DATESSO: Self-Adapting Service Composition with Debt-Aware Two Levels Constraint Reasoning

Satish Kumar
School of Computer Science
University of Birmingham, UK
s.kumar.8@cs.bham.ac.uk

Tao Chen
Department of Computer Science
Loughborough University, UK
t.t.chen@lboro.ac.uk

Rami Bahsoon
School of Computer Science
University of Birmingham, UK
r.bahsoon@cs.bham.ac.uk

Rajkumar Buyya
School of Computing and Information Systems
University of Melbourne, Australia
r.buyya@unimelb.edu.au

ABSTRACT

The rapidly changing workload of service-based systems can easily cause under-/over-utilization on the component services, which can consequently affect the overall Quality of Service (QoS) such as latency. Self-adaptive services composition rectifies this problem, but poses several challenges: (i) the effectiveness of adaptation can deteriorate due to over-optimistic assumptions on the latency and utilization constraints, at both local and global levels; and (ii) the benefits brought by each composition plan is short-term and not often designed for long-term benefits—a natural prerequisite for sustaining the system. To tackle these issues, we propose a two levels constraint reasoning framework for sustainable self-adaptive services composition, called DATESSO. In particular, DATESSO consists of a refined formulation that differentiates the ‘strictness’ for latency/utilization constraints in two levels. To strive for long-term benefits, DATESSO leverages the concept of technical debt and time-series prediction to model the utility contribution of the constituents component services in the composition. The approach embeds a debt-aware two level constraint reasoning algorithm in DATESSO to improve the efficiency, effectiveness and sustainability of self-adaptive service composition. We evaluate DATESSO on a service-based system with real-world WS-DREAM dataset and comparing it with other state-of-the-art approaches. The results demonstrate the superiority of DATESSO over the others on the utilization, latency and running time whilst being more sustainable.

CCS CONCEPTS

- Model-driven software engineering.

KEYWORDS

Self-adaptive systems, service composition, technical debt, constraint reasoning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SEAMS 2020, 25 - 26 May, 2020, Seoul, South Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/Y/M. . . $15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:


1 INTRODUCTION

Service composition allows a software being built by seamlessly composing readily available component services, each of which offers different guarantee on Quality-of-Service (QoS) on latency [35]. This is an important property given the increasing popularity of Cloud Computing, Edge Computing and Internet-of-Things. However, a known difficulty in service-based systems is the presence of rapidly changing workload, leading to under-/over-utilization on the component services [19]. On one hand, increasing workload can rise over-utilization for the component services within a composite service, which in turns, would negatively affect the latency and violate the Service Level Agreement (SLA) [25] [19]. On the other hand, decreasing workload may lead to under-utilization of the capacity of component services, reducing the revenue that should have been achieved as the infrastructural resources also impose monetary cost. To address those issues, self-adaptation on service composition is promising, but the adaptation needs to be effective while being efficient and sustainable.

When reasoning about self-adaptation for service composition, there are often two levels of latency/utilization constraints: the local constraint at each of the individual service and the global one for the entire service composition. Both of them are critical, as they can affect what alternative composition plans to be searched during the adaptation [26]. However, existing work on self-adapting service composition relies on over-optimistic assumption, such that both local and global constraints are hard and can always be satisfied [31][6][21][17][26]. This can negatively influence the adaptation quality and efficiency, rendering length reasoning process, especially when the given constraints are unrealistic/inappropriate. Further, under such a strong assumption, it would completely ignore the fact that certain composition plan may temporally violate the constraint, but likely to create much larger benefit after a certain period of time.

Given the rapidly changing workload, it is important to ensure that each adaptation can be effective over a period of time, which...
also mitigate unnecessarily frequent adaptations. However, current
work informs adaptation relies mainly on a short-term prospective,
i.e., the immediate improvement of a composition plan, according
the detected (or predicted) latency constraint violation/undesired
utilization [16][33][8]. This is harmful in terms of sustainability, as
immediately low utilization/high latency in a short-term manner
may not necessarily mean an undesired composition plan; in fact,
it can be the source that stimulates largely increased benefit in
the long term. For example, under-utilization could be desirable
temporarily in order to prepared for a largely increased workload
in the long-term prospective. Similarly, over-utilization may be
acceptable in short time, as long as the workload is only a ‘spike’
and the loss can be paid off by long term benefits. As a result, despite
that adapting with composition plan that has the best immediate
improvement may lead to short-term advantages, it can easily create
instability and hinder the possibility of achieving higher benefits
in the long-term prospective.

To address the above challenges, in this paper, we propose a
framework that leverages debt-aware two levels constraint reason-
ing for self-adapting service composition (hence called DATESSO).
We show that DATESSO can achieve better utilization/latency in
the long-term prospective while being faster than state-of-the-art
approaches, providing better sustainability for self-adaptive service-
based systems. In a nutshell, the major contributions of this paper
are summarized as follows:

• Instead of formalizing the constraints at both local and global
  levels as hard ones, we refine the global constraints as the
  soft ones. This has enabled us to tailor the reasoning process
  in self-adaptation and mitigate over-optimism.

• We propose temporal debt-aware utility, a new concept that
  extends from the technical debt metaphor, to model the long-
  term benefit contribution of possible component services
  that constitute to a composition plan.

• Drawing on the above, we design an efficient two level con-
  straint reasoning algorithm in DATESSO that is debt-aware,
  and utilizes the different strictness of the two level con-
  straints to reduce the search space.

