MSR-TR-2008-61

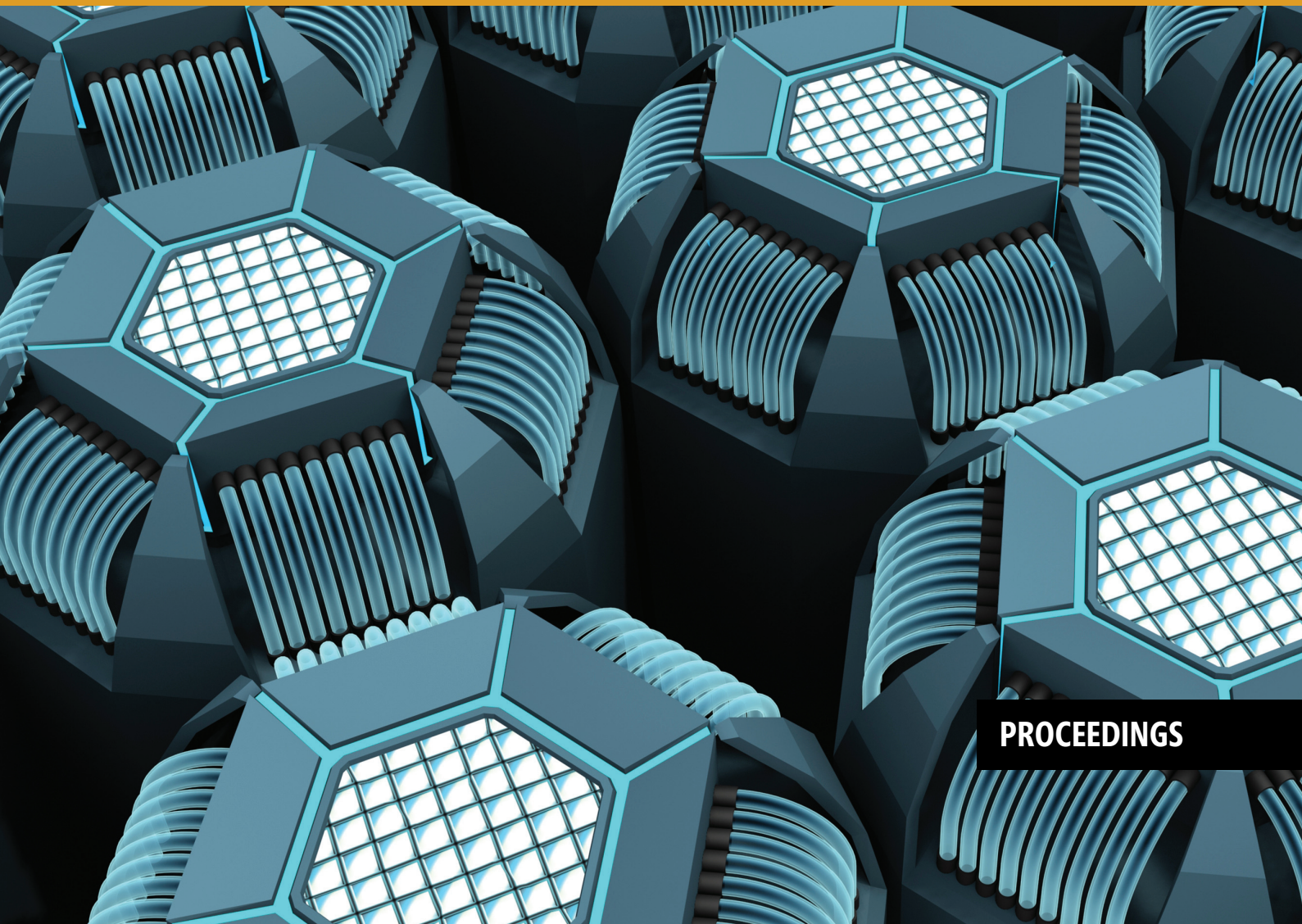
[LABS hp]

Microsoft Research

12-13 May, 2008, Microsoft Research Cambridge UK

The Rise and Rise of the Declarative Datacentre

A RESEARCH MEETING JOINTLY ORGANISED BY MICROSOFT RESEARCH AND HP LABS



PROCEEDINGS

THE RISE AND RISE OF THE DECLARATIVE DATACENTRE

A research meeting jointly organised by
Microsoft Research and HP Labs
May 12-13, 2008,
Microsoft Research Cambridge, UK

Various hardware and software technologies are transforming datacentre management into a programming problem. Network booting and virtualization allow dynamic allocation of hardware resources. Virtual machine mobility helps to optimize hardware utilization, to support maintenance, and to minimize power consumption. Virtual networking isolates mutually distrustful workloads, and allows test to proceed concurrently with production.

These technologies reduce the need for human intervention - indeed, virtual machines can even be rented by the hour over the web, eliminating physical configuration altogether. These hardware and software trends support the abstraction that a datacentre (or even a collection of datacentres) is a single programmable computer, made up of many individual servers managed in software. Going by different names – compute fabric, grid, utility computing, autonomous computing, dynamic systems, cloud – this abstraction has arisen in different research communities, albeit with varying emphases.

To help identify discussion topics at the workshop we invited participants to contribute short (unrefereed) position papers about their insights into the problems of managing this kind of system. This elicited an exciting response from a wide range of disciplines – covering, amongst other topics, practical experiences working with today's datacentres, foundational aspects of security and service design, technology trends, and performance modelling.

Karthik Bhargavan, Andrew D. Gordon, Tim Harris, Peter Toft
Meeting organisers

Contents

Keynote Speaker

Position Papers

Programming the Datacentre

Challenges in System Configuration

Paul Anderson

School of Informatics,University of Edinburgh

dcspaul@ed.ac.uk

Abstract

The hardware in a modern datacentre is highly configurable. Services run on top a virtual infrastructure whose architecture is defined by "programming" the configuration of the hardware. This invites comparison with the task of computer programming. This paper explores the analogy to see what insight it might provide into the datacentre configuration problem.

1. Introduction

In a traditional datacentre, the hardware architecture of the system is relatively fixed. This has to be carefully designed to meet the requirements of the application software. Changes in the system architecture usually involve physical changes to the hardware.

Recently, there has been a strong trend towards *virtualisation*. VLANs overlay the physical network infrastructure. Physical machines host multiple virtual machines, and storage arrays provide virtual disks. Changes in the architecture no longer require physical intervention - the capacity of the virtual machines, their attached disks, and their network connections can all be changed by software. This allows the datacentre to adapt very quickly to varying load and requirements – even if the physical capacity is exceeded, remote resources can be co-opted and configured to absorb the load (Amazon EC2, for example). As well as increasing the flexibility, this can provide huge cost savings through more efficient sharing of resources. Service providers are starting to develop systems such as Flexiscale[1] to exploit these benefits.

System Configuration[2] is a significant problem, even for traditional datacentres. This involves translating the service requirements into detailed configuration parameters for all of the software. As well as the applications, and the middleware, this includes the underlying services and the system software. A typical datacentre will have dozens of "infrastructure services" such as name services, authentication services, file services, time synchronisation services, etc. The relationships between these are complex. Failures, or misconfigurations usually lead to serious failures in the application services. There are currently no good practical solutions to the systems configuration problem – most sites rely on skilled system administrators, aided by a mixture of homegrown scripts and various other ad-hoc tools.

Virtualisation increases both the need for good configuration tools, and the difficulty of creating them. Traditional datacentres have already reached the point where the configuration is too complex to be managed manually. Virtualisation adds another dimension to the problem; the "external" infrastructure supporting the virtual machines has to be configured, as well as the "internal" infrastructure running inside them – and, of course, these both have to be configured in a consistent way. The advantages of the virtual datacentre can only be fully exploited if it can be reconfigured *autonomically* to respond to failures and changes in demand, without manual intervention.

Datacentre configuration as a subject is not well-developed. Current practical tools only address small parts of the problem. They are rarely based on sound principles, and they are largely incompatible. There is no common agreement on the scope of the subject, and different communities have different approaches – "network configuration", "system configuration", or "application configuration" are all subtly different. There are certainly no widely-applicable "standards". But there are a lot of common threads, and a lot of advantages in trying to establish a common framework for thinking about configuration problems.

There is an interesting analogy between system configuration today, and the early development of computer programming – initially the hardware designers would write their own programs. But at some point this became sufficiently complex and important that programming grew into a separate discipline with its own specialists and a profusion of supporting theories. Perhaps this is typical of an emerging discipline – in the case of system configuration, the growing complexity is gradually overwhelming the ability of the practitioners to cope, but theories and formal techniques are still in their infancy.

A modern, virtualised datacentre could be considered as a single programmable entity, with its logical function determined by the "soft" configurable parameters. The problem of configuring the datacentre then looks quite similar to the problem of programming a single machine (or at least a distributed computation). It is interesting to explore this analogy – looking at the similarities and differences – to see if it gives any insight into the system configuration problem.

[1] `http://homepages.inf.ed.ac.uk/dcspaul/publications/ukuug2008.pdf`

[2] See the SAGE booklet on System Configuration: `http://www.sage.org/pubs/14_sysconfig/`

2. Configuration as Programming

Most installations are currently managed at a very "low level" – automatically constructing configurations from "higher-level" specifications of services is rare. Many of the configuration files on a typical machine will have their contents explicitly specified by a human being – or perhaps generated by a very simple process, such as templating. This yields exactly the same problems as early computer programming – the manual translation of service requirements into detailed configurations is slow, error prone, and requires skilled staff. Of course, it also precludes any autonomic reconfiguration. And each installation is sufficiently different that re-use and sharing of configurations between sites is rarely practical.

This immediately leads us to ask whether it is possible to create the equivalent of a "compiler" which will accept some higher-level description of the service configuration and automatically generate the low-level configuration parameters. This has been the focus of a lot of the work in system configuration, and there are some tools which attempt to facilitate this in rudimentary (and different) ways – for example LCFG[3].

One of the difficulties in creating a configuration compiler is to define the nature of the input. At the highest possible level, a functional, "service-oriented" specification might seem appropriate. For example:

- Some client sends an HTTP request.
- The service responds with an appropriate reply.

In practice, such simple specifications are almost always accompanied by other requirements which are much harder to model. At this level, for example, there are likely to be requirements on the performance:

- All HTTP requests should be serviced in N seconds.

And the provision of a real service is subject to a lot of policy which cannot be determined automatically:

- "This service runs badly on model X hardware and I'd prefer it to run on something else *if possible*".
- "I want to keep 1Gb of free disk space on any machine running this service".
- "I want redundant DHCP servers in different buildings".

These additional requirements are the core of the configuration problem and they cannot be ignored. Translating them into concrete specifications for real hardware and software has some analogies with program language compilation, but there are clearly a lot of new challenges.

3. Implementing A Declarative Approach

A declarative approach to traditional programming does not always seem natural – describing the process often feels more appropriate than describing the final state. But in the case of configuration, most people agree that a declarative specification "feels right" - the above examples of service requirements are typical of those seen in real installations, and they describe "desired states", rather than processes.

Ideally, a system administrator would specify some new requirement in a declarative way. The tool would then modify the appropriate device configuration and processes to achieve the desired state. In theory, this seems reasonably straightforward. But there are a lot of issues. Some of these have analogies with programming, and some appear to be significantly new:

[3] `http://www.lcfg.org`

3.1 The appropriate model is not obvious

In conventional programming, there is a clear boundary between the hardware and the software. The underlying processor defines a small and stable model; the semantics is established by the machine code, and this forms the natural target for any higher-level translation. Different high-level languages may use different models at a higher level, but they all inter-operate at this point.

For a datacentre, the appropriate level is not so obvious. Many practical tools operate at a very low-level, simply manipulating disk images, for example. But this has a very limited use, because the semantics of the disk images are opaque. A useful common interface requires a slightly higher level – it needs to provide a meaningful connection to the internal services such as routing, authentication, etc.

In raising the abstraction level of the model, it is important to retain the flexibility of the underlying system, and not to force any particular paradigm. This allows it to be targeted by different tools which may use very different approaches. LCFG, for example, provides a very simple model with virtually no semantics. Tools can be created on top of this to support different high-level approaches. But, the expressive power of LCFG is limited. CIM[4] is an example of a much richer model. But, this enforces a particular paradigm, and is much more complex.

The underlying "machine" can also change rapidly. The introduction of a new version of some system component (DHCP server software, for example) can imply a change in the configuration "machine code" of the datacentre. It must be possible for administrators to adapt quickly to such changes, without the need for specialists and complex software changes.

3.2 The design of a suitable input language is not obvious

The input language needs to be sufficiently powerful to describe all of the necessary requirements, but it needs to be clear and simple enough to prevent misunderstandings and errors in the configuration. A lot of people are involved in the overall configuration, and they have different requirements and skill levels. The same situation occurs in traditional programming, and this has led to a wide choice of languages, suitable for different occasions. But all of these inter-operate and compile to a common machine code.

In the configuration case, the target machine is not so well-defined. There is also much less experience to inform the design of a common target language – at present, different communities tend to use domain-specific approaches which do not inter-work well. But, it seems useful to consider what a common "target language" for configuration might look like.

3.3 The consequences of an error can be serious

An administrator needs to have a good deal of trust in a configuration tool. This implies interfaces which are clear and appropriate to the task in hand. It also requires the ability to "dry run" proposed changes, and to explain any automated reasoning. Of course, the administrator also has to gain confidence in the tool itself. It seems likely that this is something which will only be achieved over time, with the the availability of good, reliable tools – early programmers would often inspect the machine code generated by their high-level programs – at least to query the "efficiency", if not the correctness.

Administrators are also conscious of the power of a configuration tool – any sufficiently powerful tool can probably reconfigure the entire datacentre to the point where the configuration tool itself no longer operates. This has a clear analogy with self-modifying code. But isolating the "code" and the "data" in a configuration

[4] The "Common Information Model":
`http://en.wikipedia.org/wiki/Common_Information_Model_%28computing%29`

tool is not so straightforward – the purpose of the tool is to modify exactly those services upon which it depends.

3.4 Deployment is unreliable

The ”target machine” for a datacentre configuration is highly distributed. A proportion of the devices will almost always be unavailable, or simply uncontactable. As with distributed programming, this requires frameworks and algorithms to isolate the programmer from these considerations. A configuration tool is more intimately bound to the infrastructure itself which makes this more difficult.

Even without failures, the deployment system will have significant latencies – this has important consequences for a declarative approach. The delay between the configuration action and its implementation means that there is often a difference between the “desired configuration” and the “actual configuration”. The term “asymptotic configuration” is very appropriate to describe real systems where the desired configuration typically changes before the previous one has been instantiated. It is very useful to model this explicitly – sometimes, one part of configuration needs to depend on the actual configuration of some other part. Sometimes the appropriate dependency is on the “actual configuration”.

Determining the actual configuration requires a monitoring system, and this itself is subject to failure. This means that there is also a degree of uncertainty in determining the actual configuration.

3.5 Implementing specified behaviour is difficult

It is easy to see how declarative specifications of the state of the system can be mapped into detailed configurations for the devices and processes. But ultimately, we are interested in the *behaviour*. In a real system, this is subject to myriad of external factors.

Implementing a declarative specification of behaviour requires a feedback mechanism to determine the actual state of the system, and more complex reasoning to adjust the configuration accordingly. This is the area of “autonomics”. It is comparatively easy to implement an autonomic solution to any specific problem, but it is less clear what a general framework might look like.

There does not seem to be any very useful analogy with programming here, except perhaps to note that this is similar to a realtime system - the behaviour of the system must be guaranteed in the face of unpredictable external factors. It is certainly possible to statically validate configurations and prove that their behaviour remains within certain bounds when external conditions change. Generating appropriate configurations to meet such requirements is a harder problem.

3.6 Transitional states are important

In a conventional declarative program, we do not care about the state of the system during the computation – this is purely an internal concern. In contrast, the actual transitional states between two desired configurations are usually critical; when changing a network router, it is necessary to configure the new router before decommissioning the old one – otherwise the network breaks so badly that the tool cannot recover.

There does not seem to be a good analogy to this in conventional programming – it is the equivalent of changing the algorithm for a program while it is actually running, without disrupting the computation. The transition between states becomes an AI planning problem with declarative constraints in the configuration at any intermediate stage.

3.7 Declarative thinking is not yet common

We have already said that declarative approach seems to be a more natural way of thinking about configuration specification. But this is not reflected in most current practice. Practical tools from both vendors, and open-source are overwhelmingly concerned with deployment. System administrators (unconsciously) translate their declarative requirements into the steps necessary to implement them. This is so embedded, that it is sometimes difficult for an administrator to explain clearly what they are trying to achieve, rather than what they are “doing”.

Practical declarative programming languages only appeared in the programming world once the technology was in place to implement them effectively. The transition from writing procedural programs, to declarative ones also requires a paradigm change, and effort to understand.

