Static Verification of Control and Data in Web Service Compositions *

Raman Kazhamiakin
DIT, University of Trento
via Sommarive 14, 38050, Trento, Italy
raman@dit.unitn.it

Marco Pistore
DIT, University of Trento
via Sommarive 14, 38050, Trento, Italy
pistore@dit.unitn.it

Abstract

The data exchanged among the web services participating to a composition are clearly very relevant for a correct behavior of the composition. Nevertheless, most of the approaches existing in the literature for the static verification of web service compositions ignore data, or require very small ranges to be associated to the data types.

In this paper, we propose an approach for the verification of web service compositions that takes into account the data flows among the component process. The approach exploits abstraction techniques for modeling those aspects of data that are relevant for the correctness of the composition and hiding the aspects that are irrelevant. We show that building the right abstraction corresponds to defining those assumptions on the data manipulations performed by the component services which are crucial for the correctness of the composition.

1. Introduction

Web services provide the basis for the development and execution of business processes that are distributed over the network and available via standard interfaces and protocols [10]. Service composition [12] is one of the most promising ideas underlying Web services: new functionalities can be defined and implemented by combining and interacting with pre-existing services. Different standards and languages have been proposed to develop Web service compositions. Among them Business Process Execution Language for Web Services (BPEL, [1]) is one of the emerging standards for describing a key aspect for the composition of Web services: the behavior of the services. This opens up the possibility of applying a range of formal techniques to the analysis of the composition behavior, and different approaches have been defined for addressing this problem, see e.g. [6, 13, 14, 7, 16].

In this paper, we focus on the fact that, along with the definition of the control flow, Web service compositions define also a data flow among the component services. While the former is used to specify the order of the activities carried out by the participants, the latter defines the exchanged data values. Clearly, the data flow is as important for the static analysis of Web service compositions as the control flow, but most of the existing approaches ignore data, thus potentially invalidating the analysis results.

The necessity to manage data, however, brings the problem that the data domains operated by the Web service specifications are often infinite (e.g., user-defined type in XML documents), and the semantics of data manipulating operations is complex (e.g., the XPath expressions used in BPEL specifications). This restricts the applicability of traditional formal analysis techniques that rely on simple and finite system representations. Abstraction techniques such as the ones defined in [9, 3] are needed to take into account only those data-related properties that are relevant for the verification, and discard the aspects that are irrelevant.

The abstraction-based techniques are widely used in the traditional domains, like program verification. Differently from these domains, Web service-based systems exhibit a specific feature that should be taken into account. The coarse granularity and distributed nature of the Web service (composition) specifications implies that the internal details of the implementations of the component services may be hidden to the analysts. This lack of knowledge is useful as far as it allows for a higher-level definition of composite business processes which is independent from the implementation details of the components. However, it may also make it impossible to prove the correctness of the composition. In this case, the system model should be refined by introducing those assumptions on the internal of service implementations that are necessary to guarantee a correct composition.

We see the analysis of Web service compositions in presence of data as an iterative process, where the verifica-

*This work is partially funded by the MIUR-FIRB project RBNE0195K5, "KLASE", by the MIUR-PRIN 2004 project "STRAP", and by the EU-IST project FP6-016004 "SENSORIA".

tion of expected properties is interleaved with the elicitation of the assumptions on the data flow that ensure these properties. At the end, this will produce a refined model, where the specification is enriched with a set of assumptions and constraints that are crucial for the system correctness, and where all the expected properties have been formally verified. We remark that while these assumptions and constraints are discovered and collected during the static, design-time verification of the compositions, they may be further exploited for the dynamic, run-time analysis of the compositions using monitoring techniques.

In this paper we present a theoretical framework for the abstraction-based verification of Web service compositions that supports the iterative analysis process. In this framework we address the following goals:

- we provide a foundation for the modeling and analysis of data in compositions through abstractions;
- we provide the ability to specify and verify different behavioral properties in this setting;
- we manage the incompleteness in the composition specification allowing the definition of assumptions on the hidden properties of the systems;
- we support an iterative refinement process by combining the abstraction-based verification with the requirements extraction.

We also present a preliminary experimental results on the prototype implementation of the introduced framework.

The structure of the paper is as follows. In Section 2 we present a case study describing the problems of data flow analysis in Web service compositions. Section 3 presents a formalism used to model Web service composition taking into account both control and data flow. In Section 4 we introduce the problem of the data abstraction and the models of abstraction we use for the verification. The analysis approach, its implementation and some experimental results justifying its applicability are described in Section 5. Conclusions and future works are presented in Section 6.

