Service Orchestration Using the Chemical Metaphor

Jean-Pierre Banâtre[1], Thierry Priol[1], and Yann Radenac[2⋆]

[1] INRIA/IRISA
Campus de Beaulieu, F-35042 Rennes Cedex, France.
jean-pierre.banatre@irisa.fr, thierry.priol@irisa.fr

[2] Research Center for Grid and Service Computing
Institute of Computing Technology, Academy of Sciences
Beijing 100080, P.R. China.
yann.radenac@software.ict.ac.cn

Abstract. Service-oriented architectures (SOA) provide sets of operations through a network. A program built mainly upon calling services is called an orchestration of services. Different programming languages can be used to be the "glue" between services in an orchestration. This article shows how a programming language inspired by a chemical metaphor can be used to program service orchestration.

1 Introduction

Service-based infrastructures are shaping tomorrow's distributed computing systems. It is rather difficult to come up with a strict definition, widely accepted, of what is a service. In the scope of this paper, a service can be seen as a set of operations that are available on a machine or through a network. After several attempts to design distributed programming paradigms, such as remote-procedure call [1], distributed objects [2] or distributed components [3], the service paradigm seems to solve one of the main issue when dealing with distributed systems: how to design loosely-coupled distributed applications based on the composition of a set of independent software modules called services that are spread over a set of resources available on a network such as the Internet. The loosely-coupled aspect is important when dealing with a distributed system. It allows an application to adopt a late binding to software modules. Services are discovered and brokered at runtime and bound when needed. This provides a lot of flexibility enabling the selection of the best services, in terms of Qualities of Service (QoS) such as performance and cost, but also to cope with failures since a given service can be replaced at runtime. It is foreseen in the near future that the programming of distributed applications will be just the expression of the composition of services available on the Internet. In fact, the Internet will be

⋆ This work was partially carried out for the EchoGRID IST project n°045520, funded by the European Commission.

considered as a large scale computing system that shares some similarities with the microprocessors we have in our desktop or laptop computers but of course with a larger computing granularity. Internet will provide access, acting as a bus, to a large number of processing and storage units under the form of utility computing systems such as Grids [4] or Cloud computers like the Google [5] and Amazon [6] ones. To conclude with this analogy, services could be considered as the instruction set of such distributed computing infrastructures. With such analogy, the issue that is immediately emerging is how to express the instruction and data flows? Another issue that prevents us to go further in the analogy between a microprocessor and a service-based computing infrastructure is that failures can occur at any time and it is considered as a basic property of any distributed system.

Expressing the control and data flows, or simply workflows, in such large scale distributed computing infrastructures is thus challenging in many aspects. We think that existing approaches to express workflows need to be rethought to take into account the large scale dimension of these infrastructures allowing massively parallel coarse-grain computations and the dynamicity due to frequent failures. This paper investigates the use of an unconventional approach which is chemical programming that possesses two nice properties: it is implicitly parallel and autonomic. It gets its inspiration from the chemical metaphor, formally represented here by a chemical language called HOCL which stands for Higher-Order Chemical Language [7]. In HOCL, computation is seen as reactions between molecules in a chemical solution. HOCL is higher-order: reaction rules are molecules that can be manipulated like any other molecules, i.e., HOCL programs can manipulate other HOCL programs. Reactions only occur locally between few molecules that are chosen non-deterministically. The execution is implicitly parallel since several reactions can occur simultaneously and it can also be be seen as chaotic and possesses nice autonomic properties as shown in [8]. This model has already been applied in the contexts of Grid workflow enactment [9, 10] and of Desktop Grids [11] and shown its suitability to express coordination of computations.

The objective of this paper is to show, through an example, that chemical programming can be a good candidate for service programming, such as the composition and coordination of services. On one side, applications are programmed in an abstract manner describing essentially the chemical coordination between (not necessarily chemical) abstract services. On the other side, chemical programs are specifically provided to the service run-time system in order to obtain the expected qualities of service in terms of efficiency, reliability, security, etc. These programs can be seen as special coordination programs providing guidelines to the runtime system allowing a better use of resources in order to obtain the expected Quality of Service.

Section 2 introduces chemical programming through the HOCL language. Section 3 shows how to orchestrate services using HOCL. In section 4, we present briefly some of the main existing approaches to express workflow of services in the Web Services framework. Finally, we conclude in Section 5.