• We evaluate DATESSO on a commonly used service-based
  system [12][11][20] whose component services are derived
  from the real-world WS-DREAM dataset [36] and under the
  FIFA98 workload trace [7]. The results show that, in contrast
to state-of-the-art approaches [6][16][23], DATESSO achieves
  better utilization and latency while having smaller overhead,
  leading to more sustainable self-adaptation for service com-
  position.

The remaining of the paper is organized as follows: Section 2
presents the background information of service composition, the
constraints, technical debt and a running example of the issues.
Section 3 shows an overview of DATESSO. Section 4 discusses our
formalization of the two level constraints with different strictness.
The temporal debt-aware utility model and the debt-aware two level
reasoning algorithm are specified in Section 5 and 6, respectively.
Then, we presents the experiment results in Section 7, following
by discussion of threats to validity in Section 8. Section 9 compares
DATESSO with existing work and Section 10 concludes the paper.

2 PRELIMINARIES

2.1 Self-Adaptation in Service Composition
A service composition is a special software form that consists of a
particular workflow of connected abstract services, denoted as
\( \{a_1, a_2, \ldots, a_n\} \). Each of these abstract services can be realized by
using a readily available component service selected from the In-
ternet. Typically, there could be multiple component services to be
selected, and the xth component service for the xth abstract service
is denoted as \( c_x \). Therefore the possible component services for
the xth abstract service form a set, denoted as \( \{c_{x1}, c_{x2}, \ldots\} \), each
of which has different generic latency guarantee on its capacity. For
example, \( c_{xy} \) has a capacity to process 50 requests in 0.5 seconds.

In such context, a SLA may be negotiated for providing legal
document to ensure the performance of a service composition.
The most notable elements in the SLA are the constraints on the
utilization of service capacity and the achieved latency level per
request, which we will elaborate in the next section.

As the workload changes, at runtime, the goal of self-adaptation
for service composition is to find the composition plan, \( \{c_{x1}, c_{x2}, \ldots, c_{xy}\} \)
that improves the utilization and latency so that they satisfy all the
constraints specified, for as long time as possible.

2.2 Constraints in Service Composition
In service composition, constraints denote the stakeholders’ ex-
pectation of the latency guarantee. Most commonly, a SLA can
define these constraints by specifying the bound of the latency and
utilization [17]. For example, a service’s latency should not exceed
10s or the utilization is at lest 0.7. Typically, there are two levels of
constraints:

• **Global constraint:** The global constraint specifies the mini-
  mum expectation of latency/utilization for the entire service
  composition. It is often the most common requirement in a
  service-based systems [23][6].

• **Local constraint:** The local constraints are specified for
  the latency/utilization on each abstract service\(^1\). This is im-
  portant, as each abstract services can be realized by the
  component service from different parties; any violation of
  the local constraint would in fact cause severe failure in the
  composition, leading to an outage [17][6].

It is worth noting that, satisfying all local constraints does not
necessarily mean that the global constraint can be satisfied, since
each of the constraints is documented separately [31]

2.3 Technical Debt
Technical debt is a widely recognized metaphor in software develop-
ment [29][9][3]. Its core idea is to describe the extra cost incurred by
actions that compromises long-term benefits of the developed soft-
ware, e.g., maintainability, in order to gain short-term advantages
(e.g. timely software release).

The technical debt metaphor was initially introduced by Cun-
ningham [15] in the context of agile software development, where
the definition is described as:

\(^1\)For latency, this constraint would applied for each request.
Figure 1: A running example of issues in service composition ($L$ and $T$ mean that the selected component service of an abstract service can process all $T$ requests in $L$ seconds)

“Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. The danger occurs when the debt is not repaid. Every minute spent on not quite right code counts as interest on that debt.”

In this regards, technical debt is often used in an economic-driven decision approach for communicating technical trade-off between short-term advantages and long-term benefits in software projects [29]. In the context of service composition, the notion of technical debt can be perfectly aligned with the requirement of long-term benefit: each possible component service may associate with a debt with respect to constraint violation. Such a debt, once selected, may or may not be repaid over a period of time, depending on the actual workload.

2.4 Running Example

In this section, we present a simple example of service composition to explain the problems. As shown in Figure 2, there is a service composition in the form of sequentially connected abstract service, each of which has select a particular component service for realization. In this case, each selected component service has its own overall capacity, e.g., the selected component service for SEARCH ITEM abstract service can process all 50 requests in 0.19 seconds.

As mentioned, each abstract service, along with the entire service composition, are legally documented with separated constraints on the utilization and latency per request, as specified in the SLA. Suppose that in this scenario, the local constraint of utilization and latency of each request for the abstract service PAYMENT BY CREDIT CARD could be 0.8 and 0.18 seconds, respectively. Meanwhile, the global constraint of utilization and latency of each request for the service composition is 0.85 and 0.75 seconds, respectively. Given the changing workload, it is likely that either (or both) levels of constraint may be violated, which requires self-adaptation to replace the component services. However, there are two issues with this:

(1) In this context, the different constraints are negotiated independent to each others. While it is relatively easy to find the alternative component service that satisfy the local constraints, searching for the composition plan that satisfies the global constraints is difficult, or we may not know whether there is one exist. As a result, existing approaches that treats both levels of constraints as hard constraints suffers the issue of being over-optimistic: they may struggle to find a satisfactory composition plan, especially under a scenario where such plan barely exists. Further, this would completely eliminate the composition plan that may cause temporary violation of global constraints, but can create much larger long-term benefits.