2. Data Flow in Web Service Compositions

In order to illustrate the data-related problems of composition analysis we consider a (modified) Loan Approval case study. The nominal scenario may be summarized as follows. The user requests a loan approval specifying a desired amount and the Loan Approval (LA) process is aimed to iteratively look for an amount that may be safely approved for the user. Each iteration is performed as follows. The current amount is being checked. If the amount is low, then the Assessor service is called, otherwise the Approver service is used. If the Assessor considers the amount to be risky, then it is also passed to the Approver, otherwise the amount is approved and the loop terminates. If the Approver provides a negative answer, the amount is reduced and the iteration continues. We model the composition as a set of BPEL specifications that represent the behaviors of every participant. A conceptual model of the composition scenario is represented in Fig. 1.

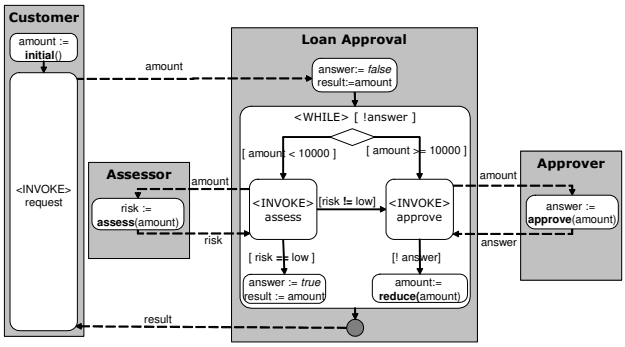


Figure 1. Loan Approval Example

Ignoring data flow. Ignoring the data in this example leads to the following problems in the analysis.

First, scenarios are possible that never happen in a real system. Indeed, the body of the loop is implemented as a `flow` construct with the synchronization links between activities. As soon as the link condition evaluates to true, the target activity is ready to start. Since the link conditions leaving the same activity are mutually exclusive, both target activities can not be fired and the behavior is correct. However, if the data is ignored, both Assessor and Approver services may be invoked and the approval fails. Moreover, if we verify the property that "*it is possible for the customer to make a request that LA accepts without involving Approver*" (which is expected to be possible for small amounts), the data-less analysis would say that the property does not hold. Another property that is violated when the data is ignored is the following: "*for any request the LA service should provide a result*".

Second, the data-less analysis may hide potential problems that can be revealed only in the presence of data. In our example, if the answer of the Approver service is positive, the approved amount should be returned to Customer. In the specification, however, the value of the `result` variable is returned instead. It is easy to see that the wrong value may be returned to Customer, since this variable is assigned only initially, and is not changed during the execution of the loop. In order to fix this problem an operation that assigns the value of the `amount` variable to the value of the `result` variable has to be performed upon receiving the answer from the Approver service as it is done if the loan

is approved without intervention of the Approver in case of low risk. Clearly, this problem may not be revealed if the data is ignored.

Abstractions-based representation. The presented case study may not be directly verified using traditional formal techniques. Indeed, the data variables, being manipulated in this scenario have infinite domains, such as strings, integers, structures, and aggregations of them. Therefore a finite representation of the system is not possible. A predicate abstraction approach [9] allows to avoid this problem by considering only relevant data flow properties and ignoring the others. For instance, in the above case study it is not needed to enumerate all the possible amounts that the user may request. Instead, it might be sufficient to evaluate the fact that the amount is big, which can be represented with one proposition.

The abstraction, however, is not always able to faithfully describe the real system. Indeed, the abstract model may *over-approximate* the real system, that is may contain some unrealizable behaviors. If one is interested to check that every behavior satisfies a property (*universal* property), then the violation of the property in an over-approximated abstraction does not imply the violation in a real system. Analogously, if the verification of the *existential* property (i.e. some behavior satisfies the property) is violated in *under-approximation* (i.e. the abstract system that contains less behaviors then the real one), it is not necessarily violated in the real system. This will be taken into account in the analysis framework through an interplay of over- and under-approximations during the verification process.

Incompleteness of composition specifications. The specification of Web service composition often hides certain detail on the internal service implementations, complicating the analysis of the specification. Indeed, if the internal algorithms implemented by the Approver and Assessor services are not known, then the abstract models of the composition do not contain enough knowledge to infer validity of many properties. In particular, it would be impossible to check the possibility to eventually provide an answer, or that any scenario that starts with a small amount leads to an approval of this amount.

In order to eliminate this lack of knowledge, the model should be enriched with assumptions on the internal implementation that should imply the correctness of the behavior. In other words, if we consider these internal operations as *uninterpreted functions* then the assumptions may be thought of as the preconditions/effects rules on the function values. For example, if Assessor service is expected to return low risk whenever the amount is small than 100 euro, the LA process is guaranteed to terminate correctly for such small requests.

