Efficient Service Composition Using Zero-Suppressed Reduced Ordered Binary Decision Diagrams

Walter Binder
Faculty of Informatics
University of Lugano
Lugano, Switzerland
Email: walter.binder@unisi.ch

Ion Constantinescu and Boi Faltings
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Artificial Intelligence Laboratory
CH–1015 Lausanne, Switzerland
Email: {ion.constantinescu, boi.faltings}@epfl.ch

***Abstract*— Recent algorithms for automated service composition issue many complex queries to service directories. As service directories are shared resources, they may become performance bottlenecks. In order to increase scalability, we introduce a compact directory digest, which is distributed to clients and includes all information needed for automated service composition. Therefore, complex directory queries during service composition can be avoided. We encode a directory digest as a Zero-Suppressed Reduced Ordered Binary Decision Diagram (ZDD). In several steps, we refine a simple service composition algorithm in order to leverage the ZDD representation. Introducing specialized ZDD operations, we achieve a service composition algorithm that scales very well with an increasing size of the directory digest.**[1]

I. Introduction

Service-oriented computing enables the construction of distributed applications by integrating services that are available over the web [1]. The building blocks of such applications are web services[2] that are accessed using standard protocols.

Service discovery is the process of locating providers advertising services that can satisfy a given service request. Automated service composition addresses the problem of assembling individual services based on their functional specifications in order to create a value-added, composite service that fulfills a service request. Most approaches to automated service composition are based on AI planning techniques [2]–[4]. They assume that all relevant service advertisements are initially loaded into a reasoning engine.

However, due to the large number of service advertisements and to the loose coupling between service providers and consumers, services are indexed in service directories. As loading a large directory of service advertisements into a reasoning engine is not practical, planning algorithms have been modified in order to dynamically retrieve relevant service advertisements from a service directory during composition [5], [6].

In such an approach, a single service composition may involve several complex directory queries, and each query may have to process a significant part of the directory. E.g., in the case of service composition algorithms using forward chaining, a single directory query may require the processing of up to 20% of the directory data, even though the directory uses an optimized index structure [6].[3] As the service directory is a shared resource, it is likely to become a performance bottleneck. Massive replication of the service directory is needed for scalability, which is expensive due to the large number of needed directory servers.

In order to avoid complex directory queries, we promote a compact directory digest that includes all information needed for automated service composition [7]. The service composition clients download the digest and use it to solve the hard part of the composition problem locally. Only simple directory queries are issued to obtain the final result.

The scientific contributions of this paper are: (1) We present a concrete encoding of the directory digest as Zero-Suppressed Reduced Ordered Binary Decision Diagram (ZDD) [8]. (2) We show that the ZDD representation is compact, even for a large-scale directory. (3) We develop and evaluate service composition algorithms that leverage the ZDD datastructure. (4) We introduce specialized ZDD operations that enable highly scalable service composition.

This paper is structured as follows: Section II presents a simplified service description formalism and our definition of service composition. Section III gives an overview of service composition using a directory digest. Section IV reviews ZDDs and explains our ZDD representation of the directory digest. Section V presents our ZDD-based service composition algorithms and introduces specialized ZDD operations to achieve high performance and scalability. Section VI evaluates the size of the ZDD representation, as well as performance and scalability of our service composition algorithms. Finally, Section VII concludes this paper.

[1]This work was conducted while the first author was affiliated with the Artificial Intelligence Laboratory of the Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.

[2]In this paper, *service* stands for *web service*.

[3]A naive directory implementation may have to process the whole directory contents in order to discover all relevant service advertisements.

II. Definitions and Basic Service Composition Algorithm

Service descriptions are a key element for service discovery and service composition, as they enable automated interactions between applications. We distinguish between an invokable *service instance* (including service grounding) and its *service signature*. A service signature specifies the input/output parameters of one or more service instances.

We describe a service signature S by two parameter sets – the *required input parameters* $in(S)$ and the *generated output parameters* $out(S)$. Each parameter is identified by a unique name in its set. We assume that the parameter name defines the semantics of the parameter. Despite its simplicity, our formalism is consistent with existing service description formalisms, such as WSDL [9] and OWL-S [10]. Part of the input/output specification of services described in WSDL or OWL-S can be mapped to our simplified formalism.