(2) When self-adaptation is required, a possible component service and the entire composition plan may provide short-term, immediate benefit in relieving constraint violation, but it is difficult to know that whether such benefit can be sustainable. In contrast, it is possible to temporarily accept a composition plan that may still violate the global constraint, but will generate larger benefit in a long-term perspective. Therefore, self-adapting service composition without having any guarantee on the long-term manner can lead to frequent adaptations with merely short-term benefits, which generate unnecessary overhead.

The DATESSO proposed in this work was designed to explicitly address these two issues in self-adapting service composition.

3 DATESSO OVERVIEW

Figure 2 illustrates the overview of DATESSO. As can bee seen, there are three key stages, namely Formalization, Modeling and Reasoning, each of which is specified as follows:

(1) **Formalization**: This is the very first stage in DATESSO and it relies on the Two Levels Formalizer component. Generally, it has two tasks: (i) formulating and recording the global/local level constraints as documented in the SLA; (ii) monitoring the service composition and informing the Modeling stage, along with any information of the constraints, when any violations are detected. More details are discussed in Section 4. Note that here, we trigger adaptation only based on local constraint violations, as we formalize the global ones as soft constraints. However, the global constraint is implicitly considered in the Reasoning stage.

(2) **Modeling**: Once the local constraint violation has been detected, the Workload Predictor keeps track of the historical workload on each abstract service, and provides a time-series model to be embedded with constraint information, which together form the temporal debt-aware utility model. A detailed discussion is presented in Section 5.

(3) **Reasoning**: At the final stage, the utility model that is debt-aware, the two level constraints and the Service Repository
bring the individual service down, which cause outage of the entire service composition. Locally, for each component service $c_{xy}$ that has a capacity to process $T_{c_{xy}}$ requests in $L_{c_{xy}}$ seconds, we model normalized the constraint ($CL_{c_{xy}}$) on the normalized actual latency of each request ($L_{c_{xy}}$) to be satisfied as below, both of which are within $[0, 1]^2$:

$$L_{c_{xy}} = \frac{L_{c_{xy}} \times W_{t,c_{xy}}}{T_{c_{xy}}} \leq CL_{c_{xy}}$$  \hspace{1cm} (1)$$

where $W_{t,c_{xy}}$ is the workload for the corresponding abstract service (hence for $c_{xy}$ too) at timestep $t$. Likewise, the local constraint ($CU_{c_{xy}}$) on utilization ($U_{c_{xy}}$) to be satisfied can be formulated as:

$$U_{c_{xy}} = \frac{L_{c_{xy}} \times W_{t,c_{xy}}}{CL_{c_{xy}} \times T_{c_{xy}}} \geq CU_{c_{xy}}$$  \hspace{1cm} (2)$$

Since the local constraints are hard, we say a component service as feasible if, and only if, both utilization and latency constraints are satisfied. Otherwise it is termed infeasible.

### 4.2 Soft Global Constraints

Unlike existing work that model global constraint as hard threshold, we model its soft version that can tolerate certain violation, with an aim to mitigate the issue of over-optimism. Indeed, the way of aggregating the local latency toward the global value for the entire service composition depends on the connectors, which may be sequential, parallel or recursive etc. However, as shown in [34][2], sequential connector is the most fundamental type and all other connectors can be converted into sequential ones. Therefore in this work, we focus on sequential connector in our models.

Similar to its local counterpart, for all selected component services in the entire service composition, the satisfaction on normalized global constraint ($CL_{global} \in [0, 1]$) and the normalized actual latency of each request ($L_{global} \in [0, 1]$) can be calculated by aggregating the locally achieved latency. Specifically, when all the connectors are sequential or they have been converted into sequential ones, the satisfaction of global latency can be formulated as:

$$L_{global} = \sum_{x} \sum_{y} L_{c_{xy}} \leq CL_{global}$$ \hspace{1cm} (3)$$

The global aggregation for other types of connectors can be obtain easily, as suggested in [12][11]. Likewise, the global constraint ($CU_{global}$) on utilization ($U_{global}$) to be satisfied can be formulated as:

$$U_{global} = \frac{1}{N} \times \sum_{x} \sum_{y} U_{c_{xy}} \geq CU_{global}$$ \hspace{1cm} (4)$$

whereby $N$ denotes the total number of abstract services. As mentioned, there is no guarantee that satisfying the local parts at component level can lead to global satisfaction. However, it is easy to see that a violation of the global constraint is contributed by some

---

Footnotes:

1 Normalization can be achieved by using the lower and upper bounds of possible latency values.

2 Utilization naturally sits within $[0, 1]$, as any requests go beyond the capacity would be discarded.
(or all) of the component services selected, even though their local constraints may have been satisfied.

5 TEMPORAL DEBT-AWARE UTILITY MODEL

In DATESSO, we propose temporal debt-aware utility model, a notion derived from technical debt metaphor [15], that quantifies the long-term benefit of each component service that constitute to a composition plan. To this end, we adopt the notion of principal and interest [29][5][3] to analyze the debt values related to a single component service, which is feasible. Built on the concept of two level constraints and their different strictness, a debt can quantify each feasible component service’s local contribution to the overall debt at the global level over a period of time.