In the following sections we present a formal approach that (i) allows for the definition of abstract composition model, (ii) enables the specification and verification of both universal and existential properties, and (iii) supports the definition and analysis of data flow properties reflecting assumptions on service internals.

3. Web Service Composition Model

Here we define a formal model for representing and analyzing Web service compositions. We use a *ground model* to represent data, operations on data and data flow of the system; and (a composition of) *state transition systems* to represent a control flow and evolution of the Web service composition.

3.1. Ground Model

The formal model of the executions of a composition is given in terms of ground context.

Definition 1 (Ground Context) *A ground context $\mathcal{C}$ is a tuple $\langle \mathcal{T}, \mathcal{V}, \mathcal{F} \rangle$, where $\mathcal{T}$ is a finite set of (possibly infinite) types; $\mathcal{V}$ is a finite set of typed variables; $\mathcal{F}$ is a finite set of typed functions.*

Let e denote an expression and E a set of expressions. We use t to denote a term, and T to denote the set of terms. We also write x for a variable in $\mathcal{V}$, and f for a function in $\mathcal{F}$. The syntax of expression is as follows[1]:

- $E \equiv (t_1 = t_2) \mid \neg e \mid (e_1 \vee e_2)$
- $T \equiv x \mid f(t_1, \ldots, t_n)$

As one can see, this model is generic enough to define the management of the data in arbitrary Web service composition represented as a set of interacting BPEL processes. Indeed, it allows to define arbitrary data types, variables, XPath expressions and predicates. In particular, a BPEL *condition* $\phi \in \Phi$ is an expression, while an *assignment* $\omega \in \Omega$ has the form $(x := t)$.

We assume a fixed interpretation of typed functions, and define a ground state as a complete assignment over the set of typed variables $\mathcal{V}$.

Definition 2 (Ground State) *Given a ground context $\mathcal{C} = \langle \mathcal{T}, \mathcal{V}, \mathcal{F} \rangle$, we define a* ground state *g as a finite set of pairs $\langle x, v \rangle$ s.t. for all $x \in \mathcal{V}$ there exists a unique $\langle x, v \rangle \in g$, where v is a value belonging to the type of x.*

[1] We remark that in this model the constants may be represented as functions with zero parameters. Analogously, boolean variables and predicates may be also interpreted as abstract variables and functions.

Definition 3 (Evaluation Function) *Given a ground context $\mathcal{C} = \langle \mathcal{T}, \mathcal{V}, \mathcal{F} \rangle$, we define the* evaluation function Γ_g *as the function that given a term t or an expression e returns the result of the computation with respect to a ground state g:*

- $\Gamma_g(x) = v$, *where x is a variable and $\langle x, v \rangle \in g$;*
- $\Gamma_g(f(t_1, \ldots, t_n)) = f(\Gamma_g(t_1), \ldots, \Gamma_g(t_n))$;
- $\Gamma_g(t_1 = t_2) = true$ *iff* $\Gamma_g(t_1) = \Gamma_g(t_2)$;
- $\Gamma_g(\neg e) = \neg \Gamma_g(e)$;
- $\Gamma_g(e_1 \vee e_2) = \Gamma_g(e_1) \vee \Gamma_g(e_2)$.

Definition 4 (Ground Update) *The update of a ground state g with an assignment $\omega = (x := t)$, denoted as $update(g, \omega)$, is the state g' where $\langle x', v \rangle \in g'$ if $\langle x', v \rangle \in g$ and $x \neq x'$, or if $x' = x$ and $v = \Gamma_g(t)$.*

3.2. BPEL as State Transition System

In our framework, we encode BPEL processes as *state transition systems* which describe dynamic systems that can be in one of their possible *states* (some of which are marked as *initial states*) and can evolve to new states performing some *actions*. Without loss of generality, any BPEL operation may be considered as a transition where some interaction takes place (message reception or emission), or as a transition where some internal operation is performed. Each transition may have an applicability condition that defines whether the operation may be executed, and a set of assignments that define the modifications to be applied on the process variables. Following the standard approach in process algebras, we model process evolution using *input actions*, representing the message reception, *output actions*, representing the message emission, and *internal actions*, representing internal evolutions that are not visible to external services. A state s of the STS Σ is a pair $\langle pc(s), g(s) \rangle$, where $pc(s)$ is a value of the program counter, and $g(s)$ is a ground state.