4. Constraints

Constraint programming is one declarative paradigm which seems particularly appropriate. The example requirements from section 2 can be expressed quite naturally as constraints – for example, “I want redundant DHCP servers in different buildings”. This is very attractive for several reasons:

Individual constraints can usually be interpreted in isolation – they don’t depend on the order in which they are specified, and changing one constraint will not invalidate any others. This allows the administrator to have a lot of confidence that a simple policy can be absolutely enforced without worrying about dependencies. For example: “no students should be allowed to log in to the mail server”.

It is also important for autonomics that specifications are not over-constrained. If a configuration defines machine X to be the mail server, then the autonomic system cannot recover when machine X fails – it is not aware of any suitable replacements. On the other hand, if the mail server is defined simply to be any machine with certain properties, then the autonomic system has other candidates that it can configure to take over when X fails. This declarative approach to autonomics is much clearer than an ECA-based specification[5]. The same avoidance of over-specification is also essential for the aspect composition described in the next section.

However, there are some usability issues with a constraint-based specification: It is often easier to choose a specific instance of a configuration parameter, than to define a constraint which permits all valid values. Conversely, there is a danger of writing specifications which are too “loose”, in the sense that they permit inappropriate values. Such errors might not be apparent at the time; they may only surface when some interacting constraint is changed and the system selects the inappropriate value as a solution – perhaps migrating the mail server onto a laptop because there was no explicit constraint preventing this!

There are also significant implementation difficulties: fully general constraint solving at the necessary scale, in a distributed environment is not currently practical. There are likely to be performance issues even when constraints are applied to a restricted set of attributes[6], and careful manual crafting is required to obtain useful results. Two other properties of any constraint solver are also important:

- Slight changes in the configuration constraints should not produce wildly different solutions – if the mail server is moved because of a failure, we do not want to unnecessarily move the web server as well. Most solvers are liable to generate unsuitable solutions like this if they treat the two configuration states as independent constraint problems.
- Most practical specifications include a lot of “soft” constraints which are “desirable” but not essential. For example, “I’d prefer

[5] “Event/Condition/Action”:
`http://en.wikipedia.org/wiki/Event_Condition_Action`

[6] See for example:
`http://alloy.mit.edu/papers/NetConfigAlloy.pdf`

service X to run on hardware of type Y *if possible*". Satisfying all of these constraints is rarely possible, but we want to satisfy as many as possible. Of course, in practice, some constraints are also more important than others, so there is complicating issue of constraint priorities.

These additional requirements on the constraint solver are not currently addressed by any practical system.

5. Aspects

In a modern datacentre, the systems are too complex for any one person to manage all of the various subsystems. Different specialists are responsible for different aspects of the system. Any one machine will have parts of its configuration derived from specifications that were written by different people – the network specialist, the application specialist, the security specialist, etc. Some aspects may even be defined by customers, or other external organisations – for example, the memory configuration of a virtual machine, or the access controls. Yet other aspects may be determined by automatic agents – for example, an autonomic system may decide to designate a particular machine as a DHCP server.

At first sight, this appears to be very similar to a large software system where different programmers are responsible for different modules and subsystems. But many (perhaps most) areas of responsibility cut across the normal structure of the configuration – security, for example involves aspects of most services, as well as the underlying access control system. Autonomic changes to an application service may involve reconfiguration of memory requirements, firewalls, and permitted users.

Very often, concerns from different aspects overlap in some subsystem – for example, multiple services may both have requirements on the available disk space, or the configuration of the firewall. Of course, we would like to be able to specify the requirements of these services completely independently and have the configuration tool take care of their composition. In the case of programming, aspect-oriented languages are designed to support exactly such "weaving" of independent concerns into a single program. This is a nice analogy, but the declarative nature of the configuration specifications means that the similarity does not run very deep. There does not appear to be a lot to learn directly from solutions to aspect-oriented programming.

However, a constraint-based system naturally provides a way to combine arbitrary concerns - if two independent people each specify constraints for their aspect of the system, then a simple conjunction can be used as the composite specification. An arbitrary constraint system suffers from all of the difficulties mentioned in the previous section. But in practice, it seems likely that a very restricted subset is sufficient for many typical aspect compositions.

One final consideration is security. Typical production operating systems are not designed to be managed by multiple users. If a certain user has control over the specification of some part of the core operating system, it is virtually impossible to be completely confident that the user cannot abuse the privilege and gain complete access to the machine. By composing the aspects on a trusted system, it is possible to control the individual parameters which any particular user is allowed to modify. This involves tracking the provenance of each parameter value, and performing the composition in a trusted environment. If the composition occurs in a distributed way, this can be very hard to guarantee. It is interesting to speculate if there could be any equivalent of "proof carrying code"' for configuration specifications.

6. Conclusions

It is difficult to draw any overarching conclusions from this comparison between programming and datacentre configuration. However, there are some interesting similarities as well some significant differences. It worth bearing both of these in mind when considering configuration solutions. Establishing a generic model at the level of a "machine code" for the datacentre would certainly allow different tools and components to interoperate in a way which is not currently possible.

Perhaps the most significant observation is the need to bring together researchers and practising administrators from the various disciplines to create new theories and tools which address the real-world problems.

Getting Operations Logic Right

Types, Service-Orientation, and Static Analysis

Karthikeyan Bhargavan Andrew D. Gordon
Microsoft Research

Abstract

The human operators of datacentres work from a manual, sometimes known as the run book, that lists how to perform operating procedures such as the provisioning, deployment, monitoring, and upgrading of servers. To improve failure and recovery rates, it is attractive to replace human intervention by software, known as operations logic, that automates such operating procedures. We advocate a declarative programming model for operations logic, and the use of static analysis to detect programming errors, such as the potential for misconfiguration.

1. Background: The Datacentre is the Computer

A datacentre is some housing (typically a room or a building) that physically contains a cluster of commodity servers, and provides them with power, cooling, and networking. Datacentres are the computers that run applications such as websites (search and mail in particular), financial services, computational science, and virtual worlds.

If the datacentre is the computer [Patterson, 2008], what is the program? We divide the code of a datacentre program into two parts: business logic and operations logic.

Datacentre Program = Business Logic + Operations Logic

The *business logic* is code that formalizes business tasks, that is, the core functionality of the application, such as how to manage an inbox, or how to sell a book. By analogy, the *operations logic* is code that formalizes operations tasks. The operations logic determines how to run and manage the application on a cluster of machines.

To illustrate the idea of business logic, consider a website implemented conventionally as three roles: web server, application server, and database server. A typical way to run such an application is to package up the code to run on each server role in a *disk image*, that is, a file containing the contents of the server's disk; the disk image holds all the code that runs on an individual server. The disk image for each role contains an operating system together with components specific to the role, such as web server, database server, and binaries and local configuration files specific to the application. Depending on load, there may be multiple servers running off the disk image for each role; these servers are known as *instances* of the role. According to our definition, the disk images for the three roles constitute the business logic of the application: this is the code that determines how to conduct the business implemented by the website.

Turning to operations, datacentres are run by human operators who work from a *run book*, a manual that lists how to perform various operating procedures. Here are some example tasks; for a comprehensive discussion of operations tasks see Anderson [2006].

- Provisioning: how many instances of each role where.
- Deployment: install disk images, start instances.
- Interconnection: connect the instances together.
- Monitoring: monitor instances for failure, overload.
- Evolution: respond to events, change deployment, versioning.

To improve failure and recovery rates, and to reduce need for actual operator involvement, many of these tasks would be better performed in software than manually. For example, events such as abrupt increases of load or machine failure should be handled swiftly and accurately. The empirical evidence [Oppenheimer et al., 2003, Nagaraja et al., 2004] is that operator errors in datacentre applications are common and costly. The scale and diversity of datacentre applications is another reason to automate operations tasks and hence reduce the need for human intervention.

This idea (sometimes known as *run book automation*) of encoding operations tasks as software, operations logic, is now a well-established trend. Various examples include include the declarative configurations of Anderson [1994], the self-healing systems of Burgess [1998], the automated deployment and configuration of SmartFrog [Goldsack et al., 2003], the manifesto for autonomic computing [Kephart and Chess, 2003], and the fault-tolerant programming model for large-scale clusters of MapReduce [Dean and Ghemawat, 2004]. More recently, Isard [2007] reports AutoPilot, a system to automate provisioning, deployment, and monitoring in large clusters.

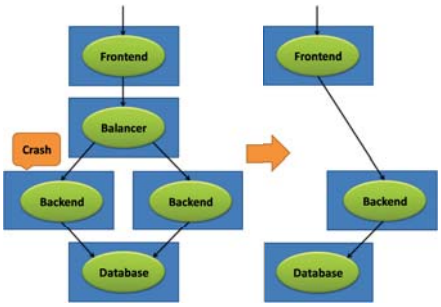
2. Problem: Getting Operations Logic Right

We claim that whereas development methods for business logic are well advanced (though not infallible), writing correct operations logic is more problematic (although much progress has been made). One reason is that the bulk of operations logic remains as low-level imperative code, written in scripting languages without the benefit of static typing.

For example, virtual machine monitors (VMMs) allow the provisioning, deployment, and monitoring of virtualized servers; the operations tasks associated with VMMs can be controlled entirely in software, but the programming interfaces exposed by today's VMMs are quite low-level. VMMs can easily be configured with scripts to be called in the event of machine failure or overload, but these scripts are hard to test, and are likely to contain bugs, particularly if they are called in response to relatively rare events.

To address the problem of getting operations logic right, our position is that operations logic would be better developed in high-level, statically typed programming languages, just as business logic is, than in low-level scripting languages, the common case at present.

Moreover, we advocate the static analysis of operations logic to help detect errors early. For example, techniques such as type-checking and model-checking are well suited to checking the behaviour of error recovery code.



(a) An Evolving Three-tier Application

```
1 let startupThreeTier db backend frontend =
2   let (d,dbServ) = start db () in
3   let (b1,bServ1) = start backend dbServ in
4   let (b2,bServ2) = start backend dbServ in
5   let (b,balance) = start balancer [bServ1;
6                                     bServ2] in
7   let (f,fServ) = start frontend balance in
8   let _ = register b1 (function Crash →
9       reconfigure f bServ2; stop b1; stop b) in
10  let _ = register b2 (function Crash →
11      reconfigure f bServ1; stop b2; stop b) in
12    fServ
```

(b) An Example Operations Logic Program

```
1 type (α,β) Service
2 val call: (α,β) Service → α→ β
3 type (γ) Server
4 type (γ,'services) Setup
5 val start: (γ,'services) Setup → γ→
6                         (γ) Server × 'services
7 val stop: (γ) Server → unit
8 val reconfigure: (γ) Server → γ→ unit
9 type event = Crash
10 val register: (γ) Server →
11                   (event → unit) → unit
12 val balancer: ((α,β) Service list,
13                (α,β) Service) Setup
```

(c) A Typed Service Management API

3. Declarative Approaches to Operations Logic

Baltic: Service Combinators for Virtual Machines The Baltic system [Bhargavan et al., 2008] addresses the problem raised in the previous section. The operations logic of a Baltic application is written in the functional language F# (a dialect of ML) [Syme et al., 2007]. Baltic applications are composed from virtualized servers and intermediaries, such as load balancers.

The programming model is based on the observation that a server is a software component that imports and exports a set of communication endpoints. For example, a web server may import an endpoint exported by a database server, and export a communication endpoint implementing a URL. The capability to create a server instance from a particular disk image is represented in Baltic as a typed F# function: the function receives the imported endpoints as a set of typed values, boots a machine instance, and returns the exported endpoints as a set of typed values. If we let a *service* be a set of endpoints, we may say that the Baltic model is *service-oriented*. The function to boot a server transforms the service it imports to the service it exports. Operations logic in Baltic manipulates a typed call graph; nodes are role instances, labelled with exported services, while edges are potential calls from one role to an endpoint on another.

This high-level view of instance creation contrasts with the low-level view in conventional programming models for operations logic. Typical programming models for VMMs (for example, Virtual Server [Armstrong, 2007]) are lower-level in that they are *device-oriented* rather than being service-oriented: programs manipulate a graph whose nodes are virtual devices (disks, processors, network adapters) and whose edges are virtual wiring. For example, the Virtual Server function to create a role instance simply boots a machine from a disk image; no matter their imports and exports, all disk images are represented as untyped files. In Baltic, by contrast, there is a distinctly typed function to create each role.

Our initial implementation [Bhargavan et al., 2008] establishes the feasibility of this new approach. We have operations logic in F# that manages pre-existing disk images from a sample multi-tier web application running on a single VMM. There is a semantics based on a typed concurrent λ-calculus with partitions, and an implementation using Virtual Server. Type-checking the operations logic statically detects some errors, such as endpoint interconnection bugs. We can also symbolically simulate the operations logic, without the business logic, to find other errors.

Still, there is much to be done to validate this approach. One direction is to demonstrate its viability at the scale of a datacentre rather than a single VMM. Another is to perform more sophisticated static analysis than type-checking.

Example We illustrate the style of the Baltic approach through the example depicted at the top of this page. For the sake of brevity, we use simplified pseudo-code, rather than the actual code running in Baltic. Our aim is to write a program to set up and manage a simple three-tier application as shown in Figure (a); it consists of a frontend server that forwards incoming requests to a backend server that in turn may send a message to a database. The backend server can be a bottleneck; hence, it is replicated and a load balancer divides requests between two backend servers to avoid overloading either one. Once the application is up and running, if one of the backend servers were to crash, we would like to reconfigure the frontend to forward requests directly to the other backend, bypassing the redundant load balancer.

The business logic consists of three packages, db, backend, and frontend, that consist of software and configuration data that fully describe each of the three server roles.

The operations logic is written as the F# program in Figure (b) using functions in the service management API of Figure (c). The function startupThreeTier takes db, backend, and frontend as its arguments (line 1) and then starts up instances of each server. Line 2 calls start to provision a server d to act as a database, install the package db on this server, start it up, and return the database service address dbServ. (Here, db requires no configuration data hence the second argument to start is unit.) Line 3 provisions a backend server b1, installs the package backend on it, configures the server with the address of the database (dbServ), starts up b1 and returns the address of its backend service bServ1. Similarly, line 4 starts up a second backend server b2 and returns its service address bServ2. Lines 5–6 provision a load balancing server b that has the balancing software balancer installed on it. The balancer is configured with a list of available backend services (here just bServ1 and bServ2) and returns a service, balance, that divides requests equally amongst the available backends. Line 7 provisions a frontend server f, installs the package frontend on it, configures it with the balancer address balance, starts it up, and returns the service fServ that can be accessed by the external world. At this stage, the three-tier application is up and running in its initial configuration. The next four lines register event handlers that prescribe how the application should be reconfigured in the event of a server crash. Lines 8–9 register, via the function register, an event handler that constantly monitors the backend server b1; when it detects a Crash event, it reconfigures the frontend f to point to the other backend service bServ2 (bypassing the balancer); it then stops the crashed server b1 and the redundant balancer b. Lines 10–11 register a similar event handler for b2. Finally, the function startupThreeTier returns the external service fServ (line 12). Hence, this program performs all the standard management tasks of operations logic: it provisions and deploys servers for each role, interconnects them using application-specific configuration, monitors them to detect events, and evolves the application by reconfiguring the frontend.

A Typed Service Management API The general forms of the functions start, stop, register, reconfigure, and balance used in our example are shown in the typed F# interface of Figure (c).

The type (α,β) Service represents the address of a service that takes requests of type α and returns responses of type β. The function call makes a remote procedure call to such a service. For instance, suppose the frontend service in our example has type FrontendService = (int,bool) Service; then one may call it with an integer request to get a boolean response.

The type (γ) Server represents the name of a server providing services that have configuration data of type γ. The type (γ,'services) Setup represents a package containing software for services of type 'services that have configuration data of type γ. For instance, the packages db, backend, and frontend are all of this type; frontend has type (BackendService,FrontendService) Setup, meaning that it can install a service of type FrontendService that needs to be configured with the address of a service of type BackendService. The function start takes a package of type (γ,'services) Setup, configuration data of type γ and returns the name of a new server of type (γ) Server that provides services of type 'services. The function stop stops a server; reconfigure installs new configuration data.

The type event records events relevant to operations logic; for simplicity, we only record server Crash events. The function register takes a server and an event handler of type event → unit and starts a process that monitors the server and invokes the handler when it detects an event.

Service management APIs typically include preprogrammed packages for common server tasks. Here, the package balancer takes a list of services, each of type (α,β) Service, as its configuration data and installs a load balancing service of the same type.

The API of Figure (c) may be implemented and interpreted in several ways. In the Baltic system, packages are implemented as disk images and servers are virtual machines within a VMM. In Autopilot, packages are self-installing software manifests and servers are generic computers running within a datacentre. For each interpretation, we can define a formal semantics for programs written against the API, by defining the functions of the API in terms of expressions in a partitioned lambda-calculus [Bhargavan et al., 2008]. Such a semantics can be used to reason about operations logic programs, such as the one in Figure (b), and to symbolically simulate and test such programs before deployment.