5.1 Modeling Temporal Debt Value

5.1.1 Principal. The principal, denoted as \( P_{cxy} \), is the one-off cost of the processes on adapting a component services \( c_{xy} \). It can be calculated as:

\[
P_{cxy} = O_{cxy} \times C_{com}
\]

Suppose that the composition process for adding a component service requires a overhead of 5 seconds (denoted as \( O_{cxy} \)) and the execution cost of computing resource is $0.005 per second (denoted as \( C_{com} \)), then it takes a principal as \( 5 \times 0.005 = 0.025 \). Note that \( P_{cxy} \) here is a normalized value in the range of \([0, 1]\) based on the lower/upper bounds of the possible execution cost and composition time. The \( O_{cxy} \) can be easily known by analyzing the time for previous rounds of composition. Alternatively, as what we have done in this work, it can be obtained via profiling the service broker.

5.1.2 Accumulated interest. Over time, interests can be accumulated due to continuous constraint violations. Since the local constraints are hard, there will be no interest incurred directly at this level. However, because we model the global constraints as the soft ones, any violation of the global constraint are contributed by the component services at the local level, even if the local constraint has been satisfied. In particular, according to Equation 3 and 4, over a period of time, any possible violation of global constraint would be contributed by all component services that have local utilization/latency worse than the global constraint, which causes potential interest. With this in mind, the accumulated interests of a component service \( c_{xy} \) between timestep \( n \) and \( m \) can be modeled as:

\[
I_{n,m,cxy} = \alpha_{n,m,cxy} + \beta_{n,m,cxy}
\]

subject to

\[
\alpha_{n,m,cxy} = \sum_{t=n}^{m} (C_{U_{global}} - U_{cxy}), \forall t \iff C_{U_{global}} \geq U_{cxy}
\]

\[
\beta_{n,m,cxy} = \sum_{t=n}^{m} (C_{L_{global}} - L_{cxy}), \forall t \iff L_{cxy} \geq L_{global}
\]

whereby \( \iff \) represents ‘such that’. Hence, \( \alpha_{n,m,cxy} \) and \( \beta_{n,m,cxy} \) consider only those timesteps between \( n \) and \( m \), at which contribution to the possible violation of global constraint exist. In particular, these equations guarantee that \( \alpha_{n,m,cxy} \geq 0 \) and \( \beta_{n,m,cxy} \geq 0 \).

It is easy to know that in general, if \( \alpha_{n,m,cxy} = 0 \) and \( \beta_{n,m,cxy} = 0 \), which means \( c_{xy} \) does not contribute to any possible violation at all, then the overall accumulated interest for \( c_{xy} \) over a period of time is 0. Otherwise, the interest, incurred by the contribution to the possible violation of either global utilization or latency constraint (or both), would be part of the debt. For example, when \( C_{U_{global}} = 0.9 \) and \( C_{L_{global}} = 0.7 \), at a particular timestep \( t \), a feasible component service has utilization and latency of \( U_{cxy} = 0.7 \) and \( L_{cxy} = 0.85 \), respectively. In this case, for any possible violation of the global utilization and latency constraint at this timestep, \( c_{xy} \) would contribute a total of \( I_{t,cxy} = 0.9 - 0.7 + 0.85 - 0.7 = 0.35 \) interest (and thus part of the debt) to cause the violations. The overall interest over a range of timesteps would be the sum of the interest incurred by the above case under each timestep.

5.1.3 Connecting debt and utility. Finally, we calculate the debt for a feasible component service between timestep \( n \) and \( m \) as:

\[
D_{n,m,cxy} = P_{cxy} + I_{n,m,cxy}
\]

Since both \( P_{cxy} \) and \( I_{n,m,cxy} \) are normalized or naturally sit between \([0, 1]\), the numeric stability can be improved. Drawing on the above, we then be able to obtain a debt-aware utility score \((S_{n,m,cxy})\) for \( c_{xy} \) between \( n \) and \( m \), defined as:

\[
S_{n,m,cxy} = \sum_{t=n}^{m} U_{cxy} - \sum_{t=n}^{m} L_{cxy} - D_{n,m,cxy}
\]

A larger \( S_{n,m,cxy} \) implies that the component service \( c_{xy} \) is more likely to contribute to the satisfaction of global constraints in a long-term perspective. Here, it is clear that we will accept certain debt, as long as it can be paid back by achieving better overall utility across the timesteps considered. In this way, during the reasoning process, DATESSO is able to quantify the long-term benefit of each feasible component service over a range of timesteps, based on which enabling better informed reasoning.

5.2 Time-Series Workload Prediction

Predicatively analyzing debt is not uncommon, especially in the software development context where the concept of technical debt was originally created [15]. Often, the fact of whether a debt can be paid off depends on the present and future values of the debt [10][28]. This is also an equivalent and important concept in our research, and therefore we seek to predict the future workload of the component services, which in turn, enabling informed reasoning of long-term benefit during self-adaptation.

In DATESSO, we use Autoregressive Fractionally Integrated Moving Average model (ARFIMA) [32], a widely used time-series model that guarantee the prediction accuracy when a time-series contains long memory pattern, to predict the workload of each abstract service.