2 Chemical Programming Model

A chemical program can be seen as a (symbolic) chemical solution where data is represented by floating molecules and computation by chemical reactions between them. When some molecules match and fulfill a reaction condition, they are replaced by the result of the reaction. That process goes on until an inert solution is reached: the solution is said to be inert when no reaction can occur anymore. Formally, a chemical solution is represented by a multiset and reaction rules specify multiset rewritings.

We use a higher-order chemical programming language called HOCL [7]. HOCL is based on the γ-calculus [12], a higher-order chemical computation model which can be seen as an higher-order extension of the Gamma language [13]. In HOCL, every entity is a molecule, including reaction rules.

A program is a molecule, that is to say, a multiset of atoms $(A_1, \ldots, A_n)$ which can be constants (integers, booleans, etc.), sub-solutions ($\langle M \rangle$) or reaction rules. Compound molecules (M_1, M_2) are built using the associative and commutative (AC) operator ",". The corresponding AC laws formalize the Brownian motion and can always be used to reorganize molecules.

The execution of a chemical program consists in triggering reactions until the solution becomes inert.

A reaction involves a reaction rule $\mathbf{one}\, P\, \mathbf{by}\, M\, \mathbf{if}\, V$ and a molecule N that satisfies the pattern P and the reaction condition V. The reaction consumes the rule and the molecule N, and produces M. Formally:

$$(\mathbf{one}\, P\, \mathbf{by}\, M\, \mathbf{if}\, V),\ N \longrightarrow \phi M \quad \text{if } P \text{ match } N = \phi \text{ and } \phi V$$

where ϕ is the substitution obtained by matching N with P. It maps every variable defined in P to a sub-molecule from N. For example, the rule in the following solution

$$\langle 0,\ 10,\ 8,\ \mathbf{one}\, x{::}\mathrm{Int}\, \mathbf{by}\, 9\, \mathbf{if}\, x > 9 \rangle$$

can react with 10 (the variable x is mapped to 10). They are replaced by 9. The solution becomes the inert solution $\langle 0,\ 8,\ 9 \rangle$.

A molecule inside a solution cannot react with a molecule outside the solution (the construct $\langle . \rangle$ can be seen as a membrane). A HOCL program is a solution which can contain reaction rules that manipulate other molecules (reaction rules, sub-solutions, etc.) of the solution.

In the remaining of the paper, we use some syntactic sugar such as declarations $\mathbf{let}\, x = M_1\, \mathbf{in}\, M_2$ which is equivalent to M_2 where all the free occurrences of x are replaced by M_1. The reaction rules $\mathbf{one}\, P\, \mathbf{by}\, M\, \mathbf{if}\, C$ are one-shot: they are consumed when they react. Their variant denoted by $\mathbf{replace}\, P\, \mathbf{by}\, M\, \mathbf{if}\, C$ are n-shot, i.e., they do not disappear when they react (like in Gamma).

There are usually many possible reactions making the execution of chemical programs highly parallel and non-deterministic. Since reactions involve only a few molecules and react independently of the context, many distinct reactions

can occur at the same time. For example, consider the following program that computes the prime numbers lower than 10 using a chemical version of the Eratosthenes' sieve:

$$\begin{array}{l}\textbf{let}\ \mathsf{sieve} = \textbf{replace}\, x, y\, \textbf{by}\, x\, \textbf{if}\, x\ \mathrm{div}\ y\, \textbf{in} \\ \langle \mathsf{sieve}, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle\end{array}$$

The rule sieve reacts with two integers x and y such that x divides y, and replaces them by x (i.e., removes y). Initially several reactions are possible, for example $\mathsf{sieve}, 2, 8$ (replaced by $\mathsf{sieve}, 2$) or $\mathsf{sieve}, 3, 9$ (replaced by $\mathsf{sieve}, 3$) or $\mathsf{sieve}, 2, 10$ or etc. The solution becomes inert when the rule sieve cannot react with any couple of integers in the solution, that is to say, when the solution contains only prime numbers. The result of the computation in our example is $\langle \mathsf{sieve}, 2, 3, 5, 7 \rangle$.