Definition 5 (STS) *STS is a 5-tuple $\Sigma = \langle \mathcal{V}, \mathcal{S}, \mathcal{S}^0, \mathcal{A}, \mathcal{R} \rangle$, where $\mathcal{V}$ is a set of local typed variables; $\mathcal{S}$ is a finite set of states, and $\mathcal{S}^0$ is a set of initial states; $\mathcal{A}$ is a finite set of actions; $\mathcal{R} \subseteq \mathcal{S} \times \Phi \times \mathcal{A} \times \Omega^* \times \mathcal{S}$ is a transition relation.*

A transition $(s, \phi, \alpha, \{\omega_i\}, s') \in \mathcal{R}$ defines modification of the program counter $pc(s)$ to $pc(s')$ and executes an action α. It is executable if the condition ϕ evaluates to true in the ground state $g(s)$, and the ground state $g(s')$ is an update of $g(s)$ w.r.t. set of assignments: $update(g, \omega_i)$. Web service composition is defined as a *composition of STS*.

Definition 6 (STS Composition)
Composition of 2 STSs Σ_1 and Σ_2, denoted as $\Sigma_1 \times \Sigma_2$, is the STS $\langle \mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{S}_1 \times \mathcal{S}_2, \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{R} \rangle$, such that $((s_1, s_2), \phi, \alpha, \Omega, (s_1', s_2')) \in \mathcal{R}$ iff:

- $(s_1, \phi, \alpha, \Omega, s_1') \in \mathcal{R}_1$, $\alpha \notin \mathcal{A}_2$, *and* $s_2' = s_2$, *or*
- $(s_2, \phi, \alpha, \Omega, s_2') \in \mathcal{R}_2$, $\alpha \notin \mathcal{A}_1$, *and* $s_1' = s_1$, *or*
- $(s_1, \phi_1, \alpha, \Omega_1, s_1') \in \mathcal{R}_1$, $(s_2, \phi_2, \alpha, \Omega_2, s_2') \in \mathcal{R}_2$, $\alpha \in \mathcal{A}_1$, $\alpha \in \mathcal{A}_2$, $\phi = \phi_1 \wedge \phi_2$, *and* $\Omega = \Omega_1 \cup \Omega_2$.

A *run* of the STS is a (possibly infinite) sequence of global states $\pi = s_0, \alpha_0, s_1, \ldots$, s.t. $s_0 \in \mathcal{S}^0$, and for each $i \geq 0$, there is a transition $(s_i, \phi, \alpha_i, \Omega, s_{i+1}) \in \mathcal{R}$, s.t. $\Gamma_{g(s_i)}(\phi) = true$, and $g(s_{i+1})$ is defined as $update(s_i, \Omega)$.

3.3. STS Composition Verification

We exploit a temporal logics of linear time (LTL) for the specification of the properties that an STS composition is expected to satisfy. The logic allows one to define properties of the system executions, i.e. runs.

Let us have a set of propositions $\mathcal{P}$. This set may include facts about process variables, functions, predicates, program counters, actions, etc. Let p range over P. LTL formula is defined as follows:

$$\phi \equiv p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \circ\,\phi \mid \phi_1 U \phi_2 \mid \Diamond\phi \mid \Box\phi$$

Here the operation $\circ\phi$ denotes that the formula ϕ is to be satisfied in the next state of the run; $\phi_1 U \phi_2$ declares that ϕ_1 should hold in every state of the run until a state where it satisfies ϕ_2. $\Diamond\phi$ and $\Box\phi$ denote that the formula should be eventually satisfied and globally satisfied (i.e. in every state) respectively.

Given an LTL formula ϕ, $\pi \models \phi$ means that the run π of the composition Σ satisfies the formula. We define the satisfaction of universal properties (hereafter *assertions*) and existential properties (hereafter *possibilities*).

Definition 7 (Assertion and Possibility Satisfiability)
Σ satisfies assertion ϕ, written as $\Sigma \models_A \phi$, iff for every run π of Σ, $\pi \models \phi$.

Σ satisfies possibility ϕ, written as $\Sigma \models_P \phi$, iff there exists a run π of Σ, s.t. $\pi \models \phi$.

In this way, the example requirement "it is possible that LA accepts the requested amount without involving Approver", may be represented as the following possibility:

$$\Diamond(reply(\text{request}) \wedge Cust.result = Cust.request) \wedge \Box\neg invoke(\text{approve})$$

Analogously, the necessity to provide a result for every request may be represented as the following assertion:

$$\Box(invoke(\text{request}) \rightarrow \Diamond\, reply(\text{request}))$$

4. Data Abstraction Model

The verification of the STS composition is not doable in general due to the fact that the data types are infinite and the functions are potentially too complex to reason on. In order to be able to perform a finite state verification, we have to provide an *abstraction* of the underlying composition model that is finitely representable and allows to obtain certain answers for the verification queries. We now define two models for abstraction, namely a *knowledge-based* model (K-model) and a *branching-based* model (B-model).