Accordingly, for each abstract service that is realized by a component service, we prepared the data at each time point to contain...
a number of observed requests at each timestep and feed this time-series data as an input to the ARFIMA for predicting the future requests workload for a future timestep. The general expression of ARFIMA \((p, d, q)\) for the process \(X_t\) is written as:

\[
\Phi(B)(1 - B)^d X_t = \Theta(B)\epsilon_t
\]  
(11)

where \((1 - B)^d\) is the fractional differencing operator and the fractional number \(d\) is the memory parameter, such that \(d \in (-0.5, 0.5)\).

\(\Phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p\) is the autoregressive polynomial of order \(p\) and \(\Phi(B) = 1 + \theta_1 B + \theta_2 B^2 + \ldots + \theta_q B^q\) is the moving average polynomial of order \(q\) in the lag operator \(B\). The operator \(B\) is the backward shift operator; \(B X_t = X_{t-1}\) and \(\epsilon_t\) represent the white noise process.

The prediction process in DATESSO is written as ARFIMA \((p, d, q)\) process, in which we estimate the value of memory parameter \(d\) using \(fdGPH\) function from the R forecast package proposed by Geweke and Porter-Hudak [24]. Specifically, the memory parameter \(d\) must be between \(-0.5\) and \(0.5\) that confirms long memory patterns in time-series. \(p\) is the autoregressive order that indicates the number of differenced lags appearing in the prediction equation; and \(q\) is the moving average order that shows the number of lagged forecast error in the prediction equation. The actual values of \(p\) and \(q\) are identified based on the autocorrelation function and partial autocorrelation function, respectively, as supported in the \(fdGPH\) function.

6 DEBT-AWARE TWO LEVELS CONSTRAINT REASONING

Drawing on our formalization of soft/hard constraints at two levels, along with the proposed temporal debt-aware utility model, we design a simple yet efficient reasoning algorithm for self-adapting service composition. In a nutshell, once violation on local constraints is detected, the algorithm has two main functions that are run in order:

(1) IDENTIFICATION: In this function, we firstly identify which are the component services that violate the local constraints. Then, those infeasible component services are what need to be replaced, as they also contributing the likely violation of global constraint. It is possible that all component services need to be replaced.

(2) SEARCH: Once we identify the set of abstract services whose component service needs a replacement, this function works on each individual abstract service. The aim is to search for the best feasible component service for each identified abstract service, such that it satisfies the local constraint\(^4\) while having the best long-term debt-aware utility, over all timesteps up to the future timestep \(m\) (Equation 10). As a result, the newly selected component services would less likely to cause local/global constraint violation in the future.

Each of the key steps are discussed in details as follows.

\[^4\]Given that the local constraint is specified at the local level, there will be at least one readily available component service to satisfy such constraint at a particular timestep, or otherwise the constraint may be too strong and needs to be relaxed.

\[^5\]Y may differ for different abstract services, but in this example we assume that same as our aim is merely to intuitively illustrate the reduction of complexity.

Algorithm 1: IDENTIFICATION

1. Input: \(S\): Set of selected component services and their abstract services at current timestep \(n\)
2. Output: \(S_{inf}\) \(\rightarrow \phi\): Set of abstract services whose component service needs a replacement
3. for \(\forall e_{xy} \in S\) do
   4. if \((L_{xy} > CL_{xy} \text{ or } U_{xy} < CU_{xy})\) then
      5. \(S_{inf} \leftarrow a_x\)
   6. end
7. end
8. return \(S_{inf}\)

6.1 Identifying Infeasible Component Services

As mentioned, since the constraint at local level is hard, the IDENTIFICATION function is designed to filter all the component services that are ‘working fine’. In fact, this steps is an effective way to reduce the search space, as only the problematic component services that violates the hard constraints are considered. These infeasible component service can actually contribute to the global constraint violation, if any.

The corresponding algorithmic procedure has been illustrated in Algorithm 1. As can be seen, the returned result is a set, denoted as \(S_{inf}\), that contains every abstract service (i.e., \(a_x\)) whose component service becomes infeasible at the current timestep \(n\).

6.2 Searching for the Best Long-term Debt-Aware Utility

The special design in the SEARCH function is that, instead of having to examine every combination of the service composition globally, we only search for the component service with the highest long-term debt-aware utility for each identified abstract service independently.

This is because, according to Equation 10, the problem of searching the best long-term debt-aware utility (between timestep \(n\) and \(m\)) for the entire service composition can be defined as follow:

\[
\arg\max Z \sum_{x=1}^{Z} S_{n,m,c_{xy}}
\]  
(12)

whereby \(Z\) is the total number of abstract services whose component service need a replacement. Clearly, this is a typical linear programming problem, in which achieving the best utility of the service composition is equals to find the optimal value of each \(S_{n,m,c_{xy}}\). From Equation 10, we know that the best \(S_{n,m,c_{xy}}\) is solely equivalent to the highest debt-aware utility from all the feasible component services of the \(x\) abstract service. In other words, the highest \(S_{n,m,c_{xy}}\) can be searched on each abstract service locally, in order to have the highest utility for the service composition globally. With this consideration, our reasoning algorithm decomposes the problem and reduces the search complexity from \(Y^N\) (when all combinations need to be searched at the global level) down to \(Y \times X\), where \(X\) is the number of problematic abstract service, each with \(Y\) feasible component services\(^5\).