To access within a sub-solution (e.g., to get the result of a sub-program), a reaction rule has to wait for its inertia. That means that a reaction rule matches only inert sub-solutions. For example, if we want to compute the largest prime number lower than 10, we can use the previous program as a sub-program, i.e., a sub-solution, and then compute the maximum of its result:

$$\begin{array}{l}\textbf{let}\ \mathsf{sieve} = \textbf{replace}\, x, y\, \textbf{by}\, x\, \textbf{if}\, x\ \mathrm{div}\ y\, \textbf{in} \\ \textbf{let}\ \mathsf{max} = \textbf{replace}\, x, y\, \textbf{by}\, x\, \textbf{if}\, x \geq y\, \textbf{in} \\ \langle \langle \mathsf{sieve}, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle,\ \textbf{one} \langle \mathsf{sieve} = s,\ \omega \rangle\, \textbf{by}\, \omega,\ \mathsf{max} \rangle\end{array}$$

Initially, the one-shot rule cannot react with the non-inert sub-solution, and only reactions inside the sub-solution can occur. When the sub-solution becomes inert, the one-shot rule matches the sub-solution: the variable s matches the rule named sieve and the variable ω matches all the remaining atoms of the solution (the prime numbers). In the reaction, the one-shot rule and the sub-solution are replaced by the prime numbers (ω) and the rule max which, in turn, triggers new reactions until one element remains. More formally, the execution steps occur as follows:

$$\begin{array}{c}\langle \langle \mathsf{sieve}, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle,\ \textbf{one} \langle \mathsf{sieve} = s,\ \omega \rangle\, \textbf{by}\, \omega,\ \mathsf{max} \rangle \\ \downarrow * \\ \langle \langle \mathsf{sieve}, 2, 3, 5, 7 \rangle,\ \textbf{one} \langle \mathsf{sieve} = s,\ \omega \rangle\, \textbf{by}\, \omega,\ \mathsf{max} \rangle \\ \downarrow \\ \langle 2, 3, 5, 7, \mathsf{max} \rangle \\ \downarrow * \\ \langle 7, \mathsf{max} \rangle\end{array}$$

This example shows the higher-order property of HOCL: the one-shot rule removes and adds other (named) rules. This property allows to express coordination of chemical programs within the language.

The examples provided here are simple and fine grain in order to illustrate the mechanisms of chemical programming through HOCL. But HOCL can also be used as a coordination language of services (coarse grain).

3 Service Orchestration Using HOCL

A service orchestration is a program that describes a coordination of services. HOCL can be used as a data-driven coordination language according a chemical metaphor. For example, the previous HOCL examples can be viewed as data-driven coordination of integers and of functions on integers (div, $\geq$).

3.1 Coordination with HOCL

In [9], workflows are expressed as chemical program. It shows that all coordination mechanism of workflow can be translated into a chemical setting. The enactment of workflows can also be described by a chemical program. In fact, many classical coordination mechanism can be expressed as a chemical coordination [14]: sequential execution, parallel execution, mutual exclusion, atomicity, message passing, shared memory, rendez-vous, Kahn networks, etc.

HOCL programs are self-organizing systems [8]. When a HOCL program reaches an inert state (i.e., a stable state), and if then some new molecules are added (i.e., perturbation of the system), then some new reactions happen with the new molecule until a new inert state is reached (i.e., a new stable state). A simple mail system has been developed as an example of programming self-organization with HOCL. It features self-healing, self-optimizing, self-protection and self-configuration.

HOCL as a coordination language has also been applied to program Desktop Grids. A Desktop Grid is made of non dedicated resources (e.g., any personal computer connected on the Internet). Such a grid can be highly volatile and non reliable. In [11], HOCL is used as a coordination language to specify the execution of a simple ray-tracer in a Desktop Grid. The HOCL program contains rules that adapt the on-going executions of programs according to the availability of resources in the Desktop Grid.

3.2 Orchestrating services with HOCL

Principle. A chemical service architecture consists of a solution of services, i.e., a solution of sub-solutions, each representing one service (cf Figure 1).

A service is represented by a solution that contains molecules performing the operations that this service proposes. To call a service, one adds a molecule of the form Call:s:n:p in the solution representing the called service, where s is the calling service, n is the identifier of the call for the calling service, and p are the parameters for the operation to be performed.