Our service management API is typed, and so are our programs. Type-checking catches common interconnection errors statically. For instance, in line 9 of Figure (b), if we mistyped bServ2 as dbServ, hence reconfiguring the frontend with a database service in place of a backend service, the type-checker would catch this error. In an untyped setting, such a misconfiguration would be detected only after deployment, when a frontend becomes unresponsive or returns fault messages after the application evolves in response to a (possibly rare) server crash. Conversely, some errors are difficult for type-checking to detect. For instance, the event handlers in Figure (b) do not attempt to restart failed servers or to notify human operators about degraded services; so the operations logic survives one failure but not two. Type-checking does not find such behavioural problems, but other static analyses such as model-checking should be applicable.

To summarize, type-checking operations logic is a good start. Still, there remains plenty of scope for applying stronger static analyses to provide better guarantees about the programs running on datacentres.

Acknowledgments

We thank Iman Narasamdya for his work on the Baltic implementation. Galen Hunt drew our attention to the term “operations logic”. Ioannis Baltopoulos commented on a draft of this paper.

References

P. Anderson. *System Configuration*, volume 14 of *Short Topics in System Administration*. SAGE, 2006.

P. Anderson. Towards a high-level machine configuration system. In *Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, pages 19–26, Berkeley, CA, 1994.

B. Armstrong. *Professional Microsoft Virtual Server 2005*. Wiley, 2007.

K. Bhargavan, A. D. Gordon, and I. Narasamdya. Service combinators for farming virtual machines. In *COORDINATION'08*, 2008. To appear. Extended version available as Microsoft Research Technical Report MSR–TR–2007–165.

M. Burgess. Computer immunology. In *LISA '98: Proceedings of the 12th USENIX conference on System administration*, pages 283–298. USENIX Association, 1998.

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and automatic ignition of distributed applications, 2003. Presented at 2003 HP Openview University Association conference. Available at `http://www.hpl.hp.com/research/smartfrog/`.

M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, 2007.

J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.

K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, pages 61–76, 2004.

D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *USITS'03: 4th Usenix Symposium on Internet Technologies and Systems*, 2003.

D. A. Patterson. Technical perspective: the data center is the computer. *Commun. ACM*, 51(1), 2008.

D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

Foundational Aspects of Contract Compliance and Choreography Conformance

Mario Bravetti and Gianluigi Zavattaro
Department of Computer Science, University of Bologna, Italy
{bravetti,zavattar}@cs.unibo.it

Abstract

In the context of Service Oriented Computing, contracts are descriptions of the externally observable behaviour of services. Given a group of collaborating services, their contracts can be used to verify whether their composition is sound, i.e., the services are compliant. In our work, we relate the theory of contracts with the notion of choreography conformance, used to check whether an aggregation of services correctly behaves according to a high level specification of their possible conversations. The main result of our work is the definition of an effective procedure that can be used to verify whether a service with a given contract can correctly play a specific role within a choreography. This procedure is achieved via composition of choreography projection and contract refinement. Our work comprises the analysis of different options for the contract language adopted, where either outputs and inputs are synchronized as in standard process algebra or a destination contract is specified in outputs, and for the notion of compliance adopted, where we consider either a standard notion of compliance which just ensures deadlock and livelock freedom or a stronger one where outputs cannot wait for the availability of a synchronizing input.

1. Introduction

Service Oriented Computing (SOC) is a novel paradigm for distributed computing based on services intended as autonomous and heterogeneous components that can be published and discovered via standard interface languages and publish/discovery protocols. One of the peculiarities of Service Oriented Computing, distinguishing it from other distributed computing paradigms (such as component based software engineering), is that it is centered around the so-called *message oriented architecture*. This means that, given a set of collaborating services, the current state of their interaction is stored inside the exchanged messages and not only within the services. From a practical viewpoint, this means that it is necessary to include, in the exchanged messages, the so-called correlation information that permits to a service to associate a received message to the correct session of interaction (in fact, the same service could be contemporaneously involved in different sessions at the same time).

Web Services is the most prominent service oriented technology: Web Services publish their interface expressed in WSDL, they are discovered through the UDDI protocol, and they are invoked using SOAP.

Two main approaches for the composition of services are currently under investigation and development inside the SOC research community: service *orchestration* and service *choreography*. According to the first approach, the activities of the composed services are coordinated by a specific component, called the orchestrator, that is responsible for invoking the composed services and collect their responses. Several languages have been already proposed for programming orchestrators such as XLANG (Tha01), WSFL (Ley01) and WS-BPEL (OAS).

Choreography languages are attracting a lot of attention within W3C, where the most credited choreography language WS-CDL (W3C) is currently under development. Choreographies represent a "more democratic" alternative approach for service composition with respect to orchestrations. Indeed, orchestrations require the implementation of central points of coordination; on the contrary, choreography languages support a high level description of peer-to-peer interactions among services that directly communicate without the mediation of any orchestrator. Unfortunately, choreography languages are not yet popular due to the difficulties encountered while translating the high level description of the composed services into an actual system obtained as combination of autonomous, loosely coupled and heterogeneous components.

As an example of service composition, let us consider a travel agency service that can be invoked by a client in order to reserve both an airplane seat and a hotel room. In order to satisfy the client's request, the travel agency contacts two separate services, one for the airplane reservation and one for the hotel reservation. A choreographic specification of this service composition describes the possible flows of invocations exchanged among the four roles (the client, the travel agency, the airplane reservation service, and the hotel reservation service). A formal specification of a choreography of this kind can be developed as follows.

(Reservation via Travel Agency) Let us consider the following choreography composed of four roles: *Client*, *TravelAgency*, *AirCompany* and *Hotel*

$$
\begin{array}{l}
Reservation_{Client \rightarrow TravelAgency};\\
(\ (Reserve_{TravelAgency \rightarrow AirCompany};\\
\qquad ConfirmFlight_{AirCompany \rightarrow TravelAgency})\ |\\
\ (Reserve_{TravelAgency \rightarrow Hotel};\\
\qquad ConfirmRoom_{Hotel \rightarrow TravelAgency})\ );\\
Confirmation_{TravelAgency \rightarrow Client} +\\
\qquad Cancellation_{TravelAgency \rightarrow Client}
\end{array}
$$

According to this choreography, the *Client* initially sends a reservation request to a travel agency, that subsequently contacts in parallel an airplane company *AirCompany* and a room reservation service *Hotel* in order to reserve both the travel and the staying of the client. Then, the travel agency either confirms or cancels the reservation request of the client.

The problem that we consider in our work (BZ07a; BZ07b; BZ07c) can be summarized as follows: given a choreography, we want to define an automatic procedure that can be used to check whether a service correctly plays one of the roles described by the choreography. For instance, given a choreographic specification of the above travel agency example, and an actual travel agency service, we want to check whether the actual service behaves correctly according to the choreographic specification. The solution that we

propose to this problem assumes that the services expose in their interface an abstract description of their behaviour. In the service oriented computing literature, this kind of information is referred to as the *service contract* (CL06). More precisely, the service contract describes the sequence of input/output operations that the service intends to execute within a session of interaction with other services. In particular, we propose to combine choreography projection with service contract refinement. The former permits to extract the expected behaviour of one role and synthesize a corresponding service contract. The latter permits to characterize an entire class of contracts (that refine the contract obtained by projection), for which it is guaranteed that the corresponding services correctly play the considered role in the choreography.

2. Contract Refinement

In (BZ07a; BZ07b) we formalize service contracts via a process calculus. The definition of contract refinement is based on the notion of contract compliance: n services/contracts are compliant if their composition is guaranteed to successfully complete without deadlocks or livelocks. After having formalized compliance, we are able to formalize the property that refinement of each service/contract should satisfy. An important property of our theory is that contract refinement is defined locally: a refinement is a subcontract pre-order if it preserves service compliance, namely, given n compliant services, and independently substituting each of them with one of its refinement, the achieved n services are still compliant. Then we define the *subcontract relation* as the union of all subcontract pre-orders. One of our main results is that for the calculus that we propose, the subcontract relation achieved according to this approach is actually the largest subcontract pre-order. In fact, in other theories of contracts recently proposed in the literature (see the section about related work), this property does not hold.

This result has not only a theoretical relevance, but we also foresee very important practical applications in the context of *service discovery* and *service update*. As far as service discovery is concerned, we can consider a service system defined in terms of the contracts are required for each of the service components. The actual services to be combined could be retrieved querying service registries collecting those services that either expose exactly the required contract, or one of its refinement. Due to the existence of a global independent refinement notion, we can retrieve services independently one from the other (possibly in parallel) without breaking compliance. Independent retrieval is not supported by other theories for service contracts such as, for instance, the standard (i.e. not strong) subcontract relation proposed in (LP07). As far as *service update* is concerned, we can use our approach to ensure backward compatibility. Consider, e.g., a service that should be updated in order to provide new functionalities; if the new version is a refinement of the previous service, our theory ensures that the new service is a correct substitute for the previous one in any previously defined service compositions.

The calculus for contracts that we propose is similar to traditional process calculi distinguishing between deadlock and successful termination. This distinction is necessary in order to model the fact that a service could internally deadlock due to its internal parallelism. Another peculiarity of the calculus is that the decision to execute output actions may not depend on the other services in the system. This reflects the fact that in asynchronous distributed systems, such as the Internet currently used as transport layer for Web Services, there is an implicit asymmetry between input and output actions. This asymmetry derives from the fact that an output operation consists of the emission of a message, while an input operation coincides with the reception of a message. The decision to send a message in an asynchronous system is taken locally, and cannot be forbidden by the remote expected receiver.

In more technical terms, we do not allow for the specification of processes with external choices guarded on output actions such as $\overline{a} + \overline{b}$, or mixed choices $a + \overline{b}$ where a is an input action and $\overline{b}$ is an output. In our calculus, the decision to execute an output is guaranteed to be taken locally imposing that all output actions $\overline{b}$ are preceded by an internal τ action; in this way a service first decides to execute the output, and only subsequently the message is actually emitted. We show that this constraint is theoretically necessary for achieving the existence of a largest subcontract pre-order (hence for a having a global notion of contract refinement).

Another important technical achievement of our work is a characterization of the subcontract relation in a testing-like scenario (DH84): we can prove that a contract C' is a subcontract of C if, after some appropriate transformations applied to both C' and C, the former is guaranteed to satisfy at least all the tests satisfied by the latter. In particular, we show how to use the theory of *should-testing* (RV05) to prove that one contract is a subcontract of another one. This characterization permits to have an effective procedure to prove whether a service is a refinement of another one. In fact, the definition of refinement is not directly applicable because it contains a universal quantification on all possible contexts.

Another important consequence of this characterization, is a precise localization of our refinement with respect to traditional refinements such failure refinement, or simulation (i.e. half-bisimulation): the refinement that we achieve as the largest one preserving compliance is coarser than both failure refinement and simulation. Notice that in our approach we do not start by arbitrarily assuming a particular notion of pre-order, e.g. one of the traditional refinements above, to be used for contract refinement, we instead proceed the other way around: we start from the property that we want our notion to satisfy (independent preservation of contract compliance) and, out of that, we induce a refinement pre-order and we classify it with respect to existing ones.

In (BZ07b) we associate to contracts also an additional information (technically called "location") that is used to specify the destination of outputs (so that an output can be received only by the contract specified). The presence of locations permits to prove new interesting results that do not hold when non-location based communication of standard process algebras is used (as in (BZ07a)): the knowledge about output actions of other initial contracts (that is expressed as a parameter of the refinement pre-order in (BZ07a)) is no longer necessary for the subcontract refinement to allow refined contracts to include new input actions. This extension also permits to associate contracts (and the corresponding services) with roles in a choreography simply by mapping roles into locations.

3. Coreography Conformance

In (BZ07b) we also address the problem of the independent retrieval of services implementing, by composition, a given choreography specification in the context of service oriented computing. In particular, we formalize service choreographies via process calculus and we define coreography conformance similarly as for contract refinement. Given a coreography H and independently considering a contract which is conformant to H for each role in the choreography the achieved services are a correct implementation of the choreography: i.e. when they are composed they produce a subset of the traces of the choreography H. This property permits to retrieve the actual services to be composed to implement the choreography independently one from the other (e.g. contemporaneously querying different service registries).

Here we have a negative result: we show that a largest conformance relation does not exist.

We therefore propose a particular conformance relation which combines a notion of choreography projection, which automati-

cally extracts an "ideal" contract for each role in the choreography, in combination with service contract refinement. We call such a conformance relation *consonance*. The consonance relation is parameterized on a given choreography H and relates service contracts to roles: if a contract C is consonant to a role r, then the services exposing contract C (or one of its refinements) correctly play role r in the considered choreography H.

4. Strong Compliance

In ordinary notions of compliance, e.g., in (CCL$^+$06; LP07; CGP08) and in our approaches of (BZ07a; BZ07b) services are compliant when they can be correctly combined in such a way that all the service invocations of one service in the system are guaranteed to be eventually served by another service in the system.

For instance, the following three services $S1$, $S2$, and $S3$ are compliant:

$$\begin{array}{ll} S_1: & invoke(a@S_2); receive(b) \\ S_2: & receive(a); invoke(c@S_3) \\ S_3: & receive(c); invoke(b@S_1) \end{array}$$

We use $invoke(a@S)$ to denote the invocation of operation a on service S, while $receive(a)$ is the receival of an invocation on operation a.

According to this notion of compliance, also the following is an example of compliant services:

$$\begin{array}{ll} S_1: & invoke(a@S_2); invoke(b@S_3) \\ S_2: & receive(a); invoke(c@S_3) \\ S_3: & receive(c); receive(b) \end{array}$$

It is worth observing that the second invocation of S_1 (i.e. $invoke(b@S3)$) cannot be served immediately, because it is necessary to wait for the interaction between S_2 and S_3.

We can generalize this example to n services as follows:

$$\begin{array}{ll} S_1: & invoke(a_1@S_2); invoke(a_n@S_n) \\ S_2: & receive(a_1); invoke(a_2@S_3) \\ S_3: & receive(a_2); invoke(a_3@S_4) \\ \cdots & \\ S_i: & receive(a_{i-1}); invoke(a_i@S_{i+1}) \\ \cdots & \\ S_n: & receive(a_{n-1}); receive(a_n) \end{array}$$

According to the above traditional notion of compliance also these services are compliant, even if the second invocation of S_1 must wait for $n-1$ subsequent interactions among $n-1$ different services. This arbitrary delay could be problematic if we consider protocols for service invocations delivery that, as usually is the case, exploit time-outs in order to avoid clients to wait indefinitely for the delivery of an invocation. In this case, the second invocation of S_1 will be timed-out generating an undelivered invocation exception. These exceptions are undesired events and they should not be generated while executing actually compliant services.

In (BZ07c) we address this lack of the traditional notions of compliance, proposing the new *strong service compliance*. Intuitively, we assume that when a service is ready to execute an invocation on another service in the system, the target service is ready to serve the invocation. According to this new assumptions, for instance, the services in the last two examples above are no longer compliant because the second invocation of the first service S_1 cannot be served as soon as it can be delivered to the target service.

Besides proposing a formalization for strong service compliance, we present a complete theory for contracts which is consistent with strong service compliance. In particular the adoption of strong compliance leads to the existence of a largest subcontract pre-order for a standard process algebra with no need for disallowing external choice among outputs (intuitively because the asymmetry between inputs and outputs is already captured in the notion of compliance).

Even in the case of strong compliance we are able to characterize of our notion of refinement by resorting to the theory of testing (DH84), in particular to the should testing pre-order investigated in (RV05). In the case of strong compliance the characterization is, however, much more involved due to the complexity of encoding the notion of failure related to the constraint about input availability into the standard notion of success in testing theory.

5. Related Work

Choreography languages have been already investigated in a process algebraic setting by Carbone et al. (CHY07) and by Busi et al. (BGG$^+$05; BGG$^+$06).

The paper (CHY07) is the first one, to the best out knowledge, in which the problem of ill-formed choreographies is considered: a choreography is ill-formed when it is not possible to achieve by projection a correct implementation that preserves the message exchanges specified by the choreography. The solution to this problem presented in (CHY07) is given by three basic principles that, when satisfied by a choreography, ensure to achieve a corresponding correct projection. On the one hand, the calculi proposed in (CHY07) are more expressive than the calculi we define in our work because they comprise name passing and an explicit notion of session. On the other hand, the basic principles imposed in (CHY07) give rise to a more restrictive notion of well formed choreography with respect to the one proposed in our work (see (BZ07b)). In (BGG$^+$05; BGG$^+$06) a more general notion of conformance between a choreography and a corresponding implementation as a service system is defined. According to this more general notion of conformance the implementation does not necessarily follow from projection, but additional services (not included at the choreography level) can be added in order to synchronize the correct scheduling of the the message flow.