A *service directory* stores *service advertisements*, describing features of service instances available over the web. Each service advertisement includes the service signature of the advertised service instance.

A *service request* R is a query for a particular service functionality. R consists of a set of *provided input parameters* $in(R)$ and a set of *required output parameters* $out(R)$.

Service discovery involves the submission of a service request R to a service directory and the retrieval of service advertisements with a service signature S that matches R. In the literature, different matching relations between R and S have been studied [11]–[13]. One particularly useful matching relation is the *plugin match*, which requires that $in(S) \subseteq in(R)$ and $out(S) \supseteq out(R)$. I.e., given the provided input parameters $in(R)$, a service instance with service signature S can be invoked, which generates all required output parameters $out(R)$.

Even if there is no single service advertisement fulfilling a given service request R, it may be still possible to *compose* multiple services in such a way that the composite service (which can be represented as a workflow) meets R. E.g., assume there are service advertisements with the following signatures S_1 and S_2: $in(S_1) = \{A, B\}$, $out(S_1) = \{C\}$, $in(S_2) = \{B, C\}$, $out(S_2) = \{D\}$ Neither S_1 nor S_2 matches the service request R, where $in(R) = \{A, B\}$ and $out(R) = \{C, D\}$. However, a service instance with signature S_1 can be invoked with the provided input parameters $\{A, B\}$, which generates the output parameter C. Now the parameters $\{A, B, C\}$ are available, enabling an invocation of a service instance with signature S_2, which generates the required output parameter D.

Algorithm 1 shows a simple service composition algorithm using forward chaining. It takes a set of service signatures Dir and a service request R as inputs and returns $success$ resp. $failure$, depending on whether R can be fulfilled. In this paper we concentrate on a decision algorithm, which however can be extended to keep track of the applied services.

The algorithm iteratively extends the set $availableInputs$. In each iteration of the loop, it selects those service signatures for which all required input parameters are available and adds the output parameters of the selected service signatures to the set $availableInputs$. The algorithm terminates if all required output parameters are available or no further service signatures can be selected. In order to avoid the repeated selection of the same service signature, the set $services$ is updated to include only those service signatures that have not been selected yet.

Algorithm 1: Simple decision algorithm based on forward-chaining to determine whether a service request R can be fulfilled by a set of service signatures Dir.

```
Compose1(Dir, R) :
    services ← Dir ;
    availableInputs ← in(R) ;
    requiredOutputs ← out(R) ;
    while requiredOutputs ⊈ availableInputs do
        applicableServices ←
            {s ∈ services | in(s) ⊆ availableInputs} ;
        if applicableServices = ∅ then
            return failure ;
        newAvailableInputs ← ⋃_{s∈applicableServices} out(s) ;
        availableInputs ←
            availableInputs ∪ newAvailableInputs ;
        services ← services \ applicableServices ;
    return success ;
```

III. Service Composition with Directory Digest

In previous work [6], we used complex directory queries in order to dynamically retrieve relevant service advertisements from a large-scale service directory during service composition. This approach caused very high workload within the service directory and consequently also slowed down the service composition algorithm because of expensive remote interactions with the service directory.

The approach presented here increases scalability by avoiding complex directory queries during composition. The service directory offers a compact digest that summarizes the service signatures of all service advertisements stored in the directory. Service composition clients download the digest, which contains sufficient information to perform service composition locally on the client side. The service composition client interacts with the service directory as follows:

1) Download digest. The client periodically downloads the most recent version of the digest. As service advertisements usually remain valid for a longer period of time, the clients do not have to reload their copy of the digest for every service request. Typical refresh rates would be once per day, once per week, etc.
2) Transform service request R. As the digest is a compressed representation of the service signatures in the directory, full parameter names are not available in the digest. Hence, the composition client sends R to the directory and receives R^T. The directory maps each

parameter of $in(R)$ (resp. $out(R)$) to the corresponding digest parameter of $in(R^T)$ (resp. $out(R^T)$). This translation is a simple mapping and can be processed in linear time with the number of parameters in R.