Satish Kumar, Tao Chen, Rami Bahsoon, and Rajkumar Buyya
The corresponding algorithmic procedure has been illustrated in Algorithm 2. Specifically, suppose that the $S_{\text{inf}}$ has been found by Algorithm 1, and that the current timestep is $n$ and we are interested in the debt up to a given timestep $m$ in the future, there are three important steps:

(1) From line 4 to 14, for each problematic abstract service $a_x$, we firstly construct an ordered list of vectors denoted as $M_x$. Each vector in $M_x$ has a size of $m - n$ and it contains all the feasible component service for $a_x$ under every particular timestep between $n$ and $m$.

(2) From line 15 to 20, for each $M_x$, we find the largest timestep $m_x$ since $n$ such that there is at least one feasible component service that satisfies the local constraint on every timestep between $n$ and $m_x$. Next, we use the smallest $m_x$ across all $M_x$ to serve as the new $m$. This process ensures that all problemmatic abstract services would have at least one component service which can be treated as feasible on all timesteps considered. Here, since there is at least one feasible component service for a particular timestep, the worst case would be $m = n + 1$.

(3) From line 21 to 24, for each $a_x$, we find the set of feasible component services ($S_x$) that satisfy the local constraints on every timestep between $n$ and $m$. The SearchUtility function searches locally on the set $S_x$, and returns the one with the highest $S_x,n,c_{xy}$ as part of the composition plan. Note that, SearchUtility can be realized by any search algorithm, e.g., exhaustive search or stochastic search like Genetic Algorithm. Since in this work the $S_x$ has been reduced to a computationally tractable size, we simply apply an exhaustive search.

As the global constraints are soft, the reasoning algorithm has never explicitly used them to act as caps or thresholds for the search (like what we did for the hard local constraints), but the global constraints, along with their potential violations contributed by the component services, are implicitly embedded in the debt-aware utility model. In this way, we aim to mitigate the problem of being over-optimism on the global constraint, while at the same time, promoting larger chance to satisfy the global constraint in a long-term manner.

7 EVALUATION

To evaluate DATESSO, we design experiments to assess the performance on self-adapting service composition by means of comparing it with the state-of-the-art approaches. In particular, we aim to answer the following research questions (RQs):

- **RQ1**: Can DATESSO achieve better global utilization and latency than the state-of-the-art approaches? If so, which parts contribute to the improvement?
- **RQ2**: Is DATESSO more sustainable than the state-of-the-art approaches?
- **RQ3**: What is the running overhead of the reasoning process in DATESSO comparing with the others?

7.1 Experimental Setup

Our experiments have used a commonly applied service-based system [12],[11],[20] with 10 abstract services, each of which has 10 possible component services to be selected. Without considering reduction, the system would have a search space of $10^{10}$ possible composition plans for self-adaptation. All the values of latency and throughput capacity for the component services are randomly chosen from the WS-DREAM dataset [36].

To emulate realistic workload for each abstract service that is realized by a component service, we extracted the FIFA98 trace [7] for the length of 6 hours with 7200 timesteps, which forms the

### Algorithm 2: Search

1. **Input**: $R_x$: The set of possible component services for the $x$th abstract service
2. $S_{\text{inf}}$: Set of abstract services whose component service needs a replacement
3. **Output**: $S_{\text{Optimal}}$: Service composition plan with the optimal long-term debt-aware utility between current timestep $n$ and the future timestep $m$
4. for $\forall a_x \in S_{\text{inf}}$
   1. $M_x$ = the ordered list of vectors of the feasible component services for the $x$th abstract service at every timestep from $n$ to a future timestep $m$ */
   2. $S_x,t$ denotes the vector of the feasible component services for the $x$th abstract service at timestep $t$ */
   3. $M_x = \{S_{x,n}, S_{x,n+1}, ..., S_{x,m}\}$
   4. for $\forall c_{xy} \in R_x$ do
      1. for $t \leftarrow n + 1$ to $m$ do
         1. if $L_{c_{xy}} \leq U_{c_{xy}}$ and $U_{c_{xy}} \geq C_{U_{cxy}}$ then
            1. $S_x,t \leftarrow c_{xy}$
         2. end
      2. end
   5. $M_x \leftarrow M_x$
   6. end
   7. for $\forall M_x \in M$ do
      1. if $m_x < m$ then
         1. $m_x = \text{getLargestFeasibleStep}(M_x)$
      2. end
   8. end
   9. for $\forall M_x \in M$ do
      1. $S_x = \text{getFeasibleServices}(M_x,m)$ */
      2. $S_{\text{Optimal}} \leftarrow \text{searchUtility}(S_x,n,m)$ */
   10. end
11. return $S_{\text{Optimal}}$
workload dataset. Such a workload trace is used on all the different workflows of service composition. We pre-processed the first 4 hours workload trace as the samples for training the time-series prediction model, while the remaining 2 hours workload data, which equals to 7200 seconds, is used for testing the accuracy. In DATESSO, we feed the training data into the ARFIMA, which is implemented using the arfinla package [30] and the fGPH function in R [24].

Table 1 shows the parameter settings of the SLA used in the experiments, including the executing resource of selecting a component service ($C_{com}$), the local and global constraint for latency ($C_{l_{xy}}$ and $C_{l_{global}}$) and utilization ($C_{u_{xy}}$ and $C_{u_{global}}$). For simplicity of exposition, we have set the same local constraint for all abstract services. All the settings above have been tailored to be reasonable throughout the experiments.

All experiments were carried out on a machine with Intel Core i7 2.60 GHz. CPU, 8GB RAM and Windows 10.