The notion of contract refinement that we propose is achieved resorting to the theory of testing. There are some important differences between our form of testing and the traditional one proposed by De Nicola-Hennessy (DH84). The main difference is that, besides requiring the success of the test, we impose also that the tested process should successfully complete its execution. Moreover, in the case of strong compliance, all output actions of both the tester and the tested process should be immediately receivable. Another difference is in the treatment of divergence: we do not follow the traditional catastrophic approach, but the fair approach introduced by the theory of should-testing by Rensink-Vogler (RV05). In fact, we do not impose that all computations must succeed, but that all computations can always be extended in order to reach success.

We conclude our analysis of related work considering the theory of contracts by Fournet et al. (FHR$^+$04) and the one proposed by Carpineti et al. (CCL$^+$06) and extended in (LP07) and (CGP08).

In (FHR$^+$04) contracts are CCS-like processes; a generic process P is defined as compliant to a contract C if, for every tuple of names $\tilde{a}$ and process Q, whenever $(\nu\tilde{a})(C|Q)$ is stuck-free then also $(\nu\tilde{a})(P|Q)$ is. Our notion of contract refinement differs from stuck-free conformance mainly because we consider a different notion of stuckness. In (FHR$^+$04) a process state is stuck, on a tuple of channel names $\tilde{a}$, if it has no internal moves but it can execute at least one action on one of the channels in $\tilde{a}$. In our approach, an end-state different from successful termination is stuck (independently of any tuple $\tilde{a}$). Thus, we distinguish between internal deadlock and successful completion while this is not the case in (FHR$^+$04). Another difference follows from the exploitation of the restriction $(\nu\tilde{a})$; this is used in (FHR$^+$04) to explicitly indicate the local channels of communication used between the contract C and the process Q. In our context we can make a stronger *closed-*

world assumption (corresponding to a restriction on all channel names) because service contracts do not describe the entire behaviour of a service, but the flow of execution of its operations inside one session of communication.

The closed-world assumption is considered also by Carpinets et al. in (CCL$^+$06) where, as in our case, a service oriented scenario is considered. In particular, in (CCL$^+$06) a theory of contracts is defined for investigating the compatibility between one client and one service. Our work considers multi-party composition where several services are composed in a peer-to-peer manner. Moreover, we impose service substitutability as a mandatory property for our notion of refinement; this does not hold in (CCL$^+$06) where it is not in general possible to substitute a service exposing one contract with another one exposing a subcontract. Another relevant difference with respect to our work not adopting a strong notion of compliance is that the contracts in (CCL$^+$06) comprise also choices guarded by both input and output actions.

The work of Carpineti et al. has been extended in two ways, in (LP07) by explicitly associating to a contract the considered input/output alphabet, in (CGP08) by associating to services a dynamic filter which eliminates from the service behaviour those interactions that are not admitted by the considered contract. The explicit information about the input/output alphabet used in (LP07) allows the corresponding theory of contracts to be applied also in multi-party compositions, but the independent refinement that we advocate can be only partially achieved. In fact, in general, a subcontract might consider a larger input/output alphabet with respect to an initial contract, but the input/output names added by each subcontract in the system must have empty intersection. In other terms, the selection of one subcontract with a larger input/output alphabet can influence the choice of the possible other subcontracts in the system. The dynamic filters of (CGP08) allow for independent refinement, at the price of synthesizing a specific filter for each service that eliminates the additional behaviours introduced by services exposing a subcontract. Even if very interesting from a theoretical point of view, the practical application of filters is not yet clear. In fact, in general it is not possible to assume the possibility to associate a filter to a remote service. This problem can be solved in client-service systems, assuming that a co-filter is applied to the local client, but it is not clear how to solve it in multi-party systems composed of services running on different hosts.

References

[BZ07a] Mario Bravetti and Gianluigi Zavattaro. Contract based Multi-party Service Composition. In *FSEN'07*, volume 4767 of LNCS, pages 207–222, 2007.

[BZ07b] Mario Bravetti and Gianluigi Zavattaro. Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In *SC'07*, volume 4829 of LNCS, pages 34–50, 2007.

[BZ07c] Mario Bravetti and Gianluigi Zavattaro. A Theory for Strong Service Compliance. In *Coordination'07*, volume 4467 of LNCS, pages 96–112, 2007.

[BGG$^+$05] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration: A synergic approach for system design. In *ICSOC'05*, volume 3826 of LNCS, pages 228–240, 2005.

[BGG$^+$06] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration conformance for system design. In *Coordination'06*, volume 4038 of LNCS, pages 63–81, 2006.

[CHY07] Marco Carbone, Kohei Honda, and Nabuko Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume to appear of LNCS, 2007.

[CGP08] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A Theory of Contracts for Web Services. In *POPL'08*, to appear, 2008.

[CCL$^+$06] Samuele Carpineti, Giuseppe Castagna, Cosimo Laneve, and Luca Padovani. A Formal Account of Contracts for Web Services. In *WS-FM'06*, volume 4184 of LNCS, pages 148-162, 2006.

[CL06] Samuele Carpineti and Cosimo Laneve A Basic Contract Language for Web Services. In *ESOP'06*, volume 3924 of LNCS, pages 197–213, 2006.

[DH84] Rocco De Nicola and Matthew Hennessy, Testing Equivalences for Processes. *Theoretical Computer Science*, volume 34: 83–133, 1984.

[FHR$^+$04] Cédric Fournet, C. A. R. Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-Free Conformance. In *CAV'04*, volume 3114 of LNCS, pages 242–254, 2004.

[LP07] Cosimo Laneve and Luca Padovani. The must preorder revisited - An algebraic theory for web services contracts. In *Concur'07*, volume 4703 of LNCS, pages 212-225, 2007.

[Ley01] F. Leymann. Web Services Flow Language (wsfl 1.0). Technical report, IBM Software Group, 2001.

[RV05] Arend Rensink and Walter Vogler. Fair testing. *CTIT Technical Report TR-CTIT-05-64*, Department of Computer Science, University of Twente, December 2005.

[OAS] OASIS. *Web Services Business Process Execution Language Version 2.0*.

[Tha01] S. Thatte. XLANG: Web services for business process design. Microsoft Corporation, 2001.

[W3C] W3C. *Web Services Choreography Description Language*. `http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217`.

Architectural Design Rewriting as an Architecture Description Language

Roberto Bruni Alberto Lluch Lafuente
Ugo Montanari
Department of Computer Science, University of Pisa
{bruni,lafuente,ugo}@di.unipi.it

Emilio Tuosto
Department of Computer Science, University of Leicester
et52@mcs.le.ac.uk

Abstract

Architectural Design Rewriting (ADR) is a declarative rule-based approach for the design of dynamic software architectures. The key features that make ADR a suitable and expressive framework are the algebraic presentation of graph-based structures and the use of conditional rewrite rules. These features enable the modelling of, e.g. hierarchical design, inductively defined reconfigurations and ordinary computation. Here, we promote ADR as an Architectural Description Language.

Categories and Subject Descriptors D.2 [*Software Engineering*]: Design Tools and Techniques, Software Architectures; G.2.2 [*Discrete Mathematics*]: Graph Theory—Graphs

Keywords Dynamic Software Architectures, Architectural Styles, Graphs, Term Rewriting

1. Introduction

Architectural Design Rewriting (ADR) (Bruni et al. 2008b) is a proposal for the design of reconfigurable software systems, conceived in the spirit of conciliating software architectures and process calculi by means of graphical methods. ADR offers a formal setting where design development, run-time execution and reconfiguration are defined on the same foot.

The key features of ADR are: (i) hierarchical and graphical design; (ii) rule-based approach; (iii) algebraic presentation; and (iv) inductively-defined reconfigurations. Architectures are modelled by *typed designs*: a kind of interfaced graphs whose inner items represent the architectural units and their interconnections and whose interface expresses the overall type and its connection capabilites. Architectures are designed hierarchically by a set of composition operators called *design productions* which enable: (i) top-down refinement, like replacing an abstract components with a possibly partial realisation, (ii) bottom-up typing, like deducting the type of and actual architecture, and (iii) well-formed composition, like composing some well-typed actual architectures together so to guarantee that the result is still well-typed.

Domains of *valid* architectures, i.e. those compliant to styles, patterns or constraints, are defined in a declarative way by means of design productions. Such productions have both a functional reading as valid architectural compositions and a grammar reading as providing an inductive definition of valid architectures.

In the functional reading, the set of productions defines an algebra of design terms, each encoding the structure of the architecture and providing a proof of style conformance. The interpretation of a design term is a design, i.e. the actual architecture.

Reconfiguration and behaviour are given as term rewrite rules acting over design terms rather than over designs. This has many advantages: (i) terms compactly and conveniently encode the hierarchical structure of the architecture; (ii) ordinary term rewriting techniques allow to specify complex reconfigurations and computations that can exploit the hierarchical structure encoded in terms; (iii) preserving properties such as style-conformance during reconfiguration can be ensured by construction.

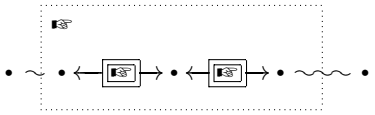


Figure 1. Design of a pipe (type ☞). The internal structure is formed by two partially specified pipes connected in sequence by binding the respective left and right ports (arrows) directly (connectors are neglected for simplicity) at the same node (bullet). Only the two ports at the extremes of the pipe are exposed (waved lines) at the interface (dotted box). The figure can also be interpreted as a design production composing two pipes in a pipe or as a refinement of a pipe as two pipes.

2. ADR as ADL

ADR was not conceived as an Architecture Description Language (ADL) but as a suitable model for style-consistent design and reconfiguration of software architectures. As a matter of fact, ADR turned out to be a general mechanism, suitable for heterogeneous models such as network topologies, architectural styles and modelling languages (Bruni et al. 2007). Despite of its generality, we think that the features of ADR are particularly tailored to ADL problematics. Indeed, we believe that ADR can be seen as an ADL itself, as a formal model of existing ADLs, possibly equipping them with extended features, like conditional reconfigurations. We promote this vision of ADR by discussing the issues that the software architectures community retains particularly relevant.

2.1 Components and Connectors

The main actors of ADR are design, which uniformly model both components and connectors. Designs are technically defined by hierarchical hyperedges whose internal structure can range from an empty graph to an arbitrarily complex graph (see Figure 1).

Interface. The interface of a design is given by the tentacles of the outerface hyperedge. Each tentacle represents a port (resp. role) of the corresponding component (resp. connector). Attaching a port to a role is done by connecting the respective tentacles to the same node. As usual in many ADLs, ports and roles are typed.

Types. Most of the ingredients of ADR such as design terms, designs, hyperedges and nodes are typed. This enables, e.g. the

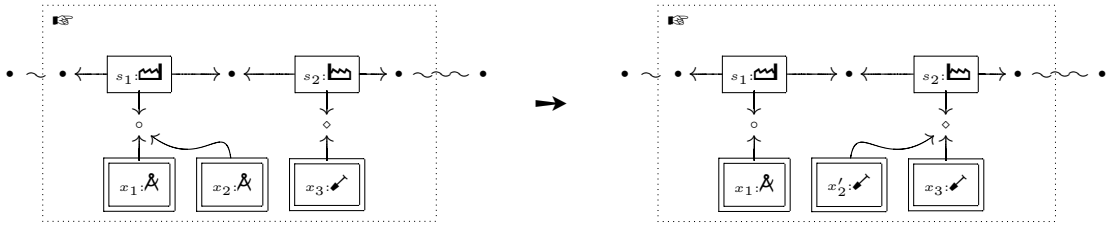


Figure 2. Reconfiguration rule for migrating tasks in a pipe. The design on the left is a pipe (type ☞) that consists of two concatenated servers of different types (and) to which a collections of tasks of appropiate type (and) are attached. The reconfiguration requires x_2 to evolve into x'_2 in order to migrate from s_1 to s_2. Types are changed consistently and the design obtained is still of type ☞.

construction of style-consistent architectures and the distinction between design classes (types) and their instances (designs).

Semantics. ADR is a formal model with a well-defined semantics. As a matter of fact, ADR builds on well-founded techniques such as algebraic approaches to graph transformation and rewrite rules in Plotkin's structural operational semantics style and Meseguer's conditional term rewriting.

Constraints. Architectural constraints are typically given in some logic-based language. Instead, ADR promotes to encode constraints as types when possible: the set of all architectures satisfying some constraint should correspond to the set of all terms of a certain type, guaranteeing constraint consistency and its preservation by construction.

Evolution. Architectural evolution can be given in terms of software modes. In each mode different behaviour, constraints or reconfigurations might apply. Modes and mode changes can be suitably modelled in ADR with types and rewrite rules, respectively.

Non-functional properties. The current version of ADR does not consider non-functional properties. However, we intend to model QoS aspects by means of constraints systems associated to designs.

2.2 Architectural Configurations

Architectural configurations are modelled by designs, allowing the uniform treatment of components, connectors and configurations. Hence, we restrict the discussion to configuration particularities.

Compositionality. Composite configurations, components and connectors are constructed hierarchically via design productions. Non admissible configurations, like connecting a client with another client instead of a server, can be ruled out by construction.

Refinement. Refinement is straightforwardly supported by a particular reading of design productions.

Traceability. ADR supports traceability aspects by means of equipping architectures (designs) with a witness of their construction (design terms). ADR promotes to carry such information over run-time to support efficient and autonomous reconfiguration.

Scalability. Adding entities to an architecture can be achieved in ADR via suitable refinements or rewrite rules. The features of ADR favour a modular approach, so that, e.g. the addition of new components can be localised in the desired sub-architecture, without affecting the rest of the system.

Dynamism. Complex behaviours and reconfigurations are expressed in ADR by flexible rewrite rules (see Figure 2). ADR does not marry any particular kind of dynamism (such as programmed or repairing); it is general enough to cover most of them.

Understandability. Architectural configurations in ADR improve understandability due to their hierarchical composition, which allows to browse complex structures inductively and to focus on the most convenient level of detail. A further support to understandability is the immediate visual representation as (typed) graphs.

2.3 Tool Support

ADR tool support is under development (Bruni et al. 2008a). At its current status prototype specifications can be simulated and some structural and behavioural properties can be analysed with verification techniques working at different levels of abstraction.

3. Conclusion

ADR suitably captures most of the features that an ADL should consider. The main advantages of ADR over similar approaches is given by the use of architectural construction information in form of design terms which enables hierarchical reconfigurations, which, most notably, are style-consistent by construction and easy to understand. Over most ADLs, ADR offers a unifying model to represent architectural design, reconfiguration, and ordinary behaviour. We believe that ADR can help in understanding and solving ADL problematics and that it can serve as the basis to formalise or extend them. For instance, the term-based feature of ADR can be exploited by ADLs that do not consider dynamism by defining architectural specifications as terms and modelling dynamism as suitable term rewrite rules. The tool-support for ADR is in a primitive but promising stage. Further information on ADR can be found at `http://www.albertolluch.com/adr.html`.

Acknowledgments

ADR is supported by the EU FETPI-GC2 project IST-2005-016004 SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*) and by the Italian FIRB project TOCAI (*Knowledge Oriented Technologies for Enterprise Integration in Internet*).

References

Roberto Bruni, Alberto Lluch Lafuente, Ugo Montanari, and Emilio Tuosto. Service Oriented Architectural Design. In *Proceedings of the 3rd International Symposium on Trustworthy Global Computing (TGC'07)*, volume 4912 of *Lecture Notes in Computer Science*, pages 186–203. Springer, 2007.

Roberto Bruni, Alberto Lluch Lafuente, and Ugo Montanari. Hierarchical Design Rewriting with Maude. In *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008a.

Roberto Bruni, Alberto Lluch Lafuente, Ugo Montanari, and Emilio Tuosto. Style Based Architectural Reconfigurations. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (94):161–180, February 2008b.

Declarative Security Specification of Virtual Networks

(Position Paper)

Serdar Cabuk Chris I. Dalton
Hewlett-Packard Labs, Bristol, United Kingdom
serdar.cabuk@hp.com,cid@hp.com

HariGovind Ramasamy
IBM T.J. Watson Research Center, Hawthorne, USA
hvramasa@us.ibm.com

Matthias Schunter
IBM Zurich Research Lab, Rüschlikon, Switzerland
mts@zurich.ibm.com

Abstract

We believe that end-users should not be required to specify how network security is provided. Instead, users should declare the isolated network as well as the corresponding security objectives (confidentiality, integrity) for each of these networks. We call this idea "Trusted Virtual Domains (TVD)".