3) Compute composition. Using the digest and R^T, the client locally computes a service composition, without any queries to the service directory. If the composition fails, the client may update its local copy of the digest and retry (the new version of the digest may include additional service signatures of recently added service advertisements).
4) Transform composition result. If the service request has been successfully solved in the step before, the client knows the signatures of the selected services that are part of the composition workflow. It asks the directory to provide service advertisements that match the selected service signatures. The resulting directory query looks only for *exact matches*, which can be processed very efficiently by the directory. E.g., the directory may simply use the service signature to compute a hash key and look up matching service advertisements in a hash table. If the desired service signature is not found in the directory (i.e., services have been removed recently), the client has to download an up-to-date digest and re-run the composition algorithm.

The digest representation should meet the following requirements:

- Incremental update. The addition or removal of a service signature shall not require to rebuild the digest from scratch.
- Incremental addition of service parameters. The addition of a new service parameter shall not require restructuring the digest.
- Compact network transfer format. The (serialized) digest shall be as small as possible in order to reduce network bandwidth.
- Compact in-memory representation. Also the in-memory representation of the digest shall be compact in order to allow clients with limited computing resources to keep a copy of the digest in memory.
- Enabling efficient service composition. The digest representation shall enable efficient service composition algorithms.

In the following Section, we introduce a directory digest encoding that meets all these requirements.

IV. Directory Digest Representation

If we consider a service signature a combination of parameters, the directory digest can be seen as a combination set. E.g., the example service signatures S_1 and S_2 shown in Section II can be represented by the following combination set: $\{\langle A_{in}, B_{in}, C_{out}\rangle, \langle B_{in}, C_{in}, D_{out}\rangle\}$

Typically, the combination set representing the directory digest is sparse, because the number of parameters per service signature is limited. As Zero-Suppressed Reduced Ordered Binary Decision Diagrams (ZDDs) are known as a particularly efficient representation of sparse combination sets [8], we decided to encode the directory digest as a ZDD.

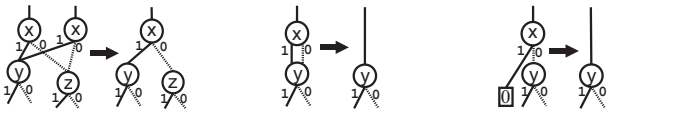

Fig. 1. Reduction rules for ROBDDs and ZDDs.

In the following, we give an overview of ZDDs and summarize the standard operations supported by ZDDs, before discussing our ZDD encoding of the directory digest in more detail.

A. ZDDs

A Binary Decision Diagram (BDD) is a directed graph representation of a boolean function. BDDs have two terminal nodes, called 0-terminal node and 1-terminal node, as well as decision nodes with the two edges called 0-edge and 1-edge. A BDD is derived by reducing a binary tree graph. In order to represent a boolean function efficiently, usually a BDD is transformed into in a Reduced Ordered BDD (ROBDD) by imposing an order on the variables in the BDD and by applying the following reduction rules [8]:

1) Share all equivalent sub-graphs in the BDD (Fig. 1 (a)).
2) Eliminate all redundant nodes whose two edges point to the same node (Fig. 1 (b)).

By mapping a combination set into the boolean space, it can be represented as the characteristic function using a ROBDD. However, ROBDDs are not well suited to represent sparse combination sets, i.e., sets where only a few objects out of many are included in each combination. The form of a ROBDD depends on the number of input variables, which has to be fixed before generating the ROBDD. Moreover, node elimination does not work well in reducing the graph in the case of sparse combinations [8].

ZDDs [8] are a special type of ROBDDs that enable an efficient encoding of sparse combination sets. For ZDDs, the following reduction rules apply:

1) Share all equivalent sub-graphs in the same manner as for conventional ROBDDs (Fig. 1 (a)).
2) Eliminate all nodes with 1-edge pointing to the 0-terminal node. Then directly connect the edge to the other subgraph (Fig. 1 (c)).

In contrast to conventional ROBDDs, ZDDs do not eliminate nodes whose two edges point to the same node. The second reduction rule for ZDDs is asymmetric for the two edges because a node remains if its 0-edge points to a terminal node. If the number and order of variables are fixed, ZDDs provide canonical forms for boolean functions.

