.

[4] J. Mendling, K. B. Lasse, and U. Zdun, "On the transformation of control flow between block-oriented and graph-oriented process modelling languages," *Int. J. of Bus. Proc. Integration and Management*, vol. 3, no. 2, pp. 96–108, 2008.

[5] W. McCune, "Prover9 and Mace4," 2005–2010, http://www.cs.unm.edu/~mccune/prover9/.

[6] N. Russell, A. ter Hofstede, W. van der Aalst, and N. Mulyar, "Workflow control-flow patterns: A revised view," BPM Center, Tech. Rep. BPM-06-22, June 2006.

[7] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[8] D. Gries, *The Science of Programming*. Springer, 1981.

[9] C. Furia and B. Meyer, "Inferring Loop Invariants Using Postconditions," in *Fields of Logic and Computation*, ser. LNCS, A. Blass, N. Dershowitz, and W. Reisig, Eds. Springer Verlag, 2010, vol. 6300, pp. 277–300.

[10] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. Volume 18 , Issue 8, pp. 453–457, August 1975.

[11] C. Ghezzi, A. Mocci, and M. Monga, "Efficient recovery of algebraic specifications for stateful components," in *IWPSE*, 2007, pp. 98–105.

[12] ——, "Synthesizing intensional behavior models by graph transformation," in *ICSE*, 2009, pp. 430–440.

[13] D. R. Smith, "Derived preconditions and their use in program synthesis." in *CADE*, ser. LNCS, vol. 138, 1982, pp. 172–193.

[14] G. Baryannis and D. Plexousakis, "The frame problem in web service specifications," in *Proc. 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, ser. PESOS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 9–12.