As a starting point towards fully automating virtual machine security, we have proposed a secure network virtualization framework that helps realize the abstraction of TVDs (see [CDRS07] for the details). The framework allows groups of related virtual machines running on separate physical machines to be connected together as though there were on their own separate network fabric and, at the same time, helps enforce cross-group security requirements such as isolation, confidentiality, security, and information flow control. The framework uses existing network virtualization technologies, such as Ethernet encapsulation, VLAN tagging, and VPNs, and combines and orchestrates them appropriately to implement TVDs.

1. Introduction

Today's virtual network implementations for VMMs are usually virtual switches or bridges that connect the virtual network cards of all VMs to the actual physical network card of the physical machine. All VMs can potentially see all traffic; hence, no isolation or other security guarantees can be given. While that level of security may be sufficient for individual and small enterprise purposes, it is certainly not sufficient for larger-scale, security-critical operations. For example, in a virtualized data center that hosts services belonging to multiple customers on the same physical infrastructure, accidental data leakage between VMs belonging to different customers is unacceptable.

Our focus in this position paper is security-enhanced network virtualization, which (1) allows groups of related VMs running on separate physical machines to be connected together as though they were on their own separate network fabric, and (2) enforces cross-group security requirements such as isolation, confidentiality, integrity, and information flow control.

2. Security Objectives and Policies

We describe the security objectives of network virtualization using a security model that enables the automatic enforcement of the objectives. The policies define integrity, confidentiality, isolation, and information flow control requirements.

2.1 Trusted Virtual Domains

A TVD is represented by a set of distributed virtual processing elements (VPE) (e.g., virtual machines) and a communication medium interconnecting the VPEs, and provides a policy and containment boundary around those VPEs. VPEs within each TVD can usually communicate freely and securely with each other. At the same time, they are sufficiently isolated from outside VPEs, including those belonging to other TVDs. Here, isolation loosely refers to the requirement that a dishonest VPE in one TVD cannot send messages to a dishonest VPE in another TVD, unless the inter-TVD policies explicitly allow such an information flow.

Each TVD has an associated *infrastructure* whose purpose is to provide a unified level of security to member VPEs, while restricting the interaction with VPEs outside the TVD to pre-specified, well-defined means only. Unified security within a domain is obtained by defining and enforcing *membership requirements* that the VPEs have to satisfy before being admitted to the TVD and for retaining the membership. Each TVD defines rules regarding inbound and out-bound network traffic. Their purpose is to restrict communication with the outside world.

2.2 Security within a TVD

Within a TVD, all VPEs can freely communicate with each other while observing TVD-specific integrity and confidentiality requirements. For this purpose, intra-TVD communication may take place only over an authenticated and encrypted channel (e.g., IPsec), or alternatively, a trusted network[1]. The trusted network alternative may be reasonable in some situations, e.g., within a data center. Informally, integrity means that a VPE cannot inject "bad" messages and pretend they are from another VPE. Confidentiality refers to the requirement that two honest VPEs (in the same TVD or different TVDs) can communicate with each other without an eavesdropper learning the content of the communication[2].

Admission control and membership management are important aspects of TVDs. A TVD should be able to restrict its membership to machines that satisfy a given set of conditions. For example, a TVD may require certificates stating that the platform will satisfy certain properties before allowing the platform to join the TVD.

Member VPEs may be required to prove their eligibility on a continual basis either periodically or on-demand. For example, members may be required to possess certain credentials such as certificates or may be required to prove that they satisfy some integrity properties (property-based attestation. The conditions may vary for different types of VPEs. For example, servers and workstations may have different TVD membership requirements. Some VPEs may be part of more than one TVDs, in which case they

[1] A network is called *trusted* with respect to a TVD security objective if it is trusted to enforce the given objective transparently. For example, a server-internal Ethernet can often be assumed to provide confidentiality without any need for encryption.

[2] Addressing covert channels would exceed the scope of this paper.

from/to	TVD_α	TVD_β	TVD_γ
TVD_α	$\mathbf{1}^*$	$\mathbf{0}^*$	$P_{\alpha\gamma}$
TVD_β	$\mathbf{0}^*$	$\mathbf{1}^*$	$\mathbf{0}$
TVD_γ	$P_{\gamma\alpha}$	$P_{\gamma\beta}$	$\mathbf{1}$

Implemented in our Xen-based prototype.

Figure 1. Example Flow Control Policy Matrix for Three TVDs.

would have to satisfy the membership requirements of all the TVDs they are part of. For a VPE to simultaneously be a member of multiple TVDs, the individual TVD membership requirements must be conflict-free.

2.3 Security across TVDs

As shown in Figure 1, inter-TVD communication can be broadly classified into three types: (1) *controlled* connections, represented by policy entries in the matrix, (2) *open* or unrestricted connections, represented by **1** elements in the matrix, and (3) *closed* connections, represented by **0** elements in the matrix.

Inter-TVD security objectives are independently enforced by each of the individual TVDs involved. To facilitate such independent enforcement, global security objectives are decomposed into per-TVD security policies. The advantage of such a decentralized enforcement approach is that each TVD is shielded from security failures in other TVDs. Security objectives may take different forms; here, we focus on information flow control among the TVDs.

An information flow control matrix is a simple way of formalizing the system-wide flow control objectives. Figure 1 shows a sample matrix for three TVDs: TVD_α, TVD_β, and TVD_γ. Each matrix element represents a policy specifying both permitted inbound and outbound flows between a pair of TVDs, as enforced by one of the TVDs. The **1** elements along the matrix diagonal convey the fact that there is free information flow within each TVD. The **0** elements in the matrix are used to specify that there should be no information flow between two TVDs, e.g., between TVD_α and TVD_β.

An information flow from one TVD to another will be overseen by both the sender TVD and the recipient TVD. Information flow control from one TVD to another is specified by two policies, with each TVD independently enforcing one. For example, $P_{\alpha\beta}$, which represents the information flow policy from TVD_α to TVD_β, would consist of two sub-policies: (1) $P^{\text{in}}_{\alpha\beta}$, which would be enforced by the recipient TVD, TVD_β, and is concerned with the integrity protection of TVD_β, and (2) $P^{\text{out}}_{\alpha\beta}$, which would be enforced by the recipient TVD, TVD_α, and is concerned with the confidentiality protection of TVD_α. The distribution of policy enforcement to both TVDs means that the recipient TVD does not have to rely solely on elements of the sender TVD to enforce rules regarding its inbound traffic.

Controlled connections restrict the flow between TVDs based on specified policies. The policies are enforced at TVD boundaries (at both TVDs) by appropriately configured firewalls (represented in Figure 3 by entities marked FW). The TVD master may push pre-checked configurations (derived from TVD policies) into the firewalls during the establishment of the TVD topology. If available, a management console at the TVD master may be used to manually set up and/or alter the configurations of the firewalls. A TVD firewall has multiple virtual network interface cards, one card for the internal VLAN that the firewall protects and one additional card for each TVD that the members of the protected TVD want to communicate with.

Open connection between two TVDs means that any two machines in either TVD can communicate freely. In such a case, the firewalls at both TVDs would have virtual network cards for the peer domain and simply serve as bridges between the domains. For example, different zones in a given enterprise may form different TVDs, but may communicate freely. As another example, two TVDs may have different membership requirements, but may have an open connection between their elements. Open connection between two domains may be implemented using an unlimited number of virtual routers. In a physical machine that is hosting two VMs belonging to different TVDs with an open connection, the corresponding vSwitches may be directly connected. Communication between two TVDs, while open, may be subject to some constraints and monitoring. For example, a TVD master may permit the creation of only a few virtual routers on certain high-assurance physical machines for information flow between the TVD and another TVD with which the former has an open connection.

A closed connection between two TVDs can be seen as a special case of a controlled connection in which the firewall does not have virtual network card for the peer TVD. In addition to the firewall filtering rules, the absence of the card will prevent any communication with the peer TVD.

3. Secure Virtual Networks

In this section we introduce the networking components forming the framework. We then present the composition of the components to form TVDs and to enforce TVD policies, and describe the management of the TVD infrastructure. Here, we focus on the static behavior of a secure network virtualization framework that is already up and running.

The main aim of our network virtualization extensions is to allow groups of related VMs running on separate physical machines to be connected together as though they were on their own separate network fabric. In particular, we would like to be able to create arbitrary virtual network topologies independently of the particular underlying physical network topology.

Our network virtualization extensions must also be inter-operable with existing non-virtualized entities (e.g., standard client machines on the Internet) and allow our virtual networks to connect to real networks.

3.1 Networking Components

One option for virtual networking is to virtualize at the IP level. However, to avoid problems regarding the support for non-IP protocols and IP support services (such as ARP) that sit directly on top of the Ethernet protocol, we have chosen to virtualize at the Ethernet level.

Our secure network virtualization framework allows multiple VMs belonging to different TVDs to be hosted on a single physical machine. The framework obtains isolation among various TVDs using a combination of virtual LANs (VLANs) and virtual private networks (VPNs). There is one *internal* VLAN for each TVD; an *external* VLAN may be used for communication with other TVDs and TVD-external entities. In the absence of a trusted underlying physical network, each VLAN segment (i.e., an Ethernet broadcast domain, as in our case) may employ an optional VPN layer to provide authentication, integrity, and confidentiality properties.

The networking infrastructure consists of a mixture of virtual entities and physical entities. Virtual entities include VMs, vSwitches, VLAN taggers, VPN, and gateways. Physical entities include the physical hosts and the physical networking infrastructure, which includes VLAN-enabled physical switches, routers, and ordinary Ethernet switches.

Virtual Ethernet cards or *vNICs* are the basic building blocks of our design. Each VM can have one or more vNICs. Each vNIC can be associated with at most one VLAN.

Each virtual LAN segment is represented by a *virtual switch* or a *vSwitch*. A VM appears on a particular VLAN if one of its vNICs is "plugged" into one of the switch ports on the vSwitch

forming that segment. The vSwitch behaves like a normal physical switch. Ethernet broadcast traffic generated by a VM connected to the vSwitch is passed to all VMs connected to that vSwitch. Like a real switch, the vSwitch also builds up a forwarding table based on observed traffic so that non-broadcast Ethernet traffic can be delivered in a point-to-point fashion to improve bandwidth efficiency.

The vSwitch is designed to operate in a distributed fashion. The VMM on each physical machine hosting a VM connected to a particular VLAN segment hosts part of the vSwitch forming that VLAN segment. A component of the VMM captures the Ethernet frames coming out of a VM's vNIC. The component is configured to know which vSwitch the VM is supposed to be connected to.

The VM Ethernet frames are encapsulated in IP packets or tagged with VLAN identifiers. The actual encapsulation is performed by an encapsulation module on request by the vSwitch. The vSwitch component then maps the Ethernet address of the encapsulated Ethernet frame to an appropriate IP address. The mapping allows the encapsulated Ethernet frame to be transmitted over the underlying physical network to physical machines hosting other VMs connected to the same physical LAN segment. The result is the same as when all VMs on the VLAN segment are connected by a real LAN. The IP address chosen to route the encapsulated Ethernet frames over the underlying physical network depends on (1) whether the encapsulated Ethernet frame is an Ethernet broadcast frame, and (2) whether the vSwitch has built up a table of the locations of the physical machines hosting other VMs on that particular physical LAN segment. The entries in such a table would be based on traffic observed on that physical LAN segment.

IP packets encapsulating *broadcast* Ethernet frames are given a *multicast* IP address and sent out over the physical network. Each VLAN segment has an IP multicast address associated with it. All physical machines hosting VMs on a particular VLAN segment are members of the multicast group for that VLAN segment. This ensures that all VMs on a particular VLAN segment receive all broadcast Ethernet frames from other VMs on that segment, whereas VMs on a different VLAN segment do not.

Encapsulated Ethernet frames that contain a directed Ethernet destination address are either flooded to all the VMs on a particular LAN segment (using the IP multicast address as in the broadcast case) or sent to a specific physical machine IP address. The particular choice depends upon whether the vSwitch component on the encapsulating VM has learned the location of the physical machine hosting the VM with the given Ethernet destination address based on traffic observation through the vSwitch.

Encapsulating Ethernet frames from VMs within IP packets allows us to connect different VMs to the same VLAN segment as long as the physical machines hosting these VMs have some form of IP-based connectivity (e.g., a WAN link) between them. There are no restrictions on the topology of the underlying physical network.

We employ VLAN tagging, an existing technology, as an alternative to Ethernet encapsulation for efficiency purposes. Each VLAN segment may employ its own *VLAN tagger(s)* to tag its Ethernet frames. The VLAN identifier, which is unique for each VLAN within a virtual network, is used as tagging information. The tag is then used by the VLAN switch to distinguish traffic flows from the various VLAN segments that connect to the switch.

A VLAN-enabled physical switch (or a *VLAN switch*, for short) connects two or more VLAN segments belonging to the same VLAN. VLAN switches should not to be confused with vSwitches. VLAN switches are part of the physical networking infrastructure, whereas vSwitches are virtual entities. Each VLAN segment is connected to a port on the VLAN switch. Multiple VLANs (i.e., VLAN segments belonging to different TVDs) may also connect to the same VLAN switch. The VLAN switch must be appropriately configured to guarantee isolation among segments belonging to different VLANs, while at the same time connecting physical machines, VMs, and vSwitches on the same VLAN to each other.

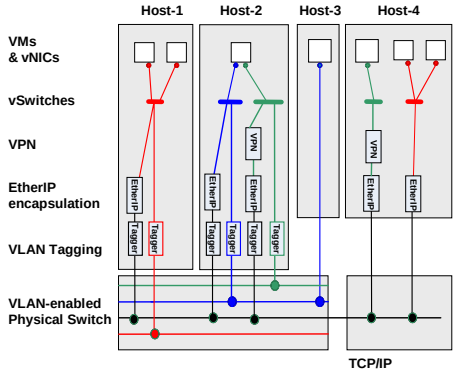


Figure 2. Components of the Secure Virtual Networking Infrastructure

Routing within Virtual Networks: Routing functionality within a virtual network may be implemented by the use of a dedicated VM with multiple vNICs. The vNICs are plugged into ports on the different vSwitches between which the VM has to provide routing services. Standard routing software is then configured and run on the VM to provide the desired routing services between the LAN segments connected.

Communication with Non-Virtualized Systems: Gateways enable communication with systems that live in the non-virtualized world. The gateway is simply a VM with two vNICs. One of the vNICs is plugged into a port on a vSwitch. The other vNIC is bridged directly onto the physical network. The gateway has two main roles. Firstly, it advertises routing information about the virtual network behind it so that hosts in the non-virtualized world can locate the VMs residing on the virtual network. Secondly, the gateway converts packets to and from the encapsulated format required by our virtual networks.

3.2 Composition of Secure Virtual Networks

Figure 2 shows how the networking components can be composed into a secure networking infrastructure that provides isolation among different TVDs, where each TVD is represented by a different color (blue, green, or red). A non-virtualized physical host, such as Host-3, is directly connected to a VLAN-enabled physical switch without employing a vSwitch. Further, a VM can be connected to multiple VLAN segments using a different vNIC for each VLAN segment; hence, the VM can be a member of multiple TVDs simultaneously. For example, the lone VM in Host-2 of Figure 2 is part of two VLAN segments, each represented by a vSwitch with a different color; hence, the VM is a member of both the blue and green TVDs.

Abstractly speaking, it is as if our secure virtual networking framework provides colored networks (in which a different color means a different TVD) with security guarantees (such as confidentiality, integrity, and isolation) to higher layers of the virtual infrastructure. Internally, the framework provides the security guarantees through admission control and the appropriate composition and configuration of VLANs, VPNs, gateways, routers, and other networking elements.

Ethernet frames originating from the source node are handled differently depending on whether the source node is virtualized and whether the destination node resides in the same LAN. We illustrate frame-processing alternatives for different scenarios in Figure 2. For a virtualized domain (e.g., Host-1), each frame is tagged us-

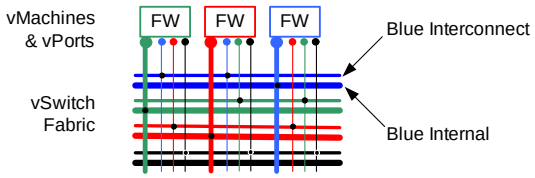


Figure 3. Internal- and Inter-connections for each TVD Type.

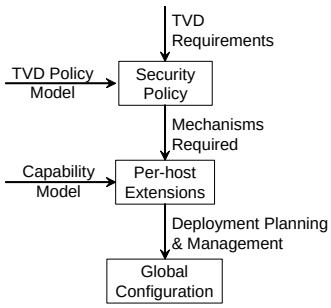


Figure 4. Steps in Auto-Deployment of TVDs

ing the IEEE 802.1Q standard for VLAN tagging [IEE98]. If the destination of the Ethernet frame is a VM on another host that is connected to the same VLAN-capable switch (e.g., another physical domain in a datacenter), this tag indicates the VLAN segment to which the VM belongs. If the destination is a host that resides outside the LAN domain (e.g., Host-4), the VLAN tag forces the switch to bridge the connection to an outgoing WAN line (indicated by the black line in the VLAN-enabled physical switch of Figure 2) that is connected to a router for further packet routing. In this case, the VM Ethernet frames are encapsulated in IP packets to indicate the VLAN segment membership. Lastly, if a non-virtualized physical host is directly connected to the VLAN switch (e.g., Host-3), no tagging is required for the outgoing connection from the host's domain.

