Self-awareness for dynamic knowledge management in self-adaptive volunteer services

Abdessalam Elhabbash, Rami Bahsoon, Peter Tino
School of Computer Science
University of Birmingham
United Kingdom
{a.elhabbash, r.bahsoon, p.tino}@cs.bham.ac.uk

***Abstract*— Engineering volunteer services calls for novel self-adaptive approaches for dynamically managing the process of composing and/or allocating volunteer services. As these services tend to be published and withdrawn without restrictions; uncertainties, dynamisms and 'dilution of control' related to the decisions of selection and composition are complex problems. These services tend to exhibit periodic performance patterns, which are often repeated over a certain time period. Consequently, the awareness of such periodic patterns enables the prediction of the services performance leading to better adaptation. In this paper, we contribute to a self-adaptive approach, namely time-awareness, which combines self-aware principles with dynamic histograms to dynamically manage and maintain the periodic trends of services performance and their evolution trends over time. Such knowledge can inform the adaptation decisions; leading to increase in the precision of selecting and composing services. We evaluate the approach using a volunteer storage composition scenario. The evaluation results show the advantages of dynamic knowledge management in self-adaptive volunteer computing in selecting dependable services and satisfying higher number of requests.**

Keywords—dependability; service composition; self-adaptive; self-aware

I. INTRODUCTION

Volunteer Computing (VC) is an emerging distributed computing paradigm in which users make their own resources available to others enabling them to do distributed computations and/or storage [1]. In the literature, many approaches have been proposed to enable volunteers to donate their resources for scientific projects, e.g. SETI@Home [2] and Storage@Home [3], among the others. The paradigm is believed to be an enabler for cost-effective large scale computation and sharing for storage, leveraging on spare resources that can be available and idle on the users' computing devices (e.g. PCs, laptops, smart phones, etc.). The paradigm has been seen as an alternative for purchasing resources in large scale projects, where utilizing volunteered resources can bring the benefits of large scale inexpensive and shared computing [4].

Volunteered resources can be composed together to satisfy users' requests in many service-oriented applications such as the cloud and smart cities [5], a practice, which we term as volunteer service composition (VSC). Engineering Volunteer Services (VSs) calls for novel self-adaptive approach for dynamically and adaptively managing the processes of selecting, composing, and allocating VSs and underlying resources. The approach shall address the following fundamental requirements, which caters for dynamisms and uncertainties in requests and service provision. More specifically, these approaches shall fundamentally address:

- Resources-awareness: the contributed resources should be composed and/or allocated to users, achieving both maximum utilization and minimum waste with minimum computation time.
- Availability-awareness: Resources availability tends to be uncertain and dynamic in VC. This is because, the publishers often contribute their resources during the time intervals in which they do not need those resources, i.e. the volunteered resources are not available permanently [6].
- Dilution of control: As volunteer services are offered on a voluntary basis by individuals and organizations willing to participate in the model, VC tends to exhibit 'dilution' of control increasing the level of uncertainty and the dynamisms of the provision. This is because volunteered resources can be offered and withdrawn at any time [7]. The right without the symmetric obligation to participate in VC makes Service Level Agreements (SLAs) less stringent as when compared to commercial services; adding further complexity.
- Dependability-awareness: Because of the dilution of control requirement, dependability information of the services, in terms of the level of providing the promised resources, should be collected and used in VS composition/allocation approaches.

In [8] the authors have reported the presence of periodic patterns in the performance of the volunteering hosts based on a long-term study. The study analyzed a large set of traces taken from the SETI@Home [2] real system. The patterns are usually repeated over a certain time period that varies from one volunteer to another. Such period can be some hours, days, or weeks. We argue that the awareness of such periodic patterns enables the prediction of the services performance, which helps to reason about the selection and adaptation decisions. However, taking into account that different volunteers contribute their resources to different systems we can deduce that traces collected from one project cannot be used in another

one, as the volunteers are different. These circumstances motivate the need for an approach that captures and manages the knowledge at runtime. This also requires data structures suitable to represent the dynamically acquired and managed knowledge such as the dynamic histograms [9]. Dynamic histograms are constructs that dynamically approximate data distributions at runtime. They have been used in database management systems' applications in order to maintain and represent the data which continuously arrive and vary with time.

Recently, self-awareness and self-expression concepts have been receiving more attention in computing systems [10]. Self-awareness is defined as the combination of (1) the knowledge on the internal state of the system and the execution environment, (2) the ability to predict the changes in the system, and (3) the ability to adapt to the changes [11]. Thus, self-awareness can provide self-adaptive systems with primitives for proactive management and behavioral control at runtime. It can also improve both the accuracy and quality of adaptation. This may in turn converge the system towards more desirable stable states.

In our previous work [12], we developed utility models for VSC, which can inform the problem of dynamically selecting and composing 'good enough' services. In [13], we proposed a framework to enable self-adaptation in VC inspired by a general framework leveraging self-awareness in computing [14]. As part of the framework, two self-adaptive approaches have been proposed to deal with uncertainty associated with VC environment. The first is the stimulus-aware VSC, which is considered as a baseline approach. The second is the 'classic' time-aware VSC, which leverages the historical observations on services' performance at the selection and composition phases. The 'classic' time-aware approach assumes the availability of historical records, which is not always the case. In this paper we take this work further: we start from zero-history and instead accumulate that history at runtime. For this purpose, we make novel use of dynamic histograms to capture the evolving knowledge on the services' performance.

The novel contribution of this paper is the time-aware approach that leverages principles of self-awareness and use dynamic histograms to dynamically acquire and manage knowledge on VSs' performance in self-adaptive VC. Our approach treats knowledge of self-adaptive VC as "moving target" that can change and evolve over time and uses this information to better inform the adaptation. This can consequently improve the quality and precision of adaptation in dynamic and uncertain environment. Specifically, we make the following novel contributions:

Firstly, unlike existing works in VC which assume the presence of historical records on services performance, we assume that the system starts from zero-history and accumulates the knowledge at runtime.

Secondly, we use dynamic histograms to capture the evolving knowledge at runtime. The dynamic histograms enable capturing the periodic patterns of the VS in case they exist.

Thirdly, we develop a method to estimate the services performance that takes into consideration the periodic patterns and the time interval in which the requester intends to use the required volunteered resources.