When a service makes a call to another service it generates a molecule of the form ExtCall:s:n:p where s is the called service, n is the identifier of the call for the current (calling) service, and p are the parameters for the operation to be performed.

At a given time, a service may be running different computations related to different simultaneous call. That's why, to prevent a service to mix the computation and the result of different and independent calls, each call has a unique identifier n.

```
let withdrawServiceCall =
      replace serv1:⟨ExtCall:serv2:n:param, w⟩
           by serv1:⟨w⟩, Call:serv1:serv2:n:param
in
let depositServiceCall =
      replace Call:serv1:serv2:n:param, serv2:⟨w⟩
           by serv2:⟨Call:serv1:n:param, w⟩
in
⟨withdrawServiceCall, depositServiceCall,
 Service1:⟨...⟩, ..., ServiceN:⟨...⟩
⟩
```

Fig. 1. Generic chemical service architecture.

A call is performed in two steps by two rules. The rule withdrawServiceCall extracts an ExtCall molecule from a service sub-solution and puts it in the main solution. The rule depositServiceCall takes a call in the main solution and forwards it into one corresponding service sub-solution.

Two remarks:

- A call to a service to perform an action, and a call to a service to provide a result are just two ordinary calls: there is no distinction between the two calls. In fact, a call message and its result message are not explicitly coupled like in a RPC for example.
- According to the semantics of HOCL, a rule may react only with an *inert* solution. So the rules withdrawServiceCall and depositServiceCall could only react with inert sub-solutions, i.e., with services that do not do any computation, i.e., with services that only perform communication (input or output of calls). In fact, that inertia constraint may be released for these two rules. These two rules manage molecules that represent messages. So these molecules are independent of computations happening inside the solutions representing the services. Adding or removing a call from these solutions do not depend on the internal state of these solutions. So adding a call to or removing a call from a solution that represents a service can happen even if the solution is not inert.

A travel organizer example. Let's take the example of a travel organizer (cf Figure 2). This travel organizer makes the reservations of a flight and a hotel according some parameters provided by a user.

The service is a solution named TravelOrgService. It contains an integer used as a counter to provide unique identifiers to separate different molecules related to different calls. For each call to the travel organizer service, the rule findFlightHotel generates two calls: one call to a flight service and one call to a hotel service. It also updates the counter, and stores the reference to that

```
let findFlightHotel =
      replace Call:s:m:p, n
           by ExtCall:FlightService:n:param,
              ExtCall:HotelService:n:param,
              s:m:n, (n+1)
in
let resultFlightHotel =
      replace Call:FlightService:n:f,
              Call:HotelService:n:h,
              s:m:n
           by ExtCall:s:m:(f:h)
in
TravelOrgService:⟨0, findFlightHotel, resultFlightHotel⟩
```

Fig. 2. A travel organizer service in HOCL.

call as a molecule $s{:}m{:}n$ where $s{:}m$ identifies the calling service, and n is the identifier to this call to the travel organizer service. The rule resultFlightHotel reacts when the results are available. The results appear as two calls: one from a flight service, and one from a hotel service. They both concern the same initial call identified by the counter n. Then the rule generates a call back to the calling service s with its identifier m, and the flight and hotel results $f{:}h$.

Execution example. We describe here a possible execution of the travel organizer example (cf Figure 3). The system is represented by a solution that contains the rules withdrawServiceCall and depositServiceCall that perform the calls, the sub-solutions representing the services (the travel organizer service, the flight services and the hotel services), and two calls to the travel organizer by two different users.

```
⟨ withdrawServiceCall, depositServiceCall,
  TravelOrgService:⟨findFlightHotel⟩,
  FlightService:⟨Name:AirFrance, ...⟩,
  FlightService:⟨Name:BritishAirways, ...⟩,
  HotelService:⟨Name:Accor, ...⟩,
  ...,
  Call:UIService1:TravelOrgService:(Dates1:Places1:Pref1),
  Call:UIService2:TravelOrgService:(Dates2:Places2:Pref2)
⟩
```

Fig. 3. Running the travel organizer service in HOCL.