4. Auto-deployment of TVDs

Figure 4 shows the steps involved in automatic deployment of secure virtual infrastructures as TVD configurations.

First, the virtual infrastructure topology must be decomposed into constituent TVDs, along with associated security requirements and policy model. Second, a *capability model* of the physical infrastructure must be developed. Capability modeling is essentially the step of taking stock of existing mechanisms that can be directly used to satisfy the TVD security requirements. In this paper, we consider the case where both steps are done manually in an offline manner; future extensions will focus on automating them and on dynamically changing the capability models based on actual changes to the capabilities.

4.1 Capability Modeling of the Physical Infra-structure

Capability modeling of the physical infrastructure considers both functional and security capabilities. The functional capabilities of a host may be modeled using a function $C : H \leftarrow \{VLAN, Ethernet, IP\}$, to describe whether a host has VLAN, Ethernet, or IP support. Modeling of security capabilities includes two orthogonal aspects: the set of security properties and the assurance that these properties are actually provided.

5. Conclusion

We have surveyed our work on automatic provisioning of network security based on a declaratioin of the security requirements and existing capabilities of the infrastructure.

This is a first step towards declarative security for datacenters. We believe that in the mid-term, a similar concept for managing storage security will emerge.

An important open challenge remains how to include legacy systems into such a framework and how to seamlessly provide security for virtual as well as non-virtualized systems.

Acknowledgements

We thank the other authors of [BGJ+05], Anna Fischer (HP Labs), and Konrad Eriksson and Augustin Fievet (IBM Research) for valuable inputs. This work has been partially funded by the European Commission as part of the OpenTC project `www.opentc.net`.

References

[BCP+08] Stefan Berger, Ramón Cáceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. Tvdc: managing security in the trusted virtual datacenter. *SIGOPS Oper. Syst. Rev.*, 42(1):40–47, 2008.

[BGJ+05] A. Bussani, J. L. Griffin, B. Jansen, K. Julisch, G. Karjoth, H. Maruyama, M. Nakamura, R. Perez, M. Schunter, A. Tanner, L. van Doorn, E. V. Herreweghen, M. Waidner, and S. Yoshihama. Trusted Virtual Domains: Secure Foundation for Business and IT Services. Research Report RC 23792, IBM Research, November 2005.

[CDRS07] Serdar Cabuk, Chris I. Dalton, HariGovind Ramasamy, and Matthias Schunter. Towards automated provisioning of secure virtualized networks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 235–245, New York, NY, USA, 2007. ACM.

[IEE98] IEEE. IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks. Technical Report ISBN 0-7381-3662-X, 1998.

High-level system configuration

Thomas Delaet
Department of Computer Science, K.U.Leuven, Belgium
thomas@cs.kuleuven.be

Wouter Joosen
Department of Computer Science, K.U.Leuven, Belgium
wouter@cs.kuleuven.be

Abstract

The high rate of requirement changes make system administration a complex task. This complexity is further influenced by the increasing scale, unpredictable behaviour of software and diversity in terms of hardware and software. In order to deal with this complexity, system configuration tools have been proposed. The processes that many system configuration tools advocate are kept close to manual system administration. We feel that this approach has failed to address the complexity of system administration in the real world. In this paper, we advocate a higher-level language for system configuration, prototyped in PoDIM. We introduce the use of constraints, dependency modeling and platform independence in the context of system configuration. We believe that high-level languages are needed to reduce system administration complexity. PoDIM is one step in that direction.

Keywords PoDIM, system configuration, configuration management, network configuration, autonomic configuration

1. Introduction

The fact is that configuration errors are the biggest contributors to service failures (between 40% and 51%). Configuration errors also take the longest time to repair (Patterson 2002; Oppenheimer 2003; Narain 2004). As the complexity of computer infrastructures increases, the risk of configuration errors increases likewise and introduces even higher change costs. Changes to a configuration can be technically - such as software upgrades - or business oriented. A difficulty with configuration changes is the high number of dependencies between systems. Systems do not operate in isolation, but in a network. A change in the configuration of one networked service may cause a complex chain of changes in dependent services. Furthermore, infrastructural complexity is influenced by increasing scale, unpredictability in software behaviour and systems variety (Evard 1997; Strassner 2005; Anderson 2006).

1. **scale**: The number of network devices, servers, desktops and laptops in a typical infrastructure is increasing significantly. New kinds of devices such as PDA's, mobile phones and sensor nodes are extending the scope of an organizational computer infrastructure.
2. **unpredictability in software behaviour**: Increasingly complex software systems tend to have more bugs, viruses and vulnerabilities. Bugs in software, viruses and vulnerabilities make full control over the system's behaviour an illusion (Burgess 2001a).
3. **systems variety**: Computer infrastructures have a large variety in terms of hardware platforms, operating systems and application software. Our definition of infrastructures includes not only desktops, servers and laptops, but also embedded devices such as palmtops, mobile phones and network devices such as routers and switches. All of these devices run on a variety of operating systems and accompanying application software.

Using a network shell or a configuration management language whose process is close to manual system administration simply does not work in large and varied computer infrastructures with complex software systems. Indeed, the subtle interactions between (different versions of) software packages can make systems with the same operating system and hardware platform unique. According to (Anderson and Couch 2004), the cost per unit becomes excessively large when using manual management processes. In our opinion, more loose, higher-level, system configuration tools are necessary.

PoDIM is our proof of concept of such a higher-level tool. PoDIM's three main building blocks are the specification of constraints, explicit dependency modeling and platform independence. These three features, who are not present in the current state of the art, allow for expressing system configurations closer to what the system administrator wants, instead of how he wants it.

The remainder of this article is structured as follows. We introduce PoDIM's features in Section 2. In Section 3.1 we look at the state of the art in system configuration. Next, we present the results of our research on PoDIM in Section 4 and end with a Conclusion.

2. High-level system configuration

PoDIM defines a language that allows the specification of high-level policies for a computing infrastructure as a whole. Without going in the details of the language syntax, we discuss some examples that illustrate PoDIM's advancement over the state of the art in system configuration.

2.1 Constraints

The state of the art in configuration management allows assigning roles to machines and setting high-level parameters for those roles. "Configure machine X to be a web server" and "configure machine Y to be DHCP server" are examples of role assignments. "The web server must run on port 80" is an example of a parameter assignment. The PoDIM language allows a higher level of abstraction. Instead of role assignments, we can express things such as: "One of my servers must be configured as a web server" and "On every subnet, there must be two DHCP servers". Instead of parameter assignments, we can express things such as: "A web server must use a port number higher than 1024".

R2D2 May 12–13, 2008, Cambridge, UK.

Constraints-based configurations allow for easy reconfiguration when something goes wrong. When one the assigned DHCP servers goes down, another machine can be automatically configured as DHCP server. In the case where you specify a rule like "configure machine Y to be DHCP server", you will have to change the policy rule before a new DHCP server becomes active.

2.2 Dependencies

PoDIM allows the modeling of fine-grained dependencies between different services on one system or between systems. Examples of such dependencies are: "Every mail server needs to put the correct MX records in the DNS server configuration file for the domains it need to serve" or "Every service needs to plug holes in the firewalls to allow communication with its clients". Making these dependencies explicit prevents inconsistent configurations and thus reduces the error rate in an infrastructure.

2.3 Platform Independence

Last, PoDIM makes abstraction of the operating systems and implementation software used on a device. Most infrastructures operate in a mixed environment of different types of Unix and Windows machines. PoDIM allows to specify rules like the "2 DHCP servers on each subnet" rule, independent of the operating system or DHCP server software used. It takes care of generating a configuration file for the DHCP server based on these system characteristics.

3. State of the art in system configuration

Related work of PoDIM's high-level language includes configuration management tools. We also discuss how generic policy languages and a model finder are related to the problem PoDIM is trying to solve. We end this discussion with a characterization of application deployment frameworks.

3.1 Configuration Management Tools

Bcfg2 (Desai et al., 2005, 2006), Cfengine (Burgess 1995, 1993, 1994-, 2001b), LCFG (Anderson and Scobie 2002; Anderson; Anderson and Scobie 2000) and Puppet (Kanies, 2006) are the most cited configuration management tools. Bcfg2 and Cfengine provide a syntax that is very close to manual system administration. LCFG and Puppet include capabilities for modeling dependencies between configurations. None of these tools support the specification of constraints or allows the specification of system configurations in a platform independent way.

3.2 Policy Languages

The PCIM (Moore et al. 2001; Moore 2003) (Policy Core Information Model) and CIM (Common Information Model) (cim) initiatives define a generic model for representing policy specifications on one side and a set of domain classes on the other side. The generic model defines policy rules in a Condition-Action format. The model includes definitions for common network functionalities such as routing protocols, network configurations and IPSec configurations. The domain model itself is object-oriented and models relations between classes. The CIM domain model provides a valuable repository of existing domain knowledge, modeled as object-oriented classes. The CIM model is platform independent, but it has no support for specifying fine-grained dependencies. PCIM/CIM also does not support constraint handling.

Other frequently cited policy languages such as Ponder (Damianou 2002) and JRules (ILOG 2002). Policy rules are Event-Condition-Action based in these languages. Again, these languages do not support the specification of constraints.

3.3 Model Finding

In (Narain 2005) a model finding approach is proposed for configuration management based on Alloy (Jackson 2002, 2006; AA). This approach is based on creating a model for an infrastructure based on first-order logic. Based on a number of inputs (such as the number of devices and network interfaces) an outcome is constructed that satisfies the model. The advantage of using a tool such as Alloy is that it allows very advanced reasoning over a configuration. The same model can be used to generate and validate configurations. The limitations are that some constraints, as we discussed them in this paper, can not be modeled. It is also difficult to set specific attributes of "things". For example, it is difficult to set a human readable name for every device.

3.4 Application Deployment Frameworks

Application deployment frameworks like SmartFrog (Low and Guijarro 2004; Laboratories 2007), Spring (spring) and JBoss Microcontainer (Group) manage applications directly by tuning their parameters. The difference with PoDIM is that PoDIM represents real world things that are only changed when a stable state is reached. Because of this difference, application deployment frameworks listed above do not provide constraint resolution of the kind that PoDIM uses.

4. Results

Our proof of concept implementation supports the configuration of mail servers, firewalls, DHCP and static IP addresses as an example subset of functionality. A more extensive description of PoDIM capabilities can be found in (Delaet and Joosen 2007) and (Delaet et al. 2008). (Delaet et al. 2008) also discusses how PoDIM can be used with an existing system configuration tool (in this case: LCFG).

5. Conclusion

The specification of constraints allows dynamic reconfigurations, which in turn results in less work for the system administrator and faster reaction to unexpected behaviour. I.e. When a system goes down, no rules need to be changed and the tool will automatically find other systems to take over the tasks of the original system.

Modeling dependencies and platform independence aid the automatic migration (as a consequence of the constraint based specification) of services to other machines.

In short, we believe that tool support for high-level system configuration will result in lesser configuration errors and a lower total cost of ownership for large computing infrastructures.

References

AA. The Alloy Analyzer. `http://alloy.mit.edu`.

Paul Anderson. Lcfg homepage. http://www.lcfg.org.

Paul Anderson. Short topics in system administration 14: System configuration. Published by the Usenix Association, Berkeley, CA, 2006.

Paul Anderson and Alva Couch. What is this thing called "system configuration"? LISA Invited Talk, November 2004.

Paul Anderson and Alastair Scobie. Large scale Linux configuration with LCFG. In USENIX, editor, *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10–14, 2000, Atlanta, Georgia, USA*, pages 363–372, Berkeley, CA, USA, 2000. USENIX. ISBN 1-880446-17-0.

Paul Anderson and Alastair Scobie. LCFG - the Next Generation. In *UKUUG Winter Conference*. UKUUG, 2002. URL `http://www.lcfg.org/doc/ukuug2002.pdf`.

M. Burgess. A site configuration engine. *Computing systems (MIT Press: Cambridge MA)*, 8:309, 1995.

M. Burgess. *GNU cfengine*. Free Software Foundation, Boston, Massachusetts, 1994-.

M. Burgess. Cfengine www site. *http://www.iu.hio.no/cfengine*, 1993.

M. Burgess. Needles in the cray stack: The myth of computer control. *Usenix ;login:*, 26(2):30–36, 2001a.

Mark Burgess. Recent developments in cfengine. In *Unix.nl Conference Proceedings*, 2001b.

cim. Common information model (CIM) standards. http://www.dmtf.org/standards/cim/.

Nicodemos C. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, University of London, Departement of Computing, 2002.

Thomas Delaet and Wouter Joosen. PoDIM: A language for high-level configuration management. In *Proceedings of the Large Installations Systems Administration (LISA) Conference*, Berkeley, CA, November 2007. Usenix Association.

Thomas Delaet, Paul Anderson, and Wouter Joosen. Managing real-world system configurations with constraints. In *Proceedings of the 7th International Conference on Networking*, April 2008.

Narayan Desai, Rick Bradshaw, and Joey Hagedorn. Bcfg2 Trac homepage. http://trac.mcs.anl.gov/projects/bcfg2.

Narayan Desai, Rick Bradshaw, Scott Matott, Sandra Bittner, Susan Coghlan, Rémy Evard, Cory Lueninghoener, Ti Leggett, John-Paul Navarro, Gene Rackow, Craig Stacey, , and Tisha Stacey. A case study in configuration management tool deployment. In *Proceedings of the Large Installations Systems Administration (LISA) Conference*, pages 39–46, Berkeley, CA, December 2005. Usenix Association.

Narayan Desai, Rick Bradshaw, and Joey Hagedorn. System management methodologies with Bcfg2. *;login: the USENIX Association newsletter*, 31(1), February 2006. ISSN 1044-6397.

R. Evard. An analysis of unix system configuration. *Proceedings of the Eleventh Systems Administration Conference (LISA XI) (USENIX Association: Berkeley, CA)*, page 179, 1997.

JBoss Group. Jboss microcontainer. http://www.jboss.com/products/jbossmc.

ILOG. Jrules: Technical white paper (version 4.0), 2002. Available at: http://www.ilog.com/products/jrules/.

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, 2006.

Luke Kanies. Puppet: Next-generation configuration management. *;login: the USENIX Association newsletter*, 31(1), February 2006. ISSN 1044-6397.

Luke Kanies. Puppet. http://reductivelabs.com/projects/puppet/.

Hewlett Packard Laboratories. The smartfrog reference manual, 2007.

Colin Low and Julio Guijarro. A smartfrog tutorial. Technical report, Hewlett Packard Laboratories, 2004.

B. Moore. Policy Core Information Model (PCIM) Extensions. RFC 3460 (Proposed Standard), January 2003. URL `http://www.ietf.org/rfc/rfc3460.txt`.

B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy Core Information Model – Version 1 Specification. RFC 3060 (Proposed Standard), February 2001. URL `http://www.ietf.org/rfc/rfc3060.txt`. Updated by RFC 3460.

Sanjai Narain. Network configuration management via model finding. In *LISA'05: Proceedings of the 19th conference on Large Installation System Administration Conference*, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

Sanjai Narain. Towards a foundation for building distributed systems via configuration. http://www.argreenhouse.com/papers/narain/Service-Grammar-Web-Version.pdf, 2004.

D. Oppenheimer. The importance of understanding distributed system configuration. In *Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop*, April 2003.

D. A. Patterson. A simple way to estimate the cost of downtime. In *Proceedings of the Sixteenth Systems Administration Conference (LISA'02)*, pages 185–188, Berkeley, CA, 2002. Usenix Association.

spring. Spring framework. http://www.springframework.org.

John Strassner. Policy management challenges for the future. Policy 2005 Keynote, 2005.

Local Hoare Reasoning about DOM

Philippa Gardner, Imperial College London

Structured data update is pervasive in computer systems: e.g. heap update on local machines, information storage on hard disks, the update of distributed XML databases, and general term rewriting. Programs for manipulating such dynamically-changing data are notoriously difficult to write correctly. We need analysis techniques and verification tools for reasoning about such programs. Hoare reasoning is a well-known technique for reasoning about programs that alter memory. Such reasoning has hardly been studied for other examples of update and there has been little attempt at an integrated theory.