Fourthly, building on a scenario from VSC, we evaluate the time-aware approach and compare it with the stimulus-aware approach, which is considered as a baseline adaptive approach in VC. The results show that the time-aware approach results in satisfying higher number of requests and better resources utilization. However, it produces certain overhead in terms of computation time.

The remainder of this paper is organized as follows: the next section introduces the motivating scenario. Section III presents the dynamic histogram and its evolution operations. In section IV we present the self-aware architectural framework. In section V we briefly introduce our utility model. Section VI introduces the self-aware VSC approaches. Section VII shows the evaluation results. Related works are outlined in section VIII and we conclude in section IX.

II. MOTIVATING SCENARIO

We motivate the need for integrating the concepts of self-awareness into VC, using a situation in which volunteered storage are offered as services. Assume a heterogeneous environment, which consists of varied computing nodes like PCs, laptops, smart phones, etc. These nodes are connected via a network. Individual people owning these nodes, known as publishers, offer their idle storage resources as services using a publish/subscribe model. Assume a subscriber needs to do some computations and store data temporarily but she has insufficient storage. To overcome this issue, she can explore the network searching for volunteer storage services to use. If she finds the required storage, while satisfying her requirements (e.g. location, security etc.), she will request it for her use. Otherwise, volunteered storages can be composed together to form a total storage that meets the subscriber's needs. Fig. 1 shows an example in which the subscriber S_1 submits a request to search for storage of 40 GB. To make this volume available to S_1, the composer service, named *FindSpace4Me*, inspects the published services and returns three possible composition strategies:

1^{st}: Using the storage promised by VS_1.
2^{nd}: Composing the storages promised by VS_2, VS_3, and VS_4.
3^{rd}: Composing the storages promised by VS_2 and VS_5.

Now, which strategy should be selected to satisfy the request? One possibility is to randomly pick any of them. Then,

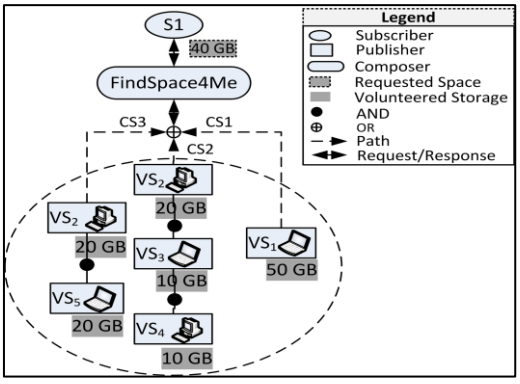

Fig. 1. Motivating Scenario - Composition request of S1 for 40GB.

when a service violates the requirements, the system initiates an adaptation action to repair the corresponding strategy. However, a question arises here about the feasibility of that adaptation action, i.e. will the undertaken adaptation result in better performance? On the other hand, if the system is able to anticipate the performance of the services, then it can select a strategy so that violations are less likely to occur, thus avoiding the violations. Also, the deeper the knowledge the system has on the services performance, the more intelligent the decision will be. In this context, self-awareness can be adopted to reason about the self-adaptation actions; henceforth, enabling intelligent selection and adaptation decisions. For example, assume that S_1 submitted a request at time t_1, and assume that the performance of VS_5 is anticipated to be poor at t_1, then the system will avoid the selection of the third strategy. But, if we assume that S_1 submitted a request at time t_2, and assume that the performance of VS_5 is anticipated to be well at t_2, then the system may select the third strategy. In our work we implement this scenario to demonstrate the adoption of the self-awareness in VC.

III. DYNAMIC HISTOGRAM EVOLUTION

As mentioned we use dynamic histograms to dynamically manage the knowledge on the services historical performance. These histograms typically consist of buckets, which are created/merged at runtime using a method based on Chebyshev's inequality [15]. Then, the data stored in each bucket are used to estimate the service's performance. In this section we give a brief background on the histograms and dynamic histograms and Chebyshev's inequality. Then, we present the dynamic histogram evolution operations we have developed.

A. Background

1) Histograms and dynamic histograms

A histogram is an estimate of the data distribution of a certain variable. Given a certain dataset, a histogram divides its data into subsets called buckets based on a partitioning rule. Dynamic Histograms have been proposed to capture and estimate the data distribution in evolving datasets. In such cases, data points arrive continuously and the dataset is built incrementally over time [9]. Dynamic histograms are continuously updated to tackle the changes in the evolving datasets. The main idea in the dynamic histograms is to reconstruct the buckets, which involves splitting and/or merging buckets, at runtime based on the partitioning rule of the histogram in order to keep the properties of the histogram.

The time-aware approach (explained below), dynamically divides the services usage time into time intervals at runtime. The time intervals correspond to the dynamic histograms buckets. The captured knowledge on the services performance is then stored in the buckets, which results in splitting/merging buckets based on the number of data points in those buckets.

2) Chebyshev's inequality

Suppose that we have a set of N data points for a random variable (e.g. observations of a service's performance) but the distribution of the random variable is unknown: We estimate the expected value using the data points and we can use the Chebyshev's inequality in order to know how close the estimated expected value is to the actual one [16]. In other words, Chebyshev's inequality bounds the probability that a random variable deviates from its expected value by a sufficiently small positive number ε, called confidence threshold. Mathematically, Chebyshev's inequality is expressed as:

$$P\left(\left|E(X) - \hat{E}(X) \geq \varepsilon\right|\right) \leq \frac{\sigma^2}{N.\varepsilon^2} \qquad (1)$$

where $E(X)$ is the actual expected value, $\hat{E}(X)$ is the estimated expected value, σ is the standard deviation, N is the number of data points, and ε is the confidence threshold.

In our approach, we use Chebyshev's inequality in a different way. Our purpose is to know when the number of data points in a bucket in the dynamic histogram will be sufficient to give a close estimate of the expected value, which helps to decide when to split the bucket and evolve the histogram. The corresponding method is presented in the next section.