Fig. 2 illustrates a ROBDD and a ZDD representing the combination set $\{\langle a\rangle, \langle b\rangle\}$ assuming two different 'object universes', $\{a, b, c\}$ versus $\{a, b, c, d\}$. While the form of

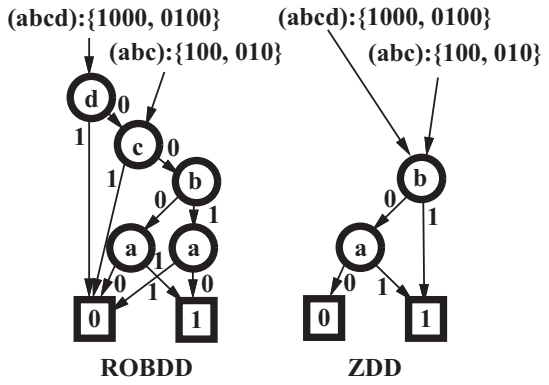

Fig. 2. ROBDD versus ZDD representation of the combination set $\{\langle a\rangle, \langle b\rangle\}$ for two different 'object universes', $\{a, b, c\}$ versus $\{a, b, c, d\}$.

the ROBDD depends on the 'object universe', the ZDD is automatically reduced into the same form for both settings, free of irrelevant variables.

Below we summarize a few ZDD operations that are used in this paper. See reference [8] for details. Z, Z_1, and Z_2 are combination sets, v is a variable representing an object in a combination, and Vs is a set of objects.

- **ZDDEmpty**() – Returns the empty set.
- **ZDDBase**() – Returns a set with a single combination that selects no object.
- **ZDDCube**(Vs) – Returns a set with a single combination that selects the objects in Vs.
- **ZDDPowerset**(Vs) – Returns the power set of the objects in Vs.
  E.g., **ZDDPowerset**($\{a, b\}$) = $\{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a, b\rangle\}$.
- **ZDDSubset0**(Z, v) – Returns the subset of Z that contains only combinations without the object v.
- **ZDDSubset1**(Z, v) – Returns the subset of Z that contains only combinations with the object v.
- **ZDDUnion**(Z_1, Z_2) – Returns $Z_1 \cup Z_2$.
- **ZDDIntsec**(Z_1, Z_2) – Returns $Z_1 \cap Z_2$.
- **ZDDDiff**(Z_1, Z_2) – Returns $Z_1 \setminus Z_2$.

B. ZDD Encoding of Directory Digest

We represent the directory digest as a single ZDD. Each service signature in the directory corresponds to a combination within the ZDD.

Assume there are n different parameters used in the set of all service signatures in the directory, which are identified by their index i ($0 \leq i < n$). We map each parameter to two ZDD variables, which are identified by integer values. The ZDD variable $2i$ corresponds to the i^{th} parameter used as input, while the variable $2i+1$ corresponds to the i^{th} parameter used as output. The function GetInputVar(p) (resp. GetOutputVar(p)) returns the ZDD variable corresponding to parameter p used as input (resp. output).

This representation is extensible, i.e., a new parameter can be included by adding two ZDD variables. Hence, our ZDD representation supports the incremental addition of service parameters (which does not change the form of the ZDD, unless a combination with one of the new variables is added to the ZDD). Incremental update of the digest is supported by the **ZDDUnion** (resp. **ZDDDiff**) operation, which allows to add (resp. to remove) a combination from the ZDD. The combination to be added or removed may be created by **ZDDCube**.

V. Service Composition Algorithms

In this Section we gradually develop an efficient decision algorithm for service composition using a directory digest represented as ZDD.

Algorithm 2 shows a straight-forward implementation of Algorithm 1 using ZDD operations. It takes a directory digest ZDD Dir, a service request R, and the set P of all parameters occurring in the digest ('parameter universe') as inputs. $selectMask$ represents the set of all possible service signatures that would be applicable given the input parameters $availableInputs$. By intersecting $selectMask$ with the set of available service signatures ($services$), we create the set of available service signatures that are actually applicable at the current state of the algorithm. The **ZDDSubset1** operation is used to determine whether an applicable service generates an output parameter which has not been available before. As we will see in the following Section, Algorithm 2 performs badly, because the operations **ZDDIntsec** and **ZDDDiff** generate new ZDDs, causing computational effort proportional to the size of the argument ZDDs (initially, one of the argument ZDDs corresponds to the whole directory digest).