My current research focuses on XML viewed as an in-place memory store, rather than a static XML document. Such a view of XML is fundamental to the W3C Document Object Model (DOM), XLinq and Javascript. E.g., consider a web page with a button to 'today's weather'; click on the button and embedded Javascript (using DOM) puts 'today's weather' in the tree. With Smith, Wheelhouse and Zarfaty, I am studying local Hoare reasoning about DOM [5, 7]. Our work transfers O'Hearn, Reynolds and Yang's local Hoare reasoning for analysing heaps to XML. In particular, we apply recent work by Calcagno, Gardner and Zarfaty on local Hoare reasoning about simple tree update to this real-world DOM application. Our reasoning formally specifies a significant subset of DOM. In addition, it can be used to verify, e.g., invariant properties of simple Javascript programs.

The Document Object Model

DOM specifies an XML update library, and is maintained by the World Wide Web Consortium (W3C). Its purpose is to be: 'a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.' A DOM implementation exists in most popular high-level languages, and is used in many applications for accessing and updating XML.

DOM provides an interesting example of a library which is important enough to be given a comparatively formal specification. Over the years, it has evolved into a specification which, by consensus, seems to be correct. DOM is, however, written in English. This means that it is liable to misinterpretation. E.g., we showed that the Python mini-DOM implementation was incorrect [7, 8]; Orendorff has recently provided a patch. It also means that DOM is not compositional, in that a specification of a composite command cannot be determined directly from the specification of its parts. DOM is thus larger than it need be, specifying e.g. composite commands such as `getPreviousSibling`. It also means that DOM is not complete; e.g., it specifies the `insertBefore` command, but not `insertAfter`. We have given formal, compositional specification of a significant fragment of DOM using local Hoare reasoning. Unlike DOM, we are able to work with a minimal set of commands and obtain complete reasoning for straight-line code.

Local Hoare Reasoning

There has been a recent breakthrough in Hoare reasoning. Researchers previously used Hoare reasoning based on First-order Logic to specify how programs interacted with the *whole* memory. O'Hearn, Reynolds and Yang instead introduced *local* Hoare reasoning based on Separation Logic (SL). Their key idea is to specify how programs interact with a small part of the memory (the *footprint*) touched by a program. Inspired by this work, Calcagno, Zarfaty and I studied local Hoare reasoning about a simple tree update language [3]. We initially assumed that we could base our reasoning on Cardelli and Gordon's Ambient Logic (AL), since AL analyses static trees (firewalls, XML) in a similar way to SL-reasoning about heaps. In fact, we proved that this is not possible [4]. Instead, we had to fundamentally change the way we reason about structured data, by introducing Context Logic (CL) for reasoning about data and contexts. Local data update typically identifies the portion of data to be replaced, removes it, and inserts new data *in the same place*. CL reasons about both data and this place of insertion (contexts).

In [5], Smith, Wheelhouse, Zarfaty and I showed that our local Hoare reasoning about simple tree update scales to DOM's richer tree structure and update commands. Our specification is compositional in that two Hoare triples compose if the post-condition of the first logically implies the precondition of the second. This compositionality enables us to focus on a minimal set of update commands, whereas DOM has to specify all the commands for which a specification is useful: e.g., we can derive the specification of `getPreviousSibling`, whereas DOM specifies it directly. Our specification is complete for straight-line code, which we prove using a standard technique of deriving the weakest preconditions of our commands: e.g., unlike DOM we can derive a specification of `insertAfter`. Finally, our reasoning can be used to verify invariant properties of simple programs: e.g., we show that a program for moving a person to a new address in an address book satisfies an XML schema invariant specifying that an XML document is indeed an address book.

On-going Projects

DOM Core Level 1 DOM is divided into a number of levels, of which the Level 1 is the most fundamental. The Level 1 specification is itself separated into two parts: Core, which 'provides a low-level set of fundamental interfaces that can represent any structured document'; and HTML, which 'provides additional, higher-level interfaces...to provide a more convenient view

of an HTML document'. We focus on the fundamental interfaces in DOM Core Level 1. In [5], we concentrated on the XML tree structure and simple text nodes of DOM Core Level 1, rather than the full DOM structure which also consists of Attributes, DocumentFragments, etc. We took the view that it was important to understand the reasoning about the fundamental tree structure first, especially since DOM treats these other structures as nodes with similar properties. Smith and I are currently extending our reasoning to DOM Core Level 1. So far, there seems to be little additional conceptual reasoning to this extension. The challenge has instead been in the interpretation of the DOM specification: e.g. DOM does not specify how the normalize method of the Element class interacts with pointers in the store, and there is no consensus amongst the different browsers as to the 'right' interpretation.

A Correct DOM Implementation An implementation of XML update, conforming with DOM, should have the same intended behaviour as other DOM implementations on other sites. This only works if the implementation really does conform with DOM. Since DOM is written in English, the interpretation of the precise conditions under which a command applies is error prone (Python mini-DOM is incorrect). Indeed, it is impossible to prove that an implementation of DOM is correct. It is possible using our formal DOM specification.

With Zarfaty, I have integrated our CL reasoning about a simple, high-level tree update language with CL reasoning about a low-level implementation in memory [6]. Such integrated reasoning is not straightforward. It requires matching the high-level footprint, the subtree affected by the high-level update program, and the low-level footprint, that part of the memory representing the subtree plus nearby nodes requiring additional pointer surgery during update. This means that the low-level reasoning relies on specific details about the particular implementation chosen (the specific pointer surgery required). Low-level reasoning is thus not appropriate for tree libraries such as DOM. The next step is to adapt this integrated reasoning to the more complicated DOM setting, to prove that a DOM implementation conforms with our DOM specification.

Verification Tool A future goal is to develop a prototype verification tool for reasoning about programs using DOM. With Calcagno and Dinsdale-Young, I have recently shown that model-checking and validity are decidable for quantifier-free CL applied to simple trees [2]. This result will extend to CL applied to DOM. With quantifiers however, CL for trees is undecidable (as is SL and AL). We will investigate tractable fragments of CL, inspired by the verification tool Smallfoot based on a fragment of SL. We aim for a CL fragment which captures enough of the horizontal field reasoning and vertical path reasoning to verify e.g. that an XML address book schema is an invariant of a simple program which moves a person to a new address [5]. An ambitious challenge is to provide a 'one-click' tool that checks if, e.g., embedded Javascript in a web page can ever violate the schema assertions on that web page.

Concurrent DOM DOM specifies sequential XML update. There are several concurrent DOM-implementations, conforming to the sequential DOM-specification, but no concurrent DOM. This is because DOM is written in English; a concurrent version would be too complicated. In contrast, our formal DOM specification lends itself to a concurrent extension. Wheelhouse and I are studying concurrent local Hoare reasoning about a simple concurrent tree update language based on multi-holed CL [1]. Our eventual aim is to develop concurrent DOM and apply our reasoning to example applications such as Google Doc, where the same document, represented in XML, can be modified concurrently by several users working from different locations.

Outcomes

For sequential reasoning, our ambitious aim is to convince the DOM community that program verification is important and achievable. We do not expect the DOM community to embrace CL right away; CL is new and evolving. We are however aiming for the DOM community to appreciate the value of being able to specify and verify, e.g., simple Javascript programs. It is too early to make such claims for the concurrent case, although we do believe that the goal to specify concurrent XML update is important. An ultimate challenge for us is neatly summarised by Milner in the Grand Challenge on *Global Ubiquitous Computing*, which aims 'to develop a coherent informatic science whose concepts, calculi, theories and automated tools allow descriptive and predictive analysis of global ubiquitous computing at each level of abstraction.'

References

[1] C. Calcagno, T. Dinsdale-Young, and P. Gardner. Adjunct elimination in context logic for trees. In *APLAS*, 2007.

[2] C. Calcagno, T. Dinsdale-Young, and P. Gardner. Decidability results for context logic using automata. In preparation, 2008.

[3] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic & tree update. In *POPL*, 2005.

[4] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: Completeness and parametric inexpressivity. In *POPL*, 2007.

[5] P. Gardner, G. Smith, and M. W. U. Zarfaty. Local Hoare reasoning about DOM. In *PODS*, 2008. Preliminary version in PLanX 2008.

[6] P. Gardner and U. Zarfaty. Integrated reasoning about high-level tree update and its low-level implementation. In preparation, 2008.

[7] G. Smith. *Local Hoare Reasoning about DOM*. PhD thesis, Imperial, in preparation.

[8] M. Wheelhouse. DOM: Towards a formal specification. Master's thesis, Imperial, 2007.

SmartFrog and Data Centre Automation

Patrick Goldsack, Paul Murray, Andrew Farrell, Peter Toft
HP Laboratories, Bristol, UK
patrick.goldsack@hp.com, pmurray@hp.com, andrew.farrell@hp.com, peter.toft@hp.com

Abstract

If we are to facilitate service provision in next generation data centres then we need to tackle a number of challenges which lie at the heart of the automation problem for these data centres, including: scale, reliability, security, and service heterogeneity.

In this position paper, we consider the requirements for a solution to these challenges which entail a shift in philosophy from imperative to declarative models for data centres. An important aspect of such a shift is the replacement of workflow as a mechanism for automation. In its place, we propose a declarative approach based on modelling individual components of a system together with their possible configuration states. Dependencies may be specified between components which guard how components may change configuration states. We determine the actions that may be performed to dynamically achieve target states for the system from these models. We have built an experimental system around these concepts and describe this approach in outline.

1. Background

If we are to facilitate service provision in next generation data centres then we need to tackle a number of challenges which lie at the heart of the automation problem for these data centres, including: scale, reliability, security, and service heterogeneity.

We need to consider how to build and manage very large data centres, capable of running very large numbers of heterogeneous services that are designed and implemented by different customers with appropriate levels of security, reliability and flexibility. This heterogeneity both in terms of the nature of the service and the range of ownership, makes the task far more complex than running sets of relatively homogenous services or where ownership is vertically integrated from hardware through to end service.

A principal benefit of next generation data centres (NGDCs) is the democratization of services. This would allow small to medium enterprises or even individuals to deliver services to the world in the same way as the web has democratized the delivery of information. For such an ambition, we need to find ways to allow these individuals to develop and run systems securely within well-connected infrastructure without the need to own significant capital assets. This requires that service providers support infrastructure on which these services may be delivered. An early entrant into the space of democratized service platforms is Amazon with its EC2 (Amazon EC2) and S3 (Amazon S3) offerings. However, although these infrastructural services show the way to a certain degree, and illustrate the possibilities for the future, they do not yet go far enough towards providing the levels of security, reliably or flexibility required for solid service delivery.

We enumerate the following challenges for NGDCs.

- Scale: simple back-of-the-envelope calculations show that it is perfectly reasonable to envisage data centres that run 10^5 different service instances on roughly an order of magnitude more virtual machines. Systems of this scale have many implications for the architecture of management systems: failure becomes common place, transactions across the system become impossible, global views or optimizations become problematical and centralized decision making becomes intractable.
- Heterogeneity: since the services will be of an arbitrary nature, decided on by the service owners and not the operators of the data centre, simplifications of aspects such as resource management that come from homogeneity or single ownership cannot be assumed. The data centre management system cannot have accurate knowledge of the services as these are not pre-defined and may change on the fly.
- Security: since in most cases the service code cannot be trusted by the data centre, nor by other services running on the same infrastructure, strong boundaries will need to be put in place between the services and the infrastructure. However, there will still be the need for the services to be managed (including autonomically) and to interact with the core system to achieve end goals. Effectively, therefore, a complete system is defined by the potential interactions of 10^5 individual management entities with limited visibility of the other entities and their states.
- Reliability: With failure being common-place, both of physical components and software, reacting to failure and taking prompt corrective action is necessary within the management system. Furthermore, it is impossible to build systems of this scale without bugs. Consequently self-repair is vital, as is the handling of the failure across the many independent management systems which may be competing to 'fix' the failures within their own sphere of influence.
- Churn: systems of that scale will always suffer churn including the coming and going of services, service reconfigurations due to changing circumstances and requirements, temporary rescaling of a service due to load, short-term overload and congestion, and so on. All of these are occurring asynchronously with respect to each other and to changes in the underlying infrastructure (failure and repair of components, the addition of new and the retiring of old equipment, etc).

These properties make the design of management systems particularly challenging. In this work, we are principally concerned with the orchestration of management actuators, which effect low-level or primitive actions on system components.

2. The Imperative Data Centre

Large-scale management systems traditionally depend on *workflow* execution engines that are triggered in response to an administrator request or, in the case of autonomic systems, some set of internal events. These actions range from provisioning one or more servers or storage volumes, through to the deployment and configuration of software packages. In many cases the actions will range over a

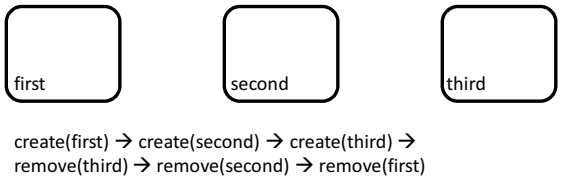


Figure 1. Management Workflow for Simple Scenario

number of devices, via interaction with a number of low-level actuators, with complex ordering requirements to ensure that system properties are not violated.

A workflow, according to (The Workflow Management Coalition February 1999), is "[t]he automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules". The sense of this definition is very much an imperative or procedural one: work passes from one agent to the next for actioning. Thus, a workflow entails an ordering of actions to be carried out by agents – a recipe for getting a task done.

It is useful to make a further distinction between localised and distributed workflow. Traditionally, with its root in automating production lines and latterly being used for the automation of office procedures, workflow has been principally characterised by a single controller co-ordinating the passing of work between agents. This is a highly localised view of workflow.

In more recent times, there has been a tendency to associate the term workflow with a collection of distributed procedures, where each procedure is the logic associated with effecting the workflow within any one of the participating entities – an end-point perspective. This distributed view of workflow is grounded within the Remote Procedure Call (RPC) paradigm, where communication between agents is prescriptively synchronised (Alonso et al. 2004). Web Services Composition Languages such as `WS-BPEL` (OASIS) and `WS-CDL` (WS-CDL W3C Working Group) have been defined with the purpose of respectively capturing localised and distributed aspects of this RPC view of distributed workflow.

Unfortunately, distributed workflow-based control of dynamic systems is substantially flawed. Many of these flaws are rooted in the fundamental characteristic of workflow-based systems that they are concerned with prescribing orderings or recipes of actions to be carried out on/by a number of agents or system components. The alternative, as we shall describe, would be to describe the system components themselves as well as relationships between them and have the recipes of actions organically emerge from the declarative models of components. This promotes a much more loosely-coupled approach to federation of distributed system components.

A very simple example is the following. Let's say that there is a system of three managed entities, on which we wish to orchestrate management actions. Each managed entity may be 'created' and subsequently 'removed'. The initial state for each entity is that it is neither created nor removed. Consider also that the second may not be created until the first has been created. Similarly, the third may not be created until the second has been created. Conversely, the second may not be removed until the third has been removed. Similarly, the first may not be removed until the second has been removed. This scenario is captured in Figure 1. If we were to capture the behaviour described (there may be other possible behaviours not described) as a workflow, it would be as a single six-action sequence of actions for creating and removing the managed entities, as depicted in the figure.

Workflow-based management systems typically suffer from the following issues:

- Workflows are hard to analyze, manipulate, and reason about. In just capturing recipes of actions, the state of components is only implied. It's hard to see which action belongs to which component, and it is hard to reason about the states of individual components.
- Capturing the full range of procedures for managing a system of components may require a large number of workflows, which may or may not be aggregated into one or more super-workflows. The overhead in maintaining such workflows is huge because of the nature of what you are representing, namely recipes/procedures for management actions.
- In a system the size of a large data centre there may be hundreds of workflows active at any time: provisioning systems, deploying software, and so on. Each of these is competing for resources and interfering over required interactions with the entities they are attempting to control.
- Workflows are very fragile to changes in the underlying assumptions about a system, such as when a new technology is introduced. Such changes generally force either refactoring or outright abandonment of existing workflows.
- Workflows can be very long-lived - each action taking from seconds to weeks (as would be the case if the steps involve the aspects such as the ordering of new equipment). It is essential to be able to assess the current state of the workflow, but as state is only captured in the steps of a workflow, it is hard to understand the current overall state.
- Error handling usually dominates the structure of a workflow: even the failure recovery can fail, often leaving the system in an unknown state. The fundamental issue is that workflows specify idealized or default behaviour, given their roots in specifying the operation of manufacturing production lines. The original workflows were never meant to be subject to exceptional behaviour (van der Aalst and Weske 2005; Casati et al. 1999). If ever there was something that is more ill-suited to the modelling/specification of dynamic data centres it is workflow! Error handling has been put in place as an afterthought in workflow systems, and it is not particularly effective. It is hard to model behaviour, hard to understand behaviour and hard to know whether you have it right.
- Workflow composition is hard: one can either assume independence thus allowing parallel execution, or impose a specific sequence. It is hard to generate more subtle compositions and interleaving, since it is impossible to reason about the dependencies between the steps of the workflows. After all, how can you effectively compose workflows when you are composing recipes? Fundamentally you need to reason over state.