B. Evolution operations

As mentioned, the system starts from 'zero-history' and then the knowledge is captured and managed incrementally at runtime using the dynamic histograms. We adopt a dynamic histogram for each service in order to continuously insert the observed data points taking into account the time interval in which the data point has been observed. Then the continuous update of the dynamic histogram, by splitting and/or merging the buckets, results in refining the histogram structure and capturing the periodic performance pattern of the services. Accordingly, a data point is defined as follows:

Definition 1. (Data point) A data point is a tuple of $(T = [a, b], value)$ where T is the time interval in which the observation has been recorded, a is the start date of T and b is the end date, and $value$ is the value of the performance metric.

The update process of the dynamic histogram involves inserting a new data point into the appropriate bucket(s), splitting a bucket when the number of data points is sufficient to estimate the performance, and merging each empty bucket with a neighbour one. In the following, we describe each of the mentioned operations and show the corresponding algorithm.

1) Insert new data point.

Based on definition 1, a data point might fall into one or more buckets depending on the intersection between the data point time interval and the bucket(s) boundaries. Algorithm 1 is used to find the appropriate bucket(s) in which the data point will be inserted.

Algorithm 1 Find Appropriate Buckets

Input: Dynamic Histogram *dhist*, Data Point *dp*
Output: Array *appropriateBuckets*
1: **for all** *bucket* in *dhist* **do**
2: // check if the time intervals of *dp* and *bucket* intersect
3: **if** *dp.start_date* < *bucket.end_date* && *dp.end_date* > *bucket.start_date* **then**
4: add *bucket* to *appropriateBuckets*
5: **end if**
6: **end for**
7: **return** *appropriateBuckets*

2) Split a bucket.

When the number of data points in a bucket is sufficient to estimate the performance in the corresponding time interval, then dividing the bucket into smaller buckets will provide more accurate estimation of the performance. For example, consider a time interval of one year. Dividing the one year into (for example) twelve time intervals (each represents one month) will provide more in-depth knowledge on the services performance instead of treating the one year as a one time interval. To resume, the sufficient number of data points in a bucket is determined using the following method which is based on Chebyshev's Inequality. Given the confidence threshold ε and the probability of confidence $P\left(\left|E(X) - \hat{E}(X) \geq \varepsilon\right|\right)$ and solving (1) we will have:

$$N \geq \frac{\sigma^2}{P\left(\left|E(X) - \hat{E}(X) \geq \varepsilon\right|\right).\varepsilon^2} \quad (2)$$

We can bound the variance σ^2. Assuming the worst case; the variance is maximum when one half of the values is at lowest possible and the other half is at the highest possible value. In this work we express the performance in terms of dependability, which will be defined in the next section. Based on that, the lowest value of the performance is 0.0 and the highest is 1.0. As a result, the maximum variance is 0.25 and the splitting threshold $split\_th$ is given by:

$$split\_th = \frac{0.25}{P\left(\left|E(X) - \hat{E}(X) \geq \varepsilon\right|\right).\varepsilon^2} \quad (3)$$

Consequently, when the number of data points in a bucket exceeds $split\_th$, the bucket will be split using Algorithm 2.

Algorithm 2 Split Bucket

Input: Bucket *bucket*
Output: Bucket $bucket_1$, Bucket $bucket_2$
1: **for all** data point *dp* in *bucket* **do**
2: Add *dp.start_date* and *dp.end_date* to *temp_array*
3: **end for**
4: Find *min(temp_array)* and *max(temp_array)*.
5: Calculate *spliting_date* = (*min*(*temp_array*) + *max*(*temp_array*)) / 2.
6: Create Bucket $bucket_1$ such that $bucket_1$*.start_date* = *bucket.start_date* and $bucket_1$*.end_date* = *spliting_date*
7: Create Bucket $bucket_2$ such that $bucket_2$*.start_date* = *spliting_date* and $bucket_2$*.end_date* = *bucket.end_date*
8: **for all** data point *dp* in *temp_array* **do**
9: **if** *dp.time_interval* intersects with $bucket_1$*.time_interval*
10: Insert *dp* into $bucket_1$
11: **end if**
12: **if** *dp.time_interval* intersects with $bucket_2$*.time_interval*
13: Insert *dp* into $bucket_2$
14: **end if**
15: **end for**
16: Delete *bucket*
17: **return** $bucket_1$ and $bucket_2$

3) Merge empty buckets

If the splitting operation resulted in an empty bucket, then that bucket will be merged with its preceding neighbour. If the empty bucket does not have a preceding neighbour, it will be merged into the following one.

Pseudo-code for the update method of the dynamic histogram is presented in Algorithm 3.

Algorithm 3 Dynamic Histograms Update

Input: Dynamic Histogram *dhist*, Data Point *dp*
Output: Updated version of *dhist*
1: *appropiateBuckets* = FindApprpoitaeBuckets (*dp*, *dhist*)
2: **for all** Bucket $bucket_i \in$ *appropiateBuckets* **do**
3: insert *dp* in $bucket_i$
4: **if** $bucket_i.size \geq split\_th$ **then**
5: Bucket[ ] *temp_array* ← SplitBucket($bucket_i$)
6: $bucket_1 \leftarrow temp\_array[0]$; $bucket_2 \leftarrow temp\_array[1]$
7: Replace $bucket_i$ by $bucket_1$ and $bucket_2$
8: Set the successor and predecessor buckets for $bucket_1$ and $bucket_2$
9: **end if**
10: **end for**
11: **for all** Bucket $bucket_i$ in *dhist* **do**
12: **if** $bucket_i$ is empty **then**
13: Merge $bucket_i$ with its successor or predecessor
14: **end if**
15: **end for**
16: **return** *dhist*

IV. SELF-AWARE VSC ARCHITECTURAL FRAMEWORK

In [17] we proposed a general framework for self-aware service composition which enables knowledge collection and representation for reasoning about adaptation in service composition. In this section we provide a quick review of the self-aware framework. The architectural diagram of the proposed framework is illustrated in Fig. 2. The framework consists of the following basic components:

- Internal/external sensors: The sensors are responsible for collecting data on the services engaged in a composition (internal) and the services available in the service repository (external). The data include any changes in the promised quality of service (internal) and the offering of new services in the service repository (external). Then the collected data are passed to the stimulus- and time-aware levels in the self-awareness component.
- Self-awareness: This component models the knowledge collected by the sensors and passes the learnt models to the self-expression component. The stimulus-awareness level represents the basic level of awareness i.e. this level enables the system to respond to the events received from the sensors. The time-awareness level assumes the presence of the stimulus-awareness and adds more awareness by considering the historical performance of the services in terms of dependability, which is defined in the next section. Thus, the time-awareness level enables the system to take more intelligent adaptation decisions by selecting services that exhibited better performance historically.
- Self-expression: this component performs the actual adaptation actions based on the learnt models received from the self-awareness component.