4.1. Abstract Propositions

Both models we introduce in this work are *parametric* with respect to the set of propositions being considered in the analysis. These propositions may express facts about the composition states, variables, relations between them, function values, etc. The propositions have the form of the data expressions according to the definition presented in Section 3.1.

For the abstraction of the composition only a subset of all the possible propositions is considered. The selection of this subset defines the parametrization of the abstraction. When the set of considered propositions increases, the abstraction gets closer to the real system. This, however, increases also the computational cost of the verification. Therefore, the parameterization allows one to drive the verification process, starting from a small set and adding new propositions only when a refinement is needed. In the following we will denote the subset of the considered propositions as $\mathcal{P}^A$. We expect that the verification properties are defined only over the propositions in $\mathcal{P}^A$.

4.2. Abstract Composition Model

We define an abstract STS composition model corresponding to a concrete one, based on a set of abstract propositions. Given a set of propositions $\mathcal{P}^A$, a *valuation* is simply a mapping from $\mathcal{P}^A$ to $\{true, false\}$. We denote the valuation as V. We call the set of ground states compatible with the valuation the *interpretation* of the valuation, denoted as $\mathcal{I}(V)$. We say that the valuation is *consistent*, written as $consistent(V)$, if and only if $\mathcal{I}(V) \neq \emptyset$.

Analogously to the concrete model, the abstract model defines the evolution of the system. Abstract state $\hat{s}$ in this model is a pair $\langle pc, V\rangle$, where pc is a value of the program counter as before, and V is the valuation of propositions. We say that a transition t may be performed in an abstract state, if it is *applicable* with respect to the proposition valuation in this state, written as $applicable[t](V)$. The *effect* of the transition defines the modification of the proposition valuation, denoted as $exec[t](V)$.

We now give a definition of the abstract STS composition corresponding to a concrete one. This definition is parametric with respect to the interpretation of the valuation of the propositions, and to the way the applicability and the effect of the transition are defined, which differ for the two abstraction models.

Definition 8 (Abstract STS)
Given the STS $\Sigma = \langle \mathcal{V}, \mathcal{S}, \mathcal{S}^0, \mathcal{A}, \mathcal{R}\rangle$*, the corresponding* abstract STS*, denoted as* $\hat{\Sigma}$*, is a 5-tuple* $\langle \mathcal{P}^A, \hat{\mathcal{S}}, \hat{\mathcal{S}^0}, \mathcal{A}, \hat{\mathcal{R}}\rangle$*, where* $\hat{\mathcal{S}}$ *is a set of abstract states, and* $\hat{\mathcal{S}^0}$ *is a set of initial abstract states;* $\hat{\mathcal{R}} \subseteq \hat{\mathcal{S}} \times \mathcal{A} \times \hat{\mathcal{S}}$ *is an abstract transition relation such that* $((pc, V), \alpha, (pc', V')) \in \hat{\mathcal{R}}$ *iff*

- $\exists\, t = (s, \phi, \alpha, \Omega, s') \in \mathcal{R}$,
- $pc = pc(s)$, $g(s) \in \mathcal{I}(V)$,
- $pc' = pc(s')$, $g(s') \in \mathcal{I}(V')$,
- $applicable[t](V)$, *and* $V' = exec[t](V)$.

A run of the abstract composition Σ is defined in the usual way.

4.3. K-Model of Abstraction

In our framework we exploit this model to show that in *any* ground model corresponding to the abstract one there *exists* a run satisfying a certain property, i.e. to verify the existential properties. For this purposes we try to build an abstraction in such a way that it defines the most "pessimistic" assumption on the composition.

In the K-model an abstract state of the composition is represented at the "knowledge level". In other terms, a certain fact about the abstract variable values may be known to be true, or unknown. If the valuation of a certain fact is *true*, then it is it is also true in all the ground state represented by the valuation. Thus, the fact is "known" to be true. On the contrary, if the valuation is *false*, the fact is "unknown" to be true, and may be true or false in some ground state.

Definition 9 (K-Interpretation) *Given a valuation* V*, its* K-interpretation*, denoted as* $\mathcal{I}^K(V)$*, is the set of ground states* $g \in \mathcal{I}^K(V)$ *s.t.* $\forall\, p \in \mathcal{P}^A$*, if* $V(p) = true$ *then* $\Gamma_g(p) = true$.