Algorithm 3 is a refinement of the previous algorithm, which avoids the ZDD operations **ZDDIntsec** and **ZDDDiff**. In order to compute the set of applicable service signatures, the **ZDDSubset0** operation is used repeatedly to exclude those service signatures that require an input parameter which is not (yet) available. The termination condition is changed in such a way that the algorithm fails, if an iteration of the **while** loop has not generated any new available parameter. Algorithm 3 is just an intermediate step towards a highly efficient algorithm that minimizes the generation of ZDDs.

Algorithm 4 is our final, optimized algorithm. It relies on two new ZDD operations, which are described below. In one iteration of the **while** loop, only a single ZDD is generated.

- **ZDDMultivarSubset0**(Z, Vs) – Returns the subset of Z that contains only combinations without any object from Vs. This operation processes Z only once, generating a single ZDD.
- **ZDDExistsSubset1**(Z, v) – Returns true if $\mathbf{ZDDSubset1}(Z, v) \neq \mathbf{ZDDEmpty}()$. Does not generate any ZDD.

VI. Evaluation

In the following we evaluate our ZDD-based directory digest with regard to digest size and composition algorithm performance. For our evaluation, we created service directories with an increasing number of randomly generated, distinct service signatures (0–10^6). Each service signature S has 3 input and 3 output parameters, randomly chosen from a set of 1000

Algorithm 2: Decision algorithm to determine whether a service request R can be fulfilled by a ZDD Dir, representing a directory digest. P is the set of all parameters. Amongst others, this algorithm relies on the ZDD operations **ZDDIntsec** and **ZDDDiff**.

$ZDDCompose2(Dir, R, P)$:
  $services \leftarrow Dir$; $availableInputs \leftarrow in(R)$; $requiredOutputs \leftarrow out(R)$;
  while $requiredOutputs \not\subseteq availableInputs$ **do**
    $selectMask \leftarrow \mathbf{ZDDPowerset}\left(\bigcup_{p \in availableInputs} GetInputVar(p) \cup \bigcup_{p \in P} GetOutputVar(p)\right)$;
    $applicableServices \leftarrow \mathbf{ZDDIntsec}(services, selectMask)$;
    if $applicableServices = \mathbf{ZDDEmpty}()$ **then return** $failure$;
    foreach $p \in P, p \notin availableInputs$ **do**
      if $\mathbf{ZDDSubset1}(applicableServices, GetOutputVar(p)) \neq \mathbf{ZDDEmpty}()$ **then**
        $availableInputs \leftarrow availableInputs \cup \{p\}$;
    $services \leftarrow \mathbf{ZDDDiff}(services, applicableServices)$;
  return $success$;

Algorithm 3: Decision algorithm to determine whether a service request R can be fulfilled by a ZDD Dir, representing a directory digest. P is the set of all parameters. This algorithm uses only the ZDD operations **ZDDSubset0**, **ZDDSubset1**, and **ZDDEmpty**.

$ZDDCompose3(Dir, R, P)$:
  $services \leftarrow Dir$; $availableInputs \leftarrow in(R)$; $requiredOutputs \leftarrow out(R)$;
  while $requiredOutputs \not\subseteq availableInputs$ **do**
    $applicableServices \leftarrow services$;
    foreach $p \in P, p \notin availableInputs$ **do**
      $applicableServices \leftarrow \mathbf{ZDDSubset0}(applicableServices, GetInputVar(p))$;
    $newInputs \leftarrow \emptyset$;
    foreach $p \in P, p \notin availableInputs$ **do**
      if $\mathbf{ZDDSubset1}(applicableServices, GetOutputVar(p)) \neq \mathbf{ZDDEmpty}()$ **then**
        $newInputs \leftarrow newInputs \cup \{p\}$;
    if $newInputs = \emptyset$ **then return** $failure$;
    else $availableInputs \leftarrow availableInputs \cup newInputs$;
  return $success$;

Algorithm 4: Decision algorithm to determine whether a service request R can be fulfilled by a ZDD Dir, representing a directory digest. P is the set of all parameters. This algorithm uses the specialized ZDD operations **ZDDMultivarSubset0** and **ZDDExistsSubset1**.