The problem of dynamic knowledge management in self-adaptive systems is still a pending issue. Dynamic knowledge management includes capturing the evolving performance datasets, which usually starts from zero-knowledge, and

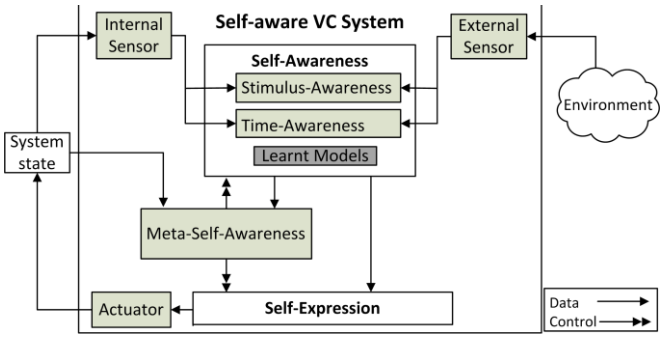

Fig. 2. Self-aware VSC framework

dynamically updating the models learnt from these datasets. In this paper, we provide an approach for dealing with this pending issue using the dynamic histograms, namely, the time-aware approach.

V. VSC FORMULATION

In this section we introduce a set of definitions in order to formulate the VSC problem.

A. Basic Definitions

Definition 2. (Service/Request Attributes). In the presence of the identical functionality of the VSs, the services' attributes are the criteria used to discriminate between services when a request is submitted. In our model, we use three generic attributes for this purpose, namely, Storage, Availability Time, and Reputation. However, other criteria can be defined without fundamental changes.

1) Storage. Given a volunteer service VS_i , the storage Stg_i is the size of the volunteered storage in Megabytes where $Stg_i > 0$.

2) Availability Time. Given a volunteer service VS_i, T_i is the time interval $[a_i, b_i]$ in which VS_i is available, where a_i is the start date and b_i is the end date.

3) Reputation. Given a volunteer service VS_i, Rep_i is the reputation level of the service which is reported from the subscribers after their use of VS_i *where* $0 \leq Rep_i \leq Rep_{max}$ *and* Rep_i, $Rep_{max} \in \mathbb{N}$. The representative reputation of a certain service VS_i is the average of the subscribers' feedback on VS_i.

Definition 3. (Volunteer Service). A volunteer service VS_i, is a 3-tuple (Stg_i, T_i, Rep_i) where Stg_i is the volunteered storage space, T_i is the time interval $[a_i, b_i]$ in which the VS_i is available, and Rep_i is the reputation level of the service. A service repository (SR) is a set of disjoint volunteer services. We denote a SR with n services as $SR = \{VS_1, VS_2, \ldots, VS_n\}$. In this paper we denote the Stg_i, T_i, and Rep_i as the attributes of the service or the quality of the service.

Definition 4. (Subscriber's Request). A subscriber's request R is a 2-tuple (Stg^R, T^R), where Stg^R denotes the storage space required in the time interval $T^R = [a^R, b^R]$.

Definition 5. (Composite Service). Given a subscriber's request R, a Composite Service CS is a set of VSs, $\{VS_1, VS_2, \ldots, VS_k\}$, such that the following global constraints are satisfied (denoted as $CS \vdash R$):

- $VS_i \in SR, 1 \leq i \leq k \leq n$
- $\sum_{i=1}^{k} Stg_i \geq Stg^R$, at any time instant in $[a^R, b^R]$.
- $a^R \geq min[a_i]$ and $b^R \leq max[b_i]\ \forall\ VS_i\ \in CS$.

B. Static VS Selection and Composition

In this section we briefly introduce the utility model and the composition approach we developed in [12]. This model is the base for our self-adaptive VSC approaches.

The utility model provides a systematic approach for selecting the VSs that are composed to satisfy a request. The idea is to measure the amount of contribution that each service exhibits to satisfy the request. When a subscriber submits a composition request, the system creates an empty CS, retrieves the available services from the service repository, and computes the utility for each service attribute using the utility functions (4) and (5). In this paper, we term these utilities as the promised utilities.

1) Storage utility:

$$U_{stg}(VS_i) = \begin{cases} e^{-\beta(Stg_i - Stg^R)}, & Stg_i \geq Stg^R \\ e^{\alpha(Stg_i - Stg^R)}, & Stg_i < Stg^R \end{cases} \quad where\ 0 < \beta < \alpha < 1 \tag{4}$$

2) Time utility

$$U_{time}(VS_i) = \begin{cases} 0, & b_i \leq a^R\ or\ b^R \leq a_i \\ e^{\gamma(b_i - b^R)}, & a_i < a^R, a^R < b_i < b^R \\ e^{\gamma(a^R - a_i)}, & b_i > b^R, a^R < a_i < b^R \\ \dfrac{e^{\alpha(b_i - a_i)}}{e^{\alpha(b^R - a^R)}}, & a_i \geq a^R, b_i \leq b^R \\ \dfrac{e^{-\beta(b_i - a_i)}}{e^{-\beta(b^R - a^R)}}, & otherwise \end{cases} \quad where\ 0 < \beta < \gamma < \alpha < 1. \tag{5}$$

After computing the utilities, the system finds the non-dominant set of services using (6), and randomly selects one of them and adds it to *CS*. Then, if the above global constraints are satisfied, the system returns *CS* to the subscriber, otherwise the process is repeated. If no composite service can be found to satisfy the request R, the system notifies the subscriber.

$$maximize\ \{U_{stg}(VS_i),\ U_{time}(VS_i), Rep(VS_i)\} \tag{6}$$

VI. SELF-AWARE VOLUNTEERED SERVICES COMPOSITION

In this section, we present the self-adaptive VSC approaches, namely, the stimulus- and the time-aware VSC as a realization of the corresponding awareness levels of the self-aware framework.