Two users have queried a search for their travel and the system has added the respective calls of the form Call:UIServiceX:TravelOrgService:... into the main solution, where UIServiceX is the identifier of the user interface service that has emitted the call to the travel organizer service TravelOrgService, where DatesX, PlacesX are the dates and places constraints for the required travels, and PrefX some additional preferences. Initially, the rule depositServiceCall can forward the two calls to the travel organizer service. Then the travel organizer will generate the calls to the flight services and the hotel services using the rule findFlightHotel. Then the rule withdrawServiceCall will extract the calls to the main solution, and the rule depositServiceCall will forward these calls to a corresponding service. After, some reactions these services will generate their result inside their solution as an ExtCall molecule addressed to the travel organizer. The rules withdrawServiceCall and depositServiceCall will then bring these messages to the travel organizer. When both result from the flight service and the hotel service are available inside the TravelOrgService for the same initial call, the rule resultFlightHotel will generate an external call to the service that invoked the travel organizer service. Finally, the rule withdrawServiceCall will extract that result from the travel organizer service solution and put it in the main solution, so that it is available to the external world (outside the main solution).

At any time, at execution time in particular, some new services may be added inside the main solution, and some services may be removed from the main solution. This is not a problem, since a coupling between a call and a corresponding service is dynamic and non deterministic. The service type of the called service must be satisfied: for example, several services provide a flight service, and a call to a service flight may react with any of them.

4 Related Work

Coordination and composition of services have attracted a lot of attention of both the industry and the academia. Several approaches have been proposed following either the orchestration or the choreography paradigms. These two paradigms differ from their execution scenario which is mainly centralized for the first one and distributed for the later.

Starting from industry-led initiatives, the standard approach to compose Web Services is the Business Process Execution Language, WS-BPEL [15]. WS-BPEL is a language that provides several powerful control flow structures such as condition, loops, switches and activities, such as Web Service invocation, can be executed either sequentially or concurrently. In addition, WS-BPEL provides variables to store temporary data and fault compensation. WS-BPEL is very verbose largely due to XML root and it shares some similarities with programming languages. Its level of abstraction is rather very low forcing programmers to “think parallel” and to anticipate all possible failures during the workflow execution. Finally, since WS-BPEL is about orchestration, workflow execution is centralized thanks to a WS-BPEL engine. Regarding choreography, the main

standard today is the Web Service Choreography Description Language, WS-CDL [16]. Choreography models the interactions and dependencies between a set of services by describing their exchanges of messages. As for WS-BPEL, WS-CDL provides control structures such as sequence, parallel and choice. Loops are allowed thanks to the WorkUnit activity that provides a way to repeat the execution of an activity depending on a guard condition. As for WS-BPEL, WS-CDL is based on XML and thus is verbose with a low level of abstraction.

On the research side, there is a vast amount of work dealing with Web Service composition. One of the main drawback of WS-BPEL is its lack of formal semantics. In [17, 18], a formal semantics of some of the WS-BPEL features, such as the specification of events, fault and compensation handler behaviors or transactions, are introduced. As chemical programming takes its root from rewriting systems, we can mention the work presented in [19] that describes a dynamic service customization and composition framework for Web services based on a rule-based service integration language with concepts borrowed from rewriting systems. Composition of services using an Event-Condition-Action rule based approach, that is even closer to chemical programming, is described in [20]. Self-coordination of Web Services using a Linda-like tuple space, similar to a multiset in our approach, is introduced in [21].

5 Conclusion and Future Work

Originally, the Gamma formalism was invented as a basic paradigm for parallel programming [13]. It was proposed to capture the intuition of a computation as the global evolution of a collection of atomic values evolving freely. Gamma appears as a very high level language which allows programmers to describe programs in a very abstract way, with minimal constraints and no articial sequentiality. Later, it became clear that a necessary extension to this simple formalism was to allow elements of a multiset to be Gamma programs themselves, thus introducing higher-order. This lead to the HOCL language used in this paper.

Basically, the chemical paradigm (as introduced in HOCL) offers four basic properties: mutual exclusion, atomic capture, parallelization and serialization. These properties have been exploited in [14] in order to give a chemical expression of well known coordination schemes.