The K-model defines a "pessimistic" view of the abstraction by implementing the applicability and the execution of the transition relation as follows. We say, that the transition is applicable in the abstract state, if it is applicable in *every* corresponding ground state. Analogously, in the K-model the effect of the transition execution is the most conservative with respect to the set of facts that can be deduced.

Definition 10 (Applicability and execution in K-Model) *A transition $t = (s, \phi, \alpha, \Omega, s')$ is* applicable *in V, written as $applicable[t](V)^K$, iff $\forall\, g \in \mathcal{I}(V)$, $\Gamma_g(\phi) = true$.*

The execution *of t on V, denoted as $exec[t](V)^K$, is a valuation V' s.t. $V'(p) = true$ iff $\forall\ g \in \mathcal{I}^K(V)$, $\Gamma_{update(g,\Omega)}(p) = true$.*

For the abstraction under K-model we also assume that set of initial states is a singleton with $V(p) = \mathit{false}$, for each $p \in \mathcal{P}^A$ (no facts are known a priori).

Under the applicability condition defined as above, a situation is possible where the execution of the abstract model reaches a state with branching, and no transition is allowed since neither the condition or its negation is "known" to be true. Such an execution is to be discarded in the verification since the concrete system can not terminate in such a state. Consider, for instance, the Loan Approval example. Since there is no information on the implementation of the Assessor service, the result of the invocation of `assess` function is undefined, and the valuation of the proposition $risk = low$ (as well as its negation) is "unknown". The process execution is blocked in this state and has to be discarded.

Let us define a verification problem of K-model of STS composition. Given a formula ϕ, its K-interpretation ϕ^K is a formula obtained from ϕ by replacing each proposition from $\mathcal{P}^A$ with the corresponding boolean variable.

We say that the K-model of STS composition satisfies the possibility ϕ if and only if there is a run of the model that satisfies ϕ^K. Analogously, we say that this model satisfies the assertion ϕ if there is no run that satisfies $(\neg\phi)^K$.

Definition 11 (Satisfiability in K-Model)
The possibility ϕ is satisfied in K-model, written as $\Sigma^K \models_P \phi$, iff there exists a run π^K of Σ^K, s.t. $\pi^K \models \phi^K$.

The assertion ϕ is satisfied in K-model, written as $\Sigma^K \models_A \phi$, iff for each π^K of Σ^K, $\pi^K \not\models (\neg\phi)^K$.

The following results immediately follow from the definition of the K-model. For the lack of space, we omit the formal proofs in this paper.

Theorem 1
Given an assertion ϕ, if $\Sigma^K \not\models_A \phi$, then $\Sigma \not\models_A \phi$.
Given a possibility ϕ, if $\Sigma^K \models_P \phi$, then $\Sigma \models_P \phi$.

4.4. B-Model of Abstraction

The B-model is aimed to express an "optimistic" view of the executions that a system can perform. Whenever a valuation of some fact can not be determined, both true and false values are assumed. In other words, the abstract state is split in this case in two sets, with the true value in one of them and false in another. Therefore, if the valuation of the proposition is false, the proposition is also false in all the ground states of $\mathcal{I}(V)$.

Definition 12 (B-Interpretation) *Given a valuation V, its* B-interpretation*, denoted as $\mathcal{I}^B(V)$, is a set of ground states g such that for each $p \in \mathcal{P}^A$, $V(p) = \Gamma_g(p)$.*

The "optimistic" approach requires that a transition is applicable in the valuation V, if V is not in conflict with the transition condition. The effect of the transition in the B-model is defined by the set of the valuations that are compatible with the effects of the transition.

Definition 13 (Applicability and execution in B-Model)
A transition $t = (s, \phi, \alpha, \Omega, s')$ is applicable *in V, written as $applicable[t](V)^B$, iff $\exists\, g \in \mathcal{I}(V)$ s.t. $\Gamma_g(\phi) = true$.*
The execution *of t on V, denoted as $exec[t](V)^B$, is a valuation from the set $\{V' \mid \exists g \in \mathcal{I}(V)$, s.t. $\forall\ p.\ V'(p) = \Gamma_{update_G(g,\omega)}(p)\}$.*

The B-model contains more states and runs than the ground model: indeed, when a variable gets the value of some abstract expression, any possible consistent valuation should be considered as an effect of the assignment. As a result, the transition leads to different states, some of which may be unreachable in a real execution. In our example, if the implementation of the $reduce$ function is not defined, we should consider any possible outcome of the assignment, including the case where the amount is not changed. Hence, an infinite loop is possible, where the Approver rejects the request and the amount does not change, which does not appear in real execution.