A. Stimulus-aware VSC

The service selection in this approach is based on the promised utilities of the volunteers. When a subscriber submits a request R, the system searches for a composite service that satisfies *R* as described in the above section.

With regards to self-adaptability, the stimulus-aware adaptation is considered as the basic level of adaptation as it is

the adaptation approach supported in the current volunteer computing systems [13]. The adaptation actions are limited to replacing the violating service by another one in order to maintain the corresponding composite service. To clarify, when a change in the promised storage or availability of a service VS_i occurs, the self-expression initiates an adaptation action in order to replace the violating service VS_i by re-executing the above steps. If the adaptation process is successful, then the violating service is replaced, otherwise the subscriber is notified that the violation cannot be treated.

B. Time-aware VSC

The aim of the time-aware approach is to use the historical performance of the services to select the most appropriate services, i.e. services that provide what they promise. In our approach, we express the services performance in terms of *dependability*. We consider a service VS_i to be dependable if VS_i provides the storage and availability it promises. In this section, we introduce a formal definition of dependability then the time-aware VSC approach.

1) VS dependabilities

The dependability evaluation provides a useful method for examining the behaviour of the service provider, i.e. the volunteer. We consider a service VS_i to be dependable if VS_i provides the storage and availability it promises. We use the *dependability* measure to express the extent to which a selected service fulfils the promised resources and quality of service. As the deviation from the promised quality can be in any attribute, there will be a dependability measure for each service attribute. We introduce the definition of dependability as follows. Given that a volunteer service VS_i has been selected in a composite service CS to serve the request R. Assume that $U^P_{stg}(VS_i)$ is the storage utility promised by the volunteer of VS_i. Assume also that the actual storage utility provided by VS_i, captured by the self-aware framework sensors, during serving R is $U^A_{stg}(VS_i)$. Then the storage dependability of VS_i, $D_{stg}(VS_i)$, is defined as in (7). The availability time dependability, $D_{time}(VS_i)$, is defined similarly as in (8).

$$D_{stg}(VS_i) = \begin{cases} \dfrac{U^P_{stg}(VS_i) - U^A_{stg}(VS_i)}{U^P_{stg}(VS_i)}, U^A_{stg}(VS_i) < U^P_{stg}(VS_i) \\ 1, \quad Otherwise \end{cases} \tag{7}$$

$$D_{time}(VS_i) = \begin{cases} \dfrac{U^P_{time}(VS_i) - U^A_{time}(VS_i)}{U^P_{time}(VS_i)}, U^A_{time}(VS_i) < U^P_{time}(VS_i) \\ 1, \quad Otherwise \end{cases} \tag{8}$$

2) Knowledge management using dynamic histograms

Our aim is to capture the periodic performance patterns of the VSs, in terms of dependabilities, so that the system can use such historical knowledge to determine the time intervals in which a service is most likely to fulfil the request requirements and the time intervals in which that service is most likely to violate the request requirements. To achieve that, a dynamic histogram is created for each service attribute. Initially each dynamic histogram contains one bucket, then the dynamic histogram evolves by dividing/merging buckets as the dependabilities' data points arrive. For each service, a new data point will arrive in two cases, (i) a service violates the promised utilities or (ii) a request, in which the service is involved to satisfy, has been satisfied. In both cases, the dependabilities will be computed using (7) and (8) and inserted into the appropriate bucket(s) using Algorithm 3. After a certain period of time, the dynamic histogram converges to a state in which the buckets represent the service's pattern periods. The length of the convergence period depends on how often the service is used.

3) Time-aware service selection

When a subscriber submits a request, the following key steps are executed in order to satisfy the request:

Step 1: For each $VS_i \in SR$, compute the U_{stg} and U_{time} using the utility functions (4) and (5) respectively. Compute also the average reputation $Rep(VS_i)$.
Step 2: For each $VS_i \in SR$ find the appropriate buckets from the corresponding dynamic histogram. Each bucket overlaps with request interval is considered an appropriate bucket.
Step 3: For each bucket, estimate the representative D_{stg} and D_{time} for each $VS_i \in SR$ by counting the number of data points which have a value greater than or equal to the dependability threshold D_{th}, which is provided by the subscriber, and dividing that number by the total number of data points in the bucket.
Step 4: Find the average storage dependability, AVD_{stg}, for each VS_i by summing the representative storage dependability of each bucket and dividing over the number of buckets. Similarly find AVD_{time}.
Step 5: Find the non-dominant set of services using (9), select one of them randomly, and add it to CS.

$$\begin{aligned} maximize\{&U_{stg}(VS_i),\ U_{time}(VS_i), Rep(VS_i) \\ &AVD_{stg},\ AVD_{time}\} \end{aligned} \tag{9}$$

After executing the above steps, the subscriber request will be partially satisfied, then the request requirements will be recalculated in order to update the remaining requirements, and the above steps will be repeated to select the next service. After selecting each service, the global constraints (see Definition 5) will be checked. If they are satisfied, the composite service CS will be returned; otherwise the above steps will be repeated. If all the services are visited and the global constraints are still not satisfied, an empty CS will be returned and the subscriber will be notified that the request cannot be satisfied.

4) Time-aware adaptation

The self-adaptability in the time-aware approach is two-fold, in terms of the question: *"When should we adapt?"*

a) Reactive adaptation: When a change in the promised quality of a service is reported to the time-awareness component (see Fig. 2), the actual utilities will be computed using (4) and (5) and subsequently the dependabilities using (7) and (8). Then the dependabilities will be stored in the corresponding dynamic histogram. After that, an adaptation action will be carried out by the self-expression component. This adaptation action involves executing the time-aware service selection steps (section VI.B.3) in order to replace the service that violated the requirements.

b) Proactive adaptation: The system performs proactive adaptation in order to adapt a composite service before a violation occurs. The proactive adaptation is trigged in two

cases, (1) the dependability of a service involved in a CS is expected to drop, according to the performance pattern captured in the service dynamic histogram, or (2) a service has become availabe in the SR which is expected to perform better than an existing one, according to its perfomance patterns. In both cases, the system will execute the time-aware service selection steps (section VI.B.3) in order to adapt the CS.