$ZDDCompose4(Dir, R, P)$:
  $services \leftarrow Dir$; $availableInputs \leftarrow in(R)$; $requiredOutputs \leftarrow out(R)$;
  while $requiredOutputs \not\subseteq availableInputs$ **do**
    $missingInputs \leftarrow \{GetInputVar(p) \mid p \in P, p \notin availableInputs\}$;
    $applicableServices \leftarrow \mathbf{ZDDMultivarSubset0}(services, missingInputs)$;
    $newInputs \leftarrow \emptyset$;
    foreach $p \in P, p \notin availableInputs$ **do**
      if $\mathbf{ZDDExistsSubset1}(applicableServices, GetOutputVar(p))$ **then**
        $newInputs \leftarrow newInputs \cup \{p\}$;
    if $newInputs = \emptyset$ **then return** $failure$;
    else $availableInputs \leftarrow availableInputs \cup newInputs$;
  return $success$;

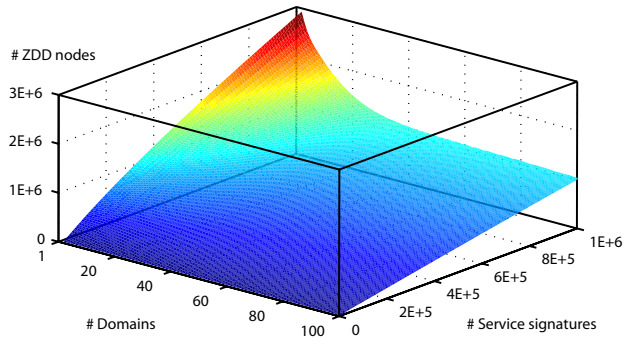

Fig. 3. Number of nodes in a ZDD representing a directory digest as a function of the number of service signatures in the digest and the number of domains.

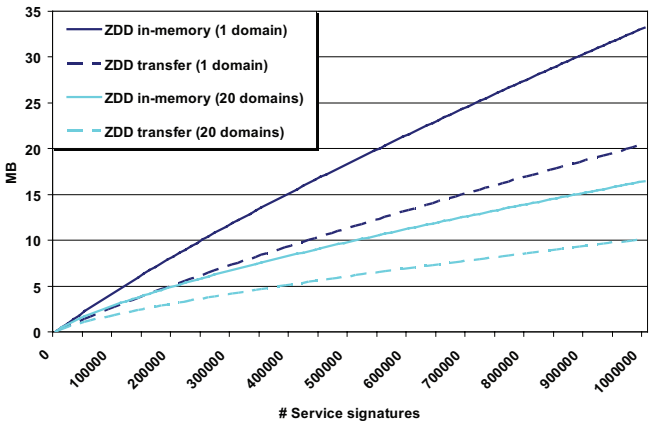

Fig. 4. Directory digest size (in megabytes) of in-memory and network transfer representations.

different parameters with the constraint $in(S) \cap out(S) = \emptyset$. In practice, services often have parameters only from a single, domain-specific ontology (e.g., travel domain, financial domain, sports domain, etc.). Hence, we partitioned the 1000 parameters into an increasing number of domains (1–100) and required the input parameters (resp. the output parameters) of each service signature to come from the same domain (the input parameters' domain can be different from the output parameters' domain).

A. Directory Digest Size

Fig. 3 shows the size of a ZDD representing a directory digest depending on the number of different service signatures and the number of domains. As ZDDs are graphs, we use the number of nodes in the graph as metric. Not surprisingly, the ZDD size increases with the number of distinct service signatures. A small number of domains results in larger ZDDs than a high number of domains. In the worst case (10^6 service signatures, 1 domain), the ZDD has about $2.9 * 10^6$ nodes. For 100 domains (each consisting of 10 parameters), 10^6 service signatures require only about 10^6 nodes. The reason for the smaller ZDD size is that the number of possible parameter combinations is reduced.