Along the same lines, this paper investigates the utilization of the Chemical Programming Model, in order to describe Service Coordination. We develop this idea with a simple, yet practical, example dealing with travel organization. The example developed throughout section 3 shows that our approach provides a very abstract and generic way of programming service orchestration. This is made possible due to the higher order property of HOCL. Programs (services) can be handled naturally by appropriate synchronization rules. Here, we have limited our investigations to service orchestration, it is clear that we could have tackled more elaborated synchronization schemes dealing with service choreography.

References

1. Birrell, A.D., Nelson, B.J.: Implementing remote procedure calls. ACM Trans. Comput. Syst. **2**(1) (1984) 39–59
2. OMG: The Common Object Request Broker: Architecture and Specifica tion V3.0. Technical Report OMG Document formal/02-06-33 (June 2002)
3. Open Management Group (OMG): CORBA components, version 3. Document formal/02-06-65 (June 2002)
4. Foster, I., Kesselman, C., eds.: The Grid 2: Blueprint for a New Computing Infrastructure. 2nd edn. Morgan Kaufmann Publishers (2003)
5. : Google app engine. `http://code.google.com/appengine`.
6. : Amazon services. `http://aws.amazon.com`.
7. Banâtre, J.P., Fradet, P., Radenac, Y.: Generalised multisets for chemical programming. Mathematical Structures in Computer Science **16**(4) (August 2006) 557–580
8. Banâtre, J.P., Fradet, P., Radenac, Y.: Chemical specification of autonomic systems. In: Proc. of the 13th Int. Conf. on Intelligent and Adaptive Systems and Software Engineering (IASSE'04). (2004)
9. Németh, Z., Pérez, C., Priol, T.: Workflow enactment based on a chemical metaphor. In: The 3rd IEEE International Conference on Software Engineering and Formal Methods. (September 2005)
10. Németh, Z., Pérez, C., Priol, T.: Distributed workflow coordination: Molecules and reactions. In: The 9th International Workshop on Nature Inspired Distributed Computing, IEEE (2006) 241
11. Banâtre, J.P., Le Scouarnec, N., Priol, T., Radenac, Y.: Towards "chemical" desktop grids. In: Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing (e-Science 2007), IEEE Computer Society Press (2007)
12. Banâtre, J.P., Fradet, P., Radenac, Y.: Principles of chemical programming. In Abdennadher, S., Ringeissen, C., eds.: Proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004). Volume 124 of ENTCS., Elsevier (2005) 133–147
13. Banâtre, J.P., Le Métayer, D.: Programming by multiset transformation. Communications of the ACM (CACM) **36**(1) (January 1993) 98–111
14. Banâtre, J.P., Fradet, P., Radenac, Y.: Classical coordination mechanisms in the chemical model. In: From semantics to computer science: essays in honor of Gilles Kahn. Cambridge University Press (2008)
15. Barreto, C., Bullard, V., Erl, T., Evdemon, J., Jordan, D., Kand, K., Knig, D., Moser, S., Stout, R., Ten-Hove, R., Trickovic, I., van der Rijn, D., Yiu, A.: Web services business process execution language version 2.0. http://www.oasis-open.org/committees/wsbpel (May 2007)
16. Ross-Talbot, S., Fletcher, T.: Web services choreography description language: Primer (Jun 2006)
17. Mazzara, M., Govoni, S.: A case study of web services orchestration. In: Proc. of the 7th International Conference on Coordination Models and Languages, Lecture Note in Computer Sciences. Volume 3454. (Dec 2005)
18. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for ws-bpel. Journal of Logic and Algebraic Programming (January 2007)
19. Chen, J.: Rewrite rules as service integrators. In: Proceeedings of the Rules And Rule Markup Languages For The Semantic web workshop (RuleML 2004), Lecture Note in Computer Sciences. (Dec 2004) 182–187

20. Chen, L., Li, M., Cao, J.: A rule-based workfow approach for service composition. In Springer, ed.: Third International Symposium Parallel and Distributed Processing and Applications (ISPA). Volume 3758 of Lecture Notes in Computer Science. (October 2005) 1036–1046
21. Maamar, Z., Benslimane, D., Ghedira, C., Mahmoud, Q.H., Yahyaoui, H.: Tuple spaces for self-coordination of web services. In: SAC '05: Proceedings of the 2005 ACM symposium on Applied computing, New York, NY, USA, ACM (2005) 1656–1660