VII. Experimental Evaluation

In this section, we conduct experiments in order to evaluate the performance of the stimulus-aware and time-aware approaches using simulations. Simulating the selection and adaptation actions has the advantage of conducting scalable experiments which are expensive to conduct on real systems. However, the results can be used to guide the real application. The experiments were conducted on a desktop PC with an Intel core i5-3570 3.5 GHZ processor, 4G RAM, Windows 7, Java Standard Edition V1.7.0.

A. Experimentations Context

We implement the example described in section II as a publish/subscribe model in which n services are published and m subscribers request their composition goals. The attributes' values of the n services were generated randomly. TABLE I. shows the ranges of the services attributes values. In the experiments we assume that 1000 service are available, which is a reasonable number to indicate scalability compared with the literature, e.g. the number of services in [18] is 5. We vary the number of requests m. The performance of the services is assumed to have a periodical daily or weekly pattern according to the long term data traces analysis conducted in [8]. For each test case, the experiment was conducted 100 times and the average was computed.

B. Comparison Criteria

The experiments compare the above approaches in the following criteria:

1) Average dependability: defined as the average summation of the dependability of each selected VS divided by the number of selected VSs. This metric is a pointer to the correctness of the time-aware approach. That is the time-aware approach should tend to select the more dependable services.

2) %Satisfies requests: defined as the number of requests that the system can successfully satisfy divided by the total number of requests. This metric is related to the efficiency of selecting services, i.e. selecting dependable services will lead to fewer violations and hence more requests will be satisfied.

TABLE I. Range Of Attributes Values

	Service		Subscriber	
Attribute	***min***	***max***	***min***	***max***
Storage	5	20	20	30
Time	1 Jan.	31 Dec.	1 Jan.	31 Dec.
Reputation	0	4	-	-

3) Time cost: defined as the average summation of the time needed to generate the composite services and the time needed to adapt to the constraints violations (whether reactively or proactively) in milliseconds.

C. Results and Discussion

1) Comparison in average dependability: The first set of experiments evaluates the tendency of selecting the the highly dependable services over time. Fig. 3 (a), (b), and (c) shows the average dependability of the selected services for varying number of requests m. The figure shows that the average dependability in the two approaches is nearly the same in the initial interval of the simulation time. The reason in that the knowledge size is zero or small to efficiently predict the services performance. After a while of accumulating the knowledge, the average dependability in the time-aware case gets higher than the stimulus-aware case. Therefore, the time-aware approah has the advantage of selecting the dependable services when the required knowledge about the services dependability becomes available.

2) Comparison in throughput: The second set of experiments compares the throughput over time. Fig. 4(a), (b), and (c) shows the average throughput for m requests. The figures show that the average throughput is almost the same in the initial period of time. After a while of accumulating the knowledge, the average throughput in the time-aware case gets higher than the stimulus-aware case. Thereforce, having the advantage of selecting the dependable services using the time-aware appoach results in reducing the constraints violations and so satisfying more requests.

3) Comparison in time cost: The third set of experiments evaluates the time cost of the two approaches. Suppose that we have n services, t buckets for each service in the dynamic histogram, and p data points in each bucket. For the stimulus-aware approach the time complexity is $O(n)$. For the time-aware approach the time complexity is $O(ntp)$ in the worst case in theory. These time complexities are for one iteration of the corresponding algorithms since the presented VSC