As a consequence, the B-model is not applicable for the verification of existential properties. If a run satisfying the property is contained in the B-model, it is not guaranteed to appear in the ground one. On the contrary, if the verification of an assertion is considered, the satisfaction of this property in the B-model implies also the satisfaction in the ground model.

Theorem 2
Given an assertion ϕ, if $\Sigma^B \models_A \phi$, then $\Sigma \models_A \phi$,
Given a possibility ϕ, if $\Sigma^B \not\models_P \phi$, then $\Sigma \not\models_P \phi$.

4.5. Relations between Abstraction Models

The following theorem summarizes the relations between the satisfaction of the property under the two abstraction models.

Theorem 3 *Given an assertion ϕ,*

$$\Sigma^B \models_A \phi \ \Rightarrow\ \Sigma \models_A \phi \ \Rightarrow\ \Sigma^K \models_A \phi.$$

Given a possibility ϕ,

$$\Sigma^K \models_P \phi \ \Rightarrow\ \Sigma \models_P \phi \ \Rightarrow\ \Sigma^B \models_P \phi.$$

The theorem suggests that the interplay of the two models of abstractions is used in order to put bounds on the satisfiability of the property. While these models can not provide an exact answer for the property verification under an arbitrary composition model (the problem is undecidable in general), they are still able to return safe answers for such a problem. Moreover, they are able to provide exact answers for a wide range of the infinite state space systems.

This theorem also suggests the following verification algorithm. Given an assertion property, we check it under B-model. If the property is satisfied, it is satisfied also under the ground model, and *true* is returned. If the property is violated by B-model, we check if it is also violated by the K-model. If it is the case, then it is also violated in the ground model, return *false*. Otherwise, he satisfiability is not defined and further refinement is needed.

In the case of a possibility, we first check it under K-model. If it is satisfied, it is indeed satisfied by the ground model and *true* is returned. Otherwise, we check its violation under the B-model. If the possibility is also violated by this model, return *false*. Again, if neither of these holds, the satisfiability is not defined and we have to refine the model.

5. Analysis Framework

The data flow analysis framework used for the verification of Web service compositions may be described as follows. Given a set of properties of interest and a composition specification, two abstract models are being constructed. This construction is parametric with respect to the set of propositions and assumptions used for the analysis. Given the two abstract models, the property is being verified based on Theorem 3. If the answer is incomplete, the abstractions are refined by adding new propositions and/or by introducing additional assumptions on the functions that represent internal operations of the services.

5.1. Elicitation of Functions Requirements

As we discussed previously, the functions (or predicates) used to abstract internal service operations implementations are uninterpreted by default. This has the following consequences on the abstract models we defined above. In the B-model of abstraction, the value of the function may be arbitrary. On the contrary, in the K-model this non-determinism means that no knowledge may be extracted on the function call. Consider for instance, the Loan Approval example and the function `assess(amount)` implemented by the Assessor service. Given a call `risk:=assess(amount)`, its effect on the proposition `risk=low` in the B-model leads to a state where it is true and to a state where it is false, while in the K-model the effect of the call leads to a state where the value of the proposition (as well as its negation) is unknown.

This non-determinism may be refined by adding assumptions on the functions. This assumptions have the form of expressions relating the values of function parameters and returned values of the function. For instance, in our example the following assumption on the function `assess` may be added in order to make the above possibility true:

$$((amount < 100) \leftrightarrow (assess(amount) = low))$$

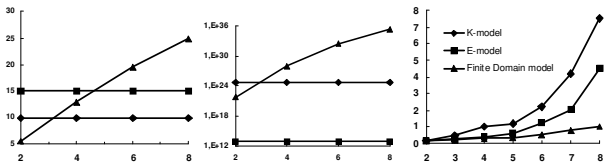
These constraints are being added to the abstract models and the verification is redone.

5.2. Implementation and Experiments

We implemented the presented analysis framework as a prototype tool on top of the Astro toolkit (available at `http://astroproject.org`). We exploited the logic of *equalities and uninterpreted functions* (UIF, [2]) for the computation of abstract models. This computation is performed by a parametric reasoner that, given a composition specification, a set of abstract predicated, and possibly a set of function constraints, generates two abstract models of the composition, namely B-model and K-model. These models define a control flow of Web service composition in terms of states and transitions, and a data flow in terms of boolean variables corresponding to equalities between terms (e.g. variables, functions, and constants). In case of B-model the resulting specification contains a set of axioms, specifying rules for equalities consistency, functions rigidness, etc. In case of K-model these axioms are used to compute a predicate valuation that defines a transition execution effect.