### 7.2 Comparative Approaches

To answer all the RQs, we examine the performance of DATESSO against the following approaches:

- **Two Level Hard Constraints Approach (TLHCA):** This is similar to DATESSO, which differs only on the way about how the strictness of the two levels constraints is formulated. TLHCA assumes that both local and global constraints are hard, and thereby in the reasoning algorithm (Algorithm 2), when the final composition plan violates the global constraint (for every timestep between $n$ and the newly defined $m$) then we examine whether all abstract services have been considered in this run. If not, we then rerun the algorithm with consideration that all the abstract services are subject to replacement; if all abstract services has been considered but the global constraint(s) is still violated, we would have no choice but to trigger the adaptation. Here, the adaptation is triggered based on both local and global constraint violations. This approach follows the existing work [6] that makes the same formulation, and by this mean, we aim to examine the usefulness of formulating the global constraints as the soft ones.

- **Debt-Oblivious Approach (DOA):** This is a similar copy of DATESSO but without the temporal debt-aware utility model. Instead, DOA assumes the predicted utility of the aggregated latency and utilization, i.e., Equation 10 without the debt, which is then used in the reasoning algorithm to find the composition plan for self-adaptation. Such a predicted approach has been used in existing work [16], and DOA helps us to examine the effectiveness of incorporating debt information for achieving long-term benefit in self-adaptation.

- **Region-Based Composition (RBC):** This is an implementation of a state-of-the-art approach, proposed by Lin et al. [23], that relies on regions, where for each abstract service, the component service is selected according to its region. Each of these regions are clustered based on the historical utilization and latency of the component services. Here, the adaptation is triggered based on global constraint violations only.

### 7.3 Metrics

We leverage the following metrics to assess the results:

- **Global utilization:** This is the value calculated by Equation 4 for each timestep.
- **Global latency:** This is the value calculated by Equation 3 for each timestep.
- **Accumulated debt:** Since the interests are accumulated, so does the debt. A lower debt means that component services, which are less likely to contribute to global constraint violation in the long-term manner, are preferred. Therefore, we measure the accumulated debt of the service composition from the beginning to the timestep $t$ using:

$$D_{l,t} = \sum_{x} \sum_{y} D_{l,t,c_{xy}}$$

- **Sustainability score:** We measure sustainability as follows:

$$Score_{n,m} = \frac{1}{V} \times \left( \frac{S_{n,m} - S_{min,n,m}}{S_{max,n,m} - S_{min,n,m}} + 1 \right)$$

whereby $S_{n,m} = \sum_{k=1}^{Z} S_{n,m,c_{xy}}$; $n = 1$ and $m = 7200$; $Z$ is the total number of abstract services; $V$ is the total number of local and global constraint violations. $S_{min,n,m}$ and $S_{max,n,m}$ are the lower and upper value among all approaches. $Score_{n,m} \in [1, 2]$ and a higher value means that the adaptations would generate more benefits in general when mitigating each constraint violation.

- **Running time:** This is the required running time for the reasoning process to produce a composition plan.

Whenever overall results are reported, we use the pairwise version of the Kruskal Wallis test ($p < .05$) [18] and $\eta^2$ value [13] to measure statistical significance and effect size, respectively.

### 7.4 RQ1: Performance of DATESSO

Figure 3 and 4 respectively illustrate the global utilization and latency for all approaches and timesteps. As can be seen, the comparison between DATESSO and any other three are statistically significant with large effect size. In particular, when comparing with ROC, DATESSO achieves much better utilization and latency overall, while at the same time, it has smaller bias than ROC.

To better understand which of our contributions in DATESSO enables such improvement, we firstly compare it with TLHCA and DOA. As shown in the boxplots, we see that DATESSO achieves much better utilization and smaller deviation. For latency, DATESSO is slightly more deviated, but provides overall better result. This has proved that, in general, the formalization of two levels constraints

**Table 1: Parameter settings**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{l_{xy}}$: local latency constraint per request</td>
<td>0.09s</td>
</tr>
<tr>
<td>$C_{l_{global}}$: global latency constraint per request</td>
<td>1s</td>
</tr>
<tr>
<td>$C_{u_{xy}}$: local utilization constraint</td>
<td>0.8</td>
</tr>
<tr>
<td>$C_{u_{global}}$: global utilization constraint</td>
<td>0.9</td>
</tr>
<tr>
<td>$C_{com}$: cost of computing resource</td>
<td>$$0.0025$</td>
</tr>
<tr>
<td>$m$: future timestep $m$ from current timestep $n$</td>
<td>$n + 5$</td>
</tr>
</tbody>
</table>
with different strictness can help to improve self-adaptation performance. Next, we compare DATESSO with DOA, for which we see that again, DATESSO achieves generally better and more stable results on utilization and latency. This evidences that the predicted debt model can provide more benefit than simply having a predicted model based solely on utilization and latency.

Remarkably, DATESSO achieves full satisfaction for global constraints on utilization and satisfy global latency for majority of the cases, which are generally superior to the other three. Therefore, for RQ1, we conclude that:

**Answering RQ1:** DATESSO is more effective than the state-of-the-art in terms of the utilization and latency, with better satisfactions. Both the design of formalizing global constraints as the soft ones and the temporal debt-aware utility model have contributed to the improvement.