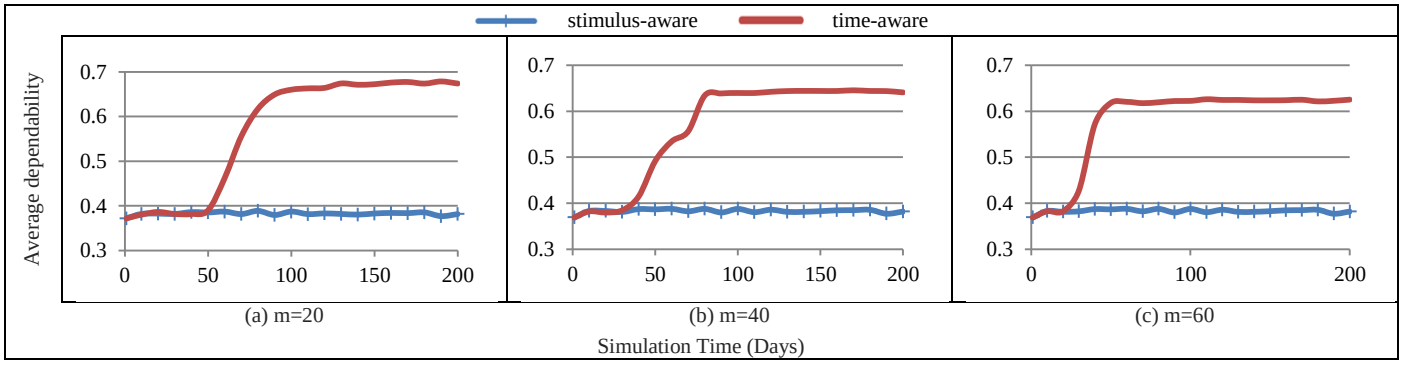

Fig. 3. Comparison in average dependability

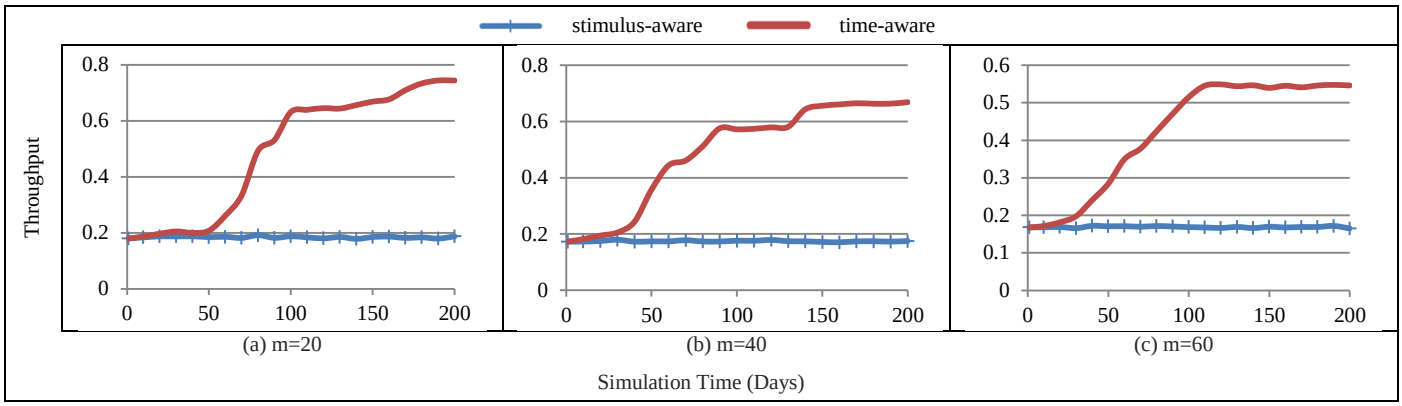

Fig. 4. Comparison in throughput

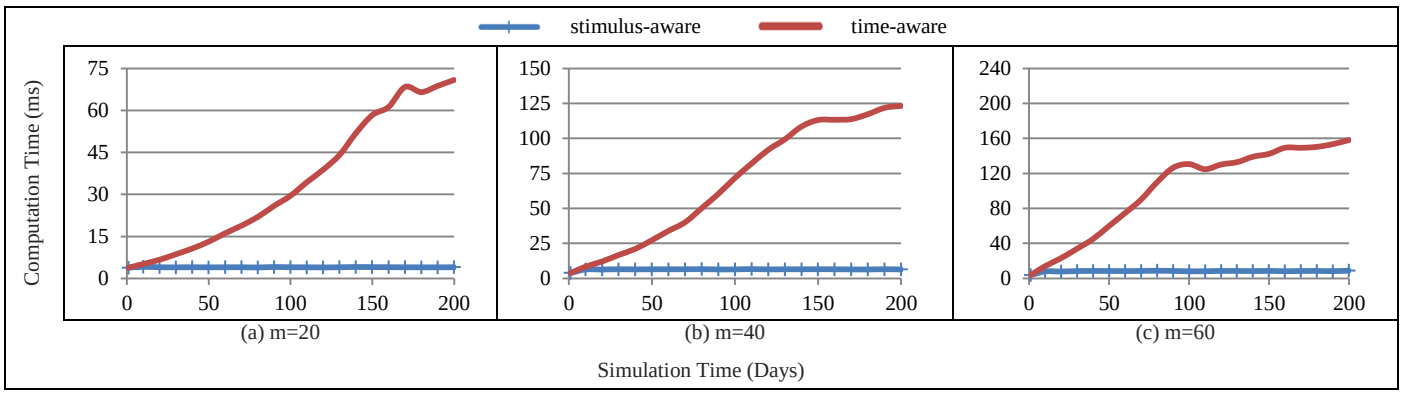

Fig. 5. Comparison in time cost.

approaches are greedy-approaches, i.e. the number of iterations needed to satisfy the request is unknown. Turning to the simulation results, Fig. 5 (a), (b), and (c) show the time cost for varying *m* requests. The figure shows that the stimulus-aware approach has the least time cost. It is notable also that the time cost in the time-aware approach increases linearly over time, whereas the time cost in the stimulus-aware approach is not affected.

4) Discussion: The experimentation results show that using the dynamic histograms for dynamic knowledge management helps to refine the performance models learnt at runtime. As the previous figures show, the advantages of selecting dependable services and satisfying more requests are noticed after the knowledge in the dynamic histograms is refined. However, the improvements are accompanied with an overhead which is mainly the time cost of updating the dynamic histogram. In our on-going research, a brain-like component, called meta-self-awareness, will be implemented in the self-awareness framework in order to assess whether the level of overhead is acceptable compared to the users' requirements. Then, it will be the responsibility of the meta-self-awareness component to switch between the awareness approaches based on overhead/advantages assessment.

VIII. Related Works

A. Volunteer Computing Paradigms

In this section, we present an overview of the deemed volunteer computing frameworks. BOINC is the earliest VC middleware [19]. It enables for creating public-resource computing projects. Through this middleware, users can participate their PCs and specify their contributions to the projects. SETI@home [2], Storage@home [3], and others are examples of VC projects that use BOINC. Cloud@Home [20] is a computing paradigm that has been proposed to enable resource sharing on either voluntary or commercial basis. Social Cloud [21] is a paradigm that takes advantage of pre-existing social networks trust relationships to share resource among users. None of the existing approaches address the dynamisms of the volunteering environment and the knowledge management problem. They do not provide answers on how to deal with the changes in the environment and how to adapt to those changes.

B. Self-adaptive Frameworks

The increased complexity of service-oriented applications stimulated the researchers to investigate for self-adaptive solutions; resulted in an extensive literature on self-adaptive systems. The dominant ones have been surveyed in [22] [23]. In the event that knowledge acquisition and management are pre-requisites for self-adaptation, self-awareness has been considered as an enabler for self-adaptation. Early works on self-awareness, e.g. [24], [10], and [25] outlined the vision for designing systems with built in self-aware systems. Afterwards, many approaches have been proposed to extend and realize the self-awareness vision. For example, In [26] Bicocchi et al. proposed an awareness framework for knowledge collection and classification in urban environments to reason about adaptation and to improve the energy efficiency in pervasive scenarios. In [27], Kounev et al.

presented a model-based approach to designing self-aware systems using an architecture-based modelling language. Works in [28] and [29] intended to characterize different levels of awareness. For example, in [29] Lewis et al. proposed a reference framework for architecting self-aware systems. The architecture defines different levels of awareness inspired by Neisser's five human self-awareness levels [30] which enables fine-grain knowledge representation. Our work extends this framework by realizing the different levels of awareness [29].

IX. CONCLUSION

We have contributed to a self-adaptive approach, namely the time-awareness, which makes novel use of the principles of self-awareness and dynamic histograms to dynamically manage knowledge in self-adaptive VC. The approach is able to capture the evolving knowledge and manage it at runtime using dynamic histograms. The knowledge is used to reason about the reactive and proactive adaptation decisions. The experimental results show that the time-aware approach can bring to a self-adaptive application the advantages of satisfying more requests since it tends to select services that exhibit high dependability and low probability of violating the requests constraints. A scenario of volunteered storage composition is introduced to illustrate and evaluate the approach. In future work, we will work on implementing other levels of awareness, e.g. meta-self-awareness which will act as a brain that enables switching between different awareness levels at runtime.