Our directory digest implementation is based on the JDD library[4], which is programmed in pure Java. Concerning the in-memory representation, JDD stores the ZDD nodes in an array of 32 bit integers. Each node uses 3 entries in the array: One entry stores a variable and the other two entries are references to other nodes. In total, a node consumes 12 bytes in memory. For the network transfer of the directory digest, the ZDD can be serialized more compactly: 11 bits are enough to store the variable[5] and 24 bits suffice to index other nodes, i.e., 59 bits per node are sufficient. Further compression of the bitstream using standard compression techniques would be possible as well. Fig. 4 illustrates the size of the in-memory and network transfer representations of our directory digest as a function of the number of service signatures, for a setting with 1 resp. 20 domains. For one million different service signatures, in the worst case the digest consumes less than 35MB in memory, resp. about 20MB on the wire.

B. Service Composition Performance

Fig. 5 compares the performance of the three service composition algorithms presented in Section V for different digest size (10^5–10^6 service signatures) and distinct number of domains (1 and 20). Because of the big differences in execution time between the different algorithms, Fig. 5 uses a logarithmic scale. Each measurement represents the total execution time for processing a set of 2000 random service requests. We chose service requests that are particularly hard to process, providing only 3 input parameters and requiring all 1000 output parameters. As we kept the set of 2000 service requests unchanged throughout all experiments, the percentage of solvable service requests is increasing with the number of service signatures in the digest. All three algorithms operate on the same digest contents and on the same set of service requests. We disabled all caches in the JDD library. As execution platform we used a machine with 4 CPUs (2 dual-core Xeon 3GHz) and 4GB RAM, running a 64-bit Windows XP installation. We employed the Sun JDK 1.5.0 with its 64-bit Hotspot Server Virtual Machine. In order to obtain reproducible measurements, we disabled background processes as much as possible. Each measurement represents the median of 15 executions on identical data.

As can be seen in Fig. 5 (a), Algorithm 2 and Algorithm 3 do not scale well with an increasing size of the directory digest. The reason is that these algorithms create many intermediary ZDDs with a size proportional to the directory digest, which also results in frequent, time-consuming garbage collection within the JDD library.

In contrast, Algorithm 4 performs very well also for a large directory digest. The high performance of Algorithm 4 is due to the reduced number of generated ZDDs. We implemented the new ZDD operations **ZDDMultivarSubset0** and

[4] `http://javaddlib.sourceforge.net/`

[5] As we have 1000 parameters in our evaluation setting, our encoding requires 2000 ZDD variables.

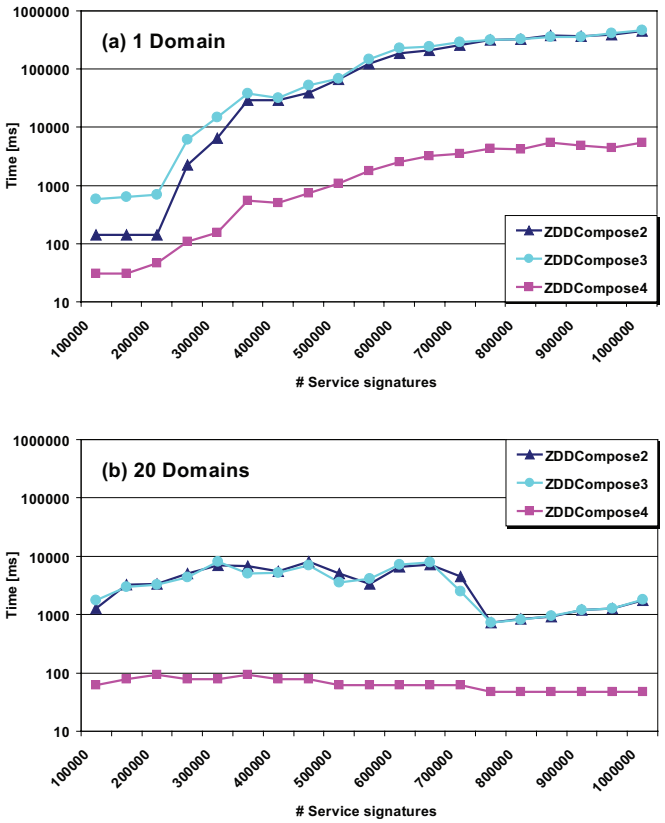

Fig. 5. Elapsed processing time [ms] for 2000 random service requests.