We evaluate this approach on series of experiments. First, we applied it to the analysis of (extended) Loan Approval case study. We used this example to compare the performance of the verification in the presented framework versus the verification under finite data domains (like, for instance, in [7]). We used 35 abstract propositions (and their negations) for 19 process variables and constants of 3 data types. The comparison of the verification times (in seconds) and of the state spaces are represented in Fig. 2(a) and Fig. 2(b) respectively. The horizontal axis of the graphs shows the number of domain values per type. In the finite domains approach the verification time (and space) grows when the data domains increase. On the contrary, the verification performance in our abstraction-based approach does not change.

Second, in order to show the scalability of the proposed approach, we conducted a set of experiments on another case study with the increasing number of processes (see [11] for the details). We compared the model extraction/verification time with that for the analysis on (small) finite domains. The results of the experiments are presented



(a) (b) (c)

Figure 2. Experimental results

in Fig. 2(c). The experiments demonstrate that the extraction of the K-model of a composition is less efficient with respect to the B-model. This happens due to the necessity to use more sophisticated reasoning techniques to extract the transition executions effects. We remark that the experimental results are preliminary and may be further improved using e.g. SAT-based reasoning techniques [5].

6. Conclusions

In this paper we present a framework that allows for the analysis of Web service compositions taking into account data flow aspects of the analyzed system. This framework allows not only to verify the composition behavior, but also to iteratively extract vital data-related properties and requirements hidden by the internal implementations of the Web services. The analysis is based on a formal model that exploits data abstraction techniques for the verification of composition behavioral. The proposed approach allows for the analysis of arbitrary and potentially infinite data types in a unified and efficient way. Two models of abstractions are defined in order to carry out the verification of both existential and universal properties, and to guide the refinement process. The presented experiments show the practical applicability of the proposed solution.

The problem of analysis of Web service compositions is investigated in the works of [6, 7, 13, 15, 14]. However, the data flow properties and their impact on the composition behavior are either not addressed [6, 13, 14, 15], or managed in a restricted way, based e.g. on finite instantiations [7]. On the contrary, we propose approach that allows to correctly analyse the Web service compositions, abstracting from the data types and their domains.

The presented work follows the approaches that address the problem of software verification based on predicate abstractions [9, 3, 4]. In particular, the work of [4] present a parametric iterative approach for the verification of C programs that is similar to ours. However, [4] is not on the verification of Web service compositions and addresses the analysis of universal properties only.

The approach of [8] allows for the analysis of both universal and existential properties exploiting 3-valued logics for this purposes. In our work we attempt to combine the key features of these techniques providing from the one side better precision and flexibility, and the ability to correctly reason also on existential properties in unified way.

As a future work we would like to further investigate the possible improvements of the abstractions-based reasoning. In particular, we are interested in applicability of SAT-based reasoners [5] for the purposes of abstraction generations. We are also interested in applications of the data flow analysis results for the runtime monitoring of Web service executions.

References

[1] T. Andrews, F. Curbera, H. Dolakia, J. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeravarana. Business Process Execution Language for Web Services (version 1.1), 2003.

[2] J. Burch and D. Dill. Automated verification of pipelined microprocessor control. In *Proc. CAV'94*, 1994.

[3] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. In *Proc. FMSD'04*, 2004.

[4] E. Clarke, S. Chaki, N. Sharygina, and N. Sinha. Dynamic Component Substitutability Analysis. In *Proc. FM'05*, 2005.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 1960.

[6] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. ASE'03*, 2003.

[7] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. WWW'04*, 2004.

[8] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-Based Model Checking Using Modal Transition Systems. In *Proc. CONCUR'01*, 2001.

[9] S. Graf and H. Saidi. Construction of abstract state graph with PVS. In *Proc. CAV'97*, 1997.

[10] S. Graham, S. Simenov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI*. Sams, 2001.

[11] R. Kazhamiakin, M. Pistore, and L. Santuari. A Parametric Communication Model for the Verification of Web Service Compositions. In *Proc. WWW'06*, 2006.

[12] R. Khalaf, N. Mukhi, and S. Weeravarana. Service Oriented Composition in BPEL4WS. In *Proc. WWW'04*, 2004.

[13] S. Nakajima. Model-checking verification for reliable web service. In *Proc. OOPSLA'02 Workshop on OOWS*, 2002.

[14] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW'02*, 2002.

[15] C. Ouyang, H. Verbeek, W. van der Aalst, S. Breutel, M. Dumas, and A. ter Hofstede. WofBPEL: A tool for automated analysis of BPEL processes. In *Proc. ICSOC'05*, 2005.

[16] M. Pistore, M. Roveri, and P. Busetta. Requirements-Driven Verification of Web Services. In *Proc. WS-FM'04*, 2004.