REFERENCES

[1] O. Nov, D. Anderson, and O. Arazy, "Volunteer computing: a model of the factors determining contribution to community-based scientific research," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 741-750.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, pp. 56-61, 2002.

[3] A. L. Beberg and V. S. Pande, "Storage@ home: Petascale distributed storage," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, p. 482.

[4] M. Nouman Durrani and J. A. Shamsi, "Volunteer computing: requirements, challenges, and solutions," *Journal of Network and Computer Applications*, vol. 39, pp. 369-380, 2014.

[5] N. Mitton, S. Papavassiliou, A. Puliafito, and K. Trivedi, "Combining Cloud and sensors in a smart city environment," *EURASIP Journal on Wireless Communications and Networking*, vol. 2012, pp. 1-10, 2012/08/08 2012.

[6] S. Dustdar, C. Dorn, F. Li, L. Baresi, G. Cabri, C. Pautasso, and F. Zambonelli, "A roadmap towards sustainable self-aware service systems," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2010, pp. 10-19.

[7] S. Sebastio, M. Amoretti, and A. L. Lafuente, "AVOCLOUDY: a simulator of volunteer clouds," *Software: Practice and Experience*, 2015.

[8] D. Lázaro, D. Kondo, and J. M. Marquès, "Long-term availability prediction for groups of volunteer resources," *Journal of Parallel and Distributed Computing*, vol. 72, pp. 281-296, 2012.

[9] D. Donjerkovic, Y. Ioannidis, and R. Ramakrishnan, "Dynamic Histograms: Capturing Evolving Data Sets," in *Data Engineering, 2000. Proceedings. 16th International Conference on*, 2000, pp. 86-86.

[10] F. Zambonelli, N. Bicocchi, G. Cabri, L. Leonardi, and M. Puviani, "On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles," in *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on*, 2011, pp. 108-113.

[11] S. Kounev, X. Zhu, J. O. Kephart, and M. Kwiatkowska, "Model-driven algorithms and architectures for self-aware computing systems (Dagstuhl Seminar 15041)," *Dagstuhl Reports*, vol. 5, 2015.

[12] A. Elhabbash, R. Bahsoon, P. Tino, and P. R. Lewis, "A Utility Model for Volunteered Service Composition," presented at the Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, 2014.

[13] A. Elhabbash, R. Bahsoon, P. Tino, and P. R. Lewis, "Self-Adaptive Volunteered Services Composition through Stimulus-and Time-Awareness," in *Web Services (ICWS), 2015 IEEE International Conference on*, 2015, pp. 57-64.

[14] T. Chen, F. Faniyi, R. Bahsoon, P. R. Lewis, X. Yao, L. L. Minku, and L. Esterle, "The handbook of engineering self-aware and self-expressive systems," *arXiv preprint arXiv:1409.1793*, 2014.

[15] C. Siang Yew, P. Tino, and Y. Xin, "Measuring Generalization Performance in Coevolutionary Learning," *Evolutionary Computation, IEEE Transactions on*, vol. 12, pp. 479-505, 2008.

[16] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," *SIGMOD Rec.*, vol. 25, pp. 294-305, 1996.

[17] A. Elhabbash, R. Bahsoon, and P. Tino, "Towards Self-Aware Service Composition," in Proceeding of *High Performance Computing and Communications, 2014*, pp. 1275-1279.

[18] Z. u. Rehman, O. K. Hussain, and F. K. Hussain, "Parallel Cloud Service Selection and Ranking Based on QoS History," *International Journal of Parallel Programming*, vol. 42, pp. 820-852, 2013.

[19] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, 2004, pp. 4-10.

[20] S. Distefano and A. Puliafito, "Cloud@Home: Toward a Volunteer Cloud," *IT Professional*, vol. 14, pp. 27-31, 2012.

[21] K. Chard, K. Bubendorfer, S. Caton, and O. F. Rana, "Social Cloud Computing: A Vision for Socially Motivated Resource Sharing," *Services Computing, IEEE Transactions on*, vol. 5, pp. 551-563, 2012.

[22] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, pp. 184-206, 2015.

[23] F. D. Macías-Escrivá, R. Haber, R. del Toro, and V. Hernandez, "Self-adaptive systems: A survey of current approaches, research challenges and applications," *Expert Systems with Applications*, vol. 40, pp. 7267-7279, 2013.

[24] T. Becker, A. Agne, P. R. Lewis, R. Bahsoon, F. Faniyi, L. Esterle, A. Keller, A. Chandra, A. R. Jensenius, and S. C. Stilkerich, "EPiCS: Engineering proprioception in computing systems," in *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, 2012, pp. 353-360.

[25] S. Kounev, F. Brosig, N. Huber, and R. Reussner, "Towards Self-Aware Performance and Resource Management in Modern Service-Oriented Systems," in *Services Computing (SCC), 2010 IEEE International Conference on*, 2010, pp. 621-624.

[26] N. Bicocchi, D. Fontana, and F. Zambonelli, "A self-aware, reconfigurable architecture for context awareness," in *Computers and Communication (ISCC), 2014 IEEE Symposium on*, 2014, pp. 1-7.

[27] S. Kounev, N. Huber, F. Brosig, and X. Zhu, "Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures," *IEEE Computer Magazine*, 2016.

[28] J. Schaumeier, J. Pitt, and G. Cabri, "A Tripartite Analytic Framework for Characterising Awareness and Self-Awareness in Autonomic Systems Research," in *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2012 IEEE Sixth International Conference on*, 2012, pp. 157-162.

[29] P. R. Lewis, A. Chandra, F. Faniyi, K. Glette, C. Tao, R. Bahsoon, J. Torresen, and Y. Xin, "Architectural Aspects of Self-Aware and Self-Expressive Computing Systems: From Psychology to Engineering," *Computer*, vol. 48, pp. 62-70, 2015.

[30] U. Neisser, "The Roots of Self-Knowledge: Perceiving Self, It, and Thoua," *Annals of the New York Academy of Sciences*, vol. 818, pp. 19-33, 1997.