### 7.5 RQ2: Sustainability of DATESSO

We now assess the sustainability of adaptation achieved by using the accumulated debt and sustainability score. Figure 5 shows the accumulated debt, in which we see that all approaches have accumulated debt in a linear and steady manner. However, clearly, DATESSO results in significantly less debt than the other three as it accumulates overtime, suggesting that DATESSO favours component services that is less likely to contribute to global constraint violation in the long-term basis.

Table 2 shows the sustainability scores for all approaches. As can be seen, despite that DATESSOand DOA have similar total number of constant violations, DATESSO has achieved the best Score\(n,m\) value among others. This implies that the adaptations in DATESSO would create the greatest benefit in mitigating per violation. All the above conclude that:

**Answering RQ2:** DATESSO is more sustainable than the other three, as it has less accumulated debt and with the highest sustainability score. This means that DATESSO favors more reliable component services in a long-term basis, and that it offers greater benefit when dealing with each violation overall.

### 7.6 RQ3: Running Time of DATESSO

Figure 6 illustrates the running time for all approaches. We can clearly see that RBC is the slowest due to the region based algorithm. TLHCA is the 2nd slowest because of the frequent need of replacing all component services. Since DATESSO and DOA differ only on whether having the debt calculation, they have similar running overhead (\(p > .05\)) but are significantly faster than the others. This is because only the problematic abstract services, along with those component services that satisfy all considered timesteps, are
involved in the actual search, which reduces the search space. However, as we have shown, DATESSO offers much better performance and sustainability than DOA. In summary, we have:

Answering RQ3: DATESSO and DOA both have similar running time, but they are faster than the other two.

8 THREATS TO VALIDITY

Threats to construct validity can be related to the metric and evaluation methods used. To mitigate such, we use a broad range of metrics for evaluating different aspects of DATESSO, including utilization, latency and sustainability etc. To examine the effectiveness of each contribution, we have compared DATESSO with specifically designed approaches, i.e., TLHCA and DOA, in addition to a direct implementation of existing work (ROC). Further, we plot all the data points in a trace, and applied statistical test and effect size interpretation when it is difficult to show all the data points.

Threats to internal validity can be mainly related to the value of the parameters for DATESSO. Particularly, the setup has been designed in a way that it produces good trade-off between the quality and overhead. They have been shown to be reasonable following preliminary runs in our experiments. The future timestep $m$ is also specifically tailored and the used value tends to be sufficient. However, it is worth noting that the actual future timesteps to use is updated dynamically depending on whether there is a feasible component service that satisfies all considered constraints.

Threats to external validity can be associated with the environment and the dataset that are used in the experiment. To improve generalization, we apply commonly used service-based system [12][11][20], whose data is randomly sampled from the real-world WS-Dream dataset [36], along with the FIFA98 workload trace [7]. A more comprehensive evaluation on different dataset and more complicated structures are parts of the future work.

9 RELATED WORK

Self-adapting service composition is certainly not new for research on service-based systems. Among others, Li. et al. [22] propose approach that search for composition plans that satisfy both the local and global constraints, which may frequently requires to re-compose the entire service composition. Lin et al. [23] and Li et al. [22] both rely on region-based composition, in which an expand region algorithm is proposed to identify the region of each component service, which forms a reduced search space. Dai et al. [16] present a self-healing approach for service composition. They applied time-series prediction on workload when reasoning about the self-adaptation. The commonality of the above work is the fact that they all assume both local and global constraints are hard ones during reasoning, which can be over-optimistic. Therefore, they can easily lead to the situation of ‘no satisfactory composition plans found’. DATESSO, in contrast, formalizes the global one as soft constraint, which mitigates the issues of over-optimism and also reward some plans that may temporarily cause global violation, but tends to be more sustainable with larger long-term benefit.

The most closely related work that also consider technical debt metaphor in service composition is attempted by [4] in which they used real options theory for managing technical debt in the service composition for the cloud-based services. However, the approaches are merely designed for design time analysis rather than runtime self-adaptation. Skourletopoulos et al. [27] describe an approach for evaluating the technical debt in mobile cloud-based service. The problem is formulated by analyzing the linear growth of users in system modeling. However, they did not consider time-varying and accumulated properties of the debt.

In summary, the key additions in DATESSO are that

- DATESSO formalizes different strictness for the two levels constraint in service composition.
- DATESSO makes use of a new debt model that was designed based on the different strictness of the two levels constraints and time-series prediction. It is therefore temporal, capable of quantifying accumulated debt and tailored to the problem context.
- Drawing on the above, DATESSO proposes to leverage a simple but effective and efficient reasoning algorithm that reduces the search space and focuses on the long-term benefits of self-adaptation.

The benefits of all the above contributions have been experimentally demonstrated in Section 7.

10 CONCLUSION

In this paper, we proposed a debt-aware two level constraint reasoning approach, dubbed DATESSO, for self-adapting service composition. DATESSO formalizes the global constraints as the soft ones while leaving only the local ones as hard constraints. Such formalization is then used to built a temporal debt-aware utility model, supported by time-series prediction. The utility model, together with the different strictness of the two level constraints, enable us to design a simple yet efficient and effective reasoning algorithm in DATESSO. Experimental results demonstrate that DATESSO is more effective that state-of-the-art in terms of utilization, latency and running time, while being about to make each self-adaptation more sustainable.

In future work, we seek to extend DATESSO for better synergy with stochastic multi-objective search algorithms, which provides better results on scenarios with complex trade-off surface.