ZDDExistsSubset1 as an extension of the JDD library. Both operations are simple recursive methods on the ZDD graph structure, each implemented in less than 10 lines of Java code; they do not rely on the standard ZDD operations introduced in Section IV-A.

A comparison of Fig. 5 (a) and Fig. 5 (b) shows that service requests are more difficult to solve if there is only a single domain. However, our optimized composition algorithm performs very well in all settings we have measured. It is up to two orders of magnitude faster than the algorithms that use only standard ZDD operations.

VII. Conclusion

Service composition algorithms that dynamically retrieve relevant service advertisements from a large-scale service directory often issue many complex directory queries, causing high workload within the directory. As such an approach does not scale well, we introduce a compact directory digest that is downloaded by service composition clients and allows to solve the hard part of a composition problem locally without expensive remote interactions with the directory. Directory queries are only needed to translate the initial composition problem and to obtain the final result. These queries are very simple and require only a lookup in a table, significantly reducing the workload in the directory and boosting scalability.

In this paper we investigated an efficient encoding of the directory digest as a Zero-Suppressed Reduced Ordered Binary Decision Diagram (ZDD). Both the in-memory and the network transfer representations of the digest are compact. In the worst case, one million different service signatures result in a digest of less than 35MB in-memory, resp. of about 20MB on the wire.

The ZDD representation of the digest enables the development of efficient service composition algorithms. Starting with a straight-forward composition algorithm, we refined it step-by-step, in order to come up with an optimized algorithm that scales well with an increasing size of the digest. An important lesson learnt is that the direct application of standard ZDD operations may result in algorithms of inferior performance, because they may require the generation of many intermediary, possibly large ZDDs. In order to leverage the benefits of the ZDD datastructure and to obtain a highly efficient composition algorithm, we had to define our own, specialized ZDD operations.

References

[1] M. P. Papazoglou and D. Georgakopoulos, "Introduction: Service-oriented computing," *Communications of the ACM*, vol. 46, no. 10, pp. 24–28, Oct. 2003.

[2] S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi, "Dynamically composing web services from on-line sources," in *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, Edmonton, Alberta, Canada, July 2002, pp. 1–7.

[3] S. A. McIlraith and T. C. Son, "Adapting Golog for composition of semantic web services," in *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, Eds. San Francisco, CA: Morgan Kaufmann Publishers, Apr. 2002, pp. 482–496.

[4] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau, "Automating DAML-S web services composition using SHOP2," in *2nd International Semantic Web Conference (ISWC-2003)*, ser. Lecture Notes in Computer Science, vol. 2870. Sanibel Island, FL, USA: Springer, Oct. 2003, pp. 195–210.

[5] I. Constantinescu, B. Faltings, and W. Binder, "Large scale, type-compatible service composition," in *IEEE International Conference on Web Services (ICWS-2004)*, San Diego, CA, USA, July 2004, pp. 506–513.

[6] I. Constantinescu, W. Binder, and B. Faltings, "Flexible and efficient matchmaking and ranking in service directories," in *2005 IEEE International Conference on Web Services (ICWS-2005)*, Florida, USA, July 2005, pp. 5–12.

[7] W. Binder, I. Constantinescu, and B. Faltings, "Scalable automated service composition using a compact directory digest," in *17th International Conference on Database and Expert Systems Applications (DEXA-2006)*, ser. Lecture Notes in Computer Science, vol. 4080. Krakow, Poland: Springer, Sept. 2006, pp. 317–326.

[8] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," in *Proceedings of the 30th ACM/IEEE Design Automation Conference*. Dallas, TX: ACM Press, June 1993, pp. 272–277.

[9] W3C, "Web services description language (WSDL) version 1.2." [Online]. Available: http://www.w3.org/TR/wsdl12

[10] OWL-S, "DAML services." [Online]. Available: http://www.daml.org/services/owl-s/

[11] A. M. Zaremski and J. M. Wing, "Specification matching of software components," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 4, pp. 333–369, 1997.

[12] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, "Semantic matching of web services capabilities," in *Proceedings of the 1st International Semantic Web Conference*, 2002, pp. 333–347.

[13] L. Li and I. Horrocks, "A software framework for matchmaking based on semantic web technology," in *Proceedings of the 12th International Conference on the World Wide Web*, 2003, pp. 331–339.