These, and other problems, lead us to consider an alternative approach to the design of the orchestration of actuators - one which provides a more robust framework for the development of large systems.

3. The Declarative Data Centre

We need to find an approach that is more robust for managing these systems – one which promotes a much more loosely-coupled approach to federation of system components. Such an approach is essential for resolving issues of scale, reliability and fault-tolerance, composition and service heterogeneity in next generation data centres. Fundamentally, this means moving away from the RPC-based model for distributed workflows, where the operations of components is rigidly and prescriptively synchronised, to a model where components and their states are explicitly modelled and individual components may decide what to do next based on the *states*

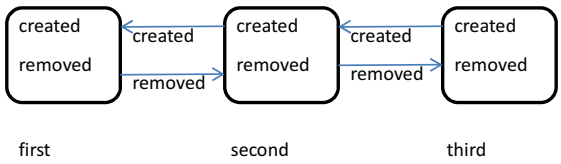


Figure 2. Management Orchestration for Simple Scenario, using Declarative Modelling Approach

of other components. Decision making is kept local, rather than it being prescribed on a global/distributed level.

By explicating components' states and grounding decision making on relationships between states, the know-how for managing systems is captured declaratively. This leads to models of systems which are more intuitive and more easily maintained. From these declarative models, we may deduce a number of possible workflows for carrying out particular management tasks. In this sense, the meaning of such declarative models is mutiplicious in entailing a number of allowable procedures or recipes. Whereas any one workflow would be singular in meaning, in prescribing a single procedure. A declarative model for orchestrating management actions on systems is typically more intuitive and simple than an imperative one.

Returning to the previous example, we would capture the given scenario simply as three components whose modelled state is characterised as containing attribute/value pairs for 'created' and 'removed'. We would specify dependencies between components, as captured by the arrows in Figure 2. These dependencies enforce the ordering prescribed in the example narrative. However, any management workflow, such as the one presented in the previous section, is simply an entailment of this model. There is no absolute prescription regarding the behaviour for individual components, and in this sense their management is much more loosely-coupled. That is, we model what *could* be done and not what *should* be done.

We have been experimenting with an approach which accommodates our view and which is based on a number of simple concepts:

- Each entity within the system is self-managed, working on local visibility of the global state - its local model which represents aspects such as desires and current actualities drive the automation of the entity. These entities include the service domains (representing the services and their owners), as well as major components in the core system such as physical nodes or essential services such as DNS.
- We define distributed interaction through 'model exchange' between entities. Protocols are provided that allow the exchange of model data between these entities with well-understood semantics regarding consistency and timeliness, such as Anubis (Paul Murray 2005). Based around a discovery and group membership protocol, we can provide a highly recoverable and reliable layer that is the core of the system: models that get out of synchrony for some reason will get resynchronized within some 'reasonable' time. Failure is recovered by automatically repopulating the models and driving the local system again to the required state.

The architecture is designed so that inconsistencies between local views will eventually be eliminated (if nothing else changes) and so the overall data centre will tend towards a correct and consistent state. The architecture is also such that any interim inconsistencies do not cause any significant or lasting problems, and in the specific case of security properties, we ensure that at no time is security jeopardised.

The approach does mean that we cannot be certain that the entire data centre is configured correctly for the sum of all requirements from the services that run upon it. However this is impossible in any case with a system of such size, as stability is in practice never reached. Furthermore, optimization is hard to achieve across the extent of the data centre with fully decentralised action and models. However, the truth is that this is also a vain expectation when it comes to such large and dynamic systems.

The advantages of having an explicit declarative statement of the desired state at each entity and the degree to which these desires have been achieved, and a clear semantics so far as the consistency of this information within the extended distributed system is concerned, have been significant. It has made building and debugging our experimental systems much easier than might have been expected. We have far greater separation of concerns, better isolation of specific problems, and a more straight-forward compositional approach to adding new features.

4. SmartFrog and Live-State Dependency Models

Our approach to orchestrating management actions for next generation data centres is based on defining declarative models of managed entities/components and their states, and defining dependencies between the components that prescribe how they may change state. Put simply, the dependencies are requirements of one part of state on another: for example 'service running' requires 'operating system installed'. Thus an assertion that the system desires the service to be running will result in the operating system being installed, and only then will it be deemed possible to install and start the service to achieve the desired state.

The system (Andrew Farrell, Paul Murray and Patrick Goldsack) we have built to support this style of automation consists of three major elements: a comprehensive modelling environment based on the concepts contained in the SmartFrog (Automated Infrastructure Laboratory, HP Labs) notation; a way of defining relationships and dependencies over the models; and a runtime environment based on the SmartFrog runtime which provides a means by which the models may be animated.

We consider a managed entity to consist of the sum of a large number of 'fine-grained' state models each representing some minimal aspect of the entity, for example the operating system on a specific node, or a software package, or a volume, and so on. The modelling can be described at whatever level is required. Actions are attached to these partial state models - actions that know how to create, terminate or modify the 'real-life' equivalent of that aspect in response to changes in the model. These models are known as live-state models: as the model changes the actions ensure that the state of the live system follows, and vice-versa.

We define dependencies between these state models (or indeed arbitrary groupings of these models - known as composite state models). These dependencies are parameterized by the states themselves and may be active or passive dependant on the current state. Actions for states that need to be achieved will then be evaluated in an order satisfying the dependencies.

Parameterizable templates can be written and instantiated, causing desired states to be defined, and actions triggered to satisfy these desires according to these dependencies. Many simultaneous or interleaved changes to the overall system model simply implies that a greater number of states need to be achieved, and their order determined by a larger set of dependencies.

A representation in the SmartFrog modelling language for the scenario presented in Figure 2 is as follows.

```
#include "org/smartfrog/components.sf"
#include "org/.../services/dependencies/statemodel/components.sf"
#include "org/.../services/dependencies/threadpool/components.sf"

ManagedEntity extends State {
 sfClass "org.smartfrog.services.dependencies.examples.ManagedEntity";
   created false;
   removed false;
   sink false;
}

createdDependency extends Dependency {
   enabled LAZY on:created;
   relevant (! LAZY by:created);
}

removedDependency extends Dependency {
   enabled LAZY on:removed;
   relevant LAZY by:created;
}

sfConfig extends Model {

   first extends ManagedEntity;
   second extends ManagedEntity;
   third extends ManagedEntity;

   firstCreated extends createdDependency {
      on LAZY first;
      by LAZY second;
   }

   secondCreated extends createdDependency {
      on LAZY second;
      by LAZY third;
   }

   thirdRemoved extends removedDependency {
      on LAZY third;
      by LAZY second;
   }

   secondRemoved extends removedDependency {
      on LAZY second;
      by LAZY first;
   }

}
```

In this SmartFrog model, we define three instances of a 'ManagedEntity': first, second, and third. We also define the four dependencies previously described between these managed entities. There is also an implementation (in Java) of the 'ManagedEntity' component, which may be found in (Andrew Farrell, Paul Murray and Patrick Goldsack). At the current, formative stage of our work the state transitions that a component may make are hard-coded. This aspect should be captured by the orchestration model, and we are maturing our approach to make this possible. This is essential as the model author should have control over it, and it is appropriate that we are able to reason over the transitions that a component may make.

We are currently characterising the semantics of our orchestration approach mathematically, in order to provide a robust account of its features. We are also considering how we may provide verification and simulation support for orchestration models. An example issue that we would typically seek to verify is an *absence* of locking, i.e. deadlock and livelock. Under normal circumstances, we would seek to avoid deploying models which have a capacity to exhibit deadlock. However, it is not necessarily inappropriate that models exhibit livelock, as long as the model author is aware of the ways in which a model may be subject to such locking. Indeed, it is consistent with the very nature of our approach that components may loop through states an unbounded number of times.

From a mathematical perspective, our modelling approach is very simple. This is an advantage in itself in being simple for a model author to understand. It is not inconceivable for workflow authoring that a model author may not fully understand the complexities and subtleties of the modelling approach. Worse still, workflow models are often deployed without any checks made with respect to their structural and behavioural integrity (van der Aalst 2004). A real benefit of our work will lie in the tools that will be developed to assist a model author understand the true nature of what they are authoring.

5. Current Status

The system outlined is currently a prototype, developed in HP Labs based on the open source SmartFrog system. We have used it in a number of experiments, including in the context of managing very large data centres. We are sufficiently encouraged by the initial experiments to start exploring how best to enhance the current pragmatic programming model with a more sound theoretical basis.

References

Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services, ISBN: 3540440089*. Springer, 2004.

Amazon EC2. Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2`.

Amazon S3. Amazon Simple Storage Service. `http://aws.amazon.com/s3`.

Andrew Farrell, Paul Murray and Patrick Goldsack. Guide to Orchestration in SmartFrog. `http://smartfrog.svn.sourceforge.net/viewvc/*checkout*/smartfrog/trunk/core/smartfrog/docs/SFOrchestration.pdf`.

Automated Infrastructure Laboratory, HP Labs. SmartFrog: Smart Framework for Object Groups. `http://www.smartfrog.org`.

Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Guiseppe Pozzi. Specification and Implementation of Exceptions in Workflow Management Systems. *ACM Trans. Database Syst.*, 24(3):405–451, 1999. ISSN 0362-5915.

OASIS. Web Services Business Process Execution Language Version 2.0, OASIS Standard, 11th April 2007, at: `http://www.oasis-open.org/apps/org/workgroup/wsbpel`.

Paul Murray. A Distributed State Monitoring Service for Adaptive Application Management. In *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan*, pages 200–205, 2005.

The Workflow Management Coalition. Workflow Management Coalition Terminology & Glossary. Document Number: WFMC-TC-1011. Document Status: Issue 3.0. February 1999.

W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management, BPM Center Report BPM-04-03. Technical report, BPMcenter.org, 2004.

W.M.P. van der Aalst and Mathias Weske. Case Handling: a New Paradigm for Business Process Support. *Data Knowledge Engineering*, 53(2):129–162, 2005. ISSN 0169-023X.

WS-CDL W3C Working Group. Web Services Choreography Description Language Version 1.0 W3C Working Draft 17 December 2004, at: `http://www.w3.org/TR/ws-cdl-10`.

What to Declare, How to Declare

Kohei Honda
Queen Mary, University of London
kohei@dcs.qmul.ac.uk

Nobuko Yoshida
Imperial College London
yoshida@doc.ic.ac.uk

This forum, R2D2, asks questions about management of datacentre/cloud computing, where numerous servers and real and virtual resources on them need be dynamically reconfigured and managed to serve diverse needs of clients. It asks for example how we can describe, control, and administer dynamic provisioning of these resources, where services may be placed at different tiers, such as operating systems, databases, and web servers. And it asks whether we can use declarative methods to meet these challenges.

While these questions are asked in the context of datacentre/-cloud computing, it may be interesting, and in fact looks doable, to cast them in a different setting and re-consider these problems from a different viewpoint. This may shed a different light on these problems and may give us a new viewpoint, if not a solution itself.

With this apology, we wish to outline some of the key aspects of our recent standardisation efforts in financial protocols [1, 3, 10] based on the theory of mobile processes [9], and how it may offer a relevant framework to think about some of the research questions relevant to this forum [2, 6, 7]. The standardisation efforts we are engaged in are called ISO20022 [10] (known as UNIversal Financial Industry message scheme, or UNIFI, located at `www.iso20022.org`). The setting of ISO20022 is to have a meta-standard for diverse electronic protocols in the financial industry. It is already a successful standard — most of the international financial protocols are registered in it, and the number of registered protocols are increasing day by day. There are more than a few billion messages per year moved by financial standards in ISO20022.

Currently ISO20022 only allows registration of message formats, in the shape of an UML model, which is mapped to an XML schema when in use. In the next couple of years, it intends to cover, in addition to static message formats, dynamic message exchange as a part of the descriptions of financial protocols. This is where our role as theorists comes in, based on the π-calculus [9] and its types [4, 5]. Towards the practical use of these theories, we developed applied theories [2, 7] as well as language implementations [8].

One of the key features of ISO20022, which has led to its wide adoption, is the use of a high-level modelling language for describing each message format and associated business model, rather than just registering an XML schema. A bare message format is apt to change and does not help interoperability. Rather the registration of a protocol is done as a UML model, along with its XML compilation: this is more abstract, is understandable by a broader public, and promotes interoperability (to enable uniformity and interoperability, each UML model should conform to a meta-model specified in ISO20022).

This key feature, the model-driven approach, needs be carried over to the description of dynamics. It should offer an understandable, flexible modelling framework for choreography [1] of messages which is also usable for varied engineering purposes such as static and dynamic protocol validation. The framework should be scalable in many dimensions: financial protocols are generally asynchronous, they can be very simple or complex, and they can be of an extremely high-volume (say a million messages per hour) and long-running (say 15 years). Moreover, the transport which may deliver financial messages are diverse, including private networks inside a corporation, high-performance networks and Internet.

Thus we need a description framework which is truly general and flexible, as well as offering a clear, simple and understandable description: so that IT architects are sure what each description means; so that they can create a program which acts as an endpoint of a protocol, to be executed correctly in spite of diversity of networks; so that they can automatically generate a monitor which checks message exchange dynamically, either locally or globally; so that automatically generated graphical description of a protocol can be looked at by business analysts and propose an update reflecting business needs; so that an integration expert can direct her/his team to work on integration of two or more protocols, mediating not only message fields but also message exchange. For all these purposes, what we need is a framework for describing and modelling of dynamics of financial protocols which are as general, as understandable, and as usable for engineers, as classes and objects in UML class diagrams.

We contend that the questions being asked in this context, the context of financial protocols' standardisation in ISO20022, have many common technical traits as those being asked in the context of data centre and cloud computing. This position paper is not a place to extend these traits in detail: and through the discussions in the workshop, we shall be able to work on closer comparisons, checking where and how they have commonness and differences. However brief discussions of three potentially salient points may still be due.

First, it looks true that, in the context of this forum too, thinking about a high-level general modelling language is interesting. One of the key merits is that it naturally leads to separation of concerns. In particular, it does not prevent us from using stratification: indeed one of the most exciting technical aspects of ISO20022 is its stratification of abstractions. The description language should be general, in the sense that it should be usable by practically anybody, in any situation in datacentre/cloud computing, deserving to be a widely used open standard that ensure interoperability.

Secondly, while not usually emphasised, datacentre/cloud computing surely includes communication as its essential ingredient. In fact, *MapReduce* is a form of stream-based computing, which in turn is a special case of communication-centred computing. While many parts of datacentre management may not be reducible to communication (the act of provisioning itself looks hard to abstract this way), describing a configuration of systems centring on communication has one notable merit: it gives us a uniform way to describe distributed hosts, which is at least in one level surely a reality of datacentre. We shall need other abstractions, but communicating processes will offer at least one useful framework. Indeed, clouds and their networks are nothing but highly reconfigurable, robust

distributed processes, with an emphasis on persistence and virtualisation.

Third and finally, the preceding viewpoints may suggest that we may broaden the notion of "declarative" approaches to any clear and precise descriptive frameworks, potentially with a well-founded theoretical basis. For example, UML class diagrams offer (probably up to some reservations) such precise, usable description, within its designated scope. What matters is that a modelling framework offers precise and unambiguous descriptions of the target reality.

References

[1] Web Services Choreography Working Group. `http://www.w3.org/2002/ws/chor/`.

[2] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[3] Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming. Published in [1], 2006.

[4] Matthew Hennessy. *A Distributed Pi-Calculus*. CUP, 2007.

[5] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

[6] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Web Services, Mobile Processes and Types. *The Bulletin of the European Association for Theoretical Computer Science*, February(91):165–185, 2007.

[7] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

[8] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, LNCS. Springer, 2008. `http://www.doc.ic.ac.uk/~rh105`.

[9] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[10] UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`, 2002.

Declaring Victory in a Declarative Datacenter: Verification and Transferring Confidence

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract *Operators may appreciate and adopt declarative approaches to defining datacenters, but they will still need sophisticated tools to locate weaknesses, identify hot-spots, and catch errors. Just as usefully, they need means to transfer their confidence from one version of the system to the next. I outline some of these challenges along with our preliminary work in this direction.*

Languages If the datacenter (or datacentre, even) is the new computer [6], what is the language for configuring it? The lure of declarative languages is that they turn low-level, ad hoc languages into targets rather than sources. Just elevating the meaning of existing operations helps: Mahajan, et al. [3] point out that adopting a transactional semantics for configuration changes would have avoided a significant portion of misconfiguration errors. But declarative specifications will not eliminate errors, only move them higher up the chain of abstraction. For instance, Anderson and Scobie [1] say their current system supports "over 2000 parameters" ranging from hardware to access-control. What are the odds operators will get their specifications right?

Analyses In their study of Internet service failures, Oppenheimer, et al. [5] implicitly observe the value in three kinds of tools (the names given to these categories are ours). It is instructive to consider the impact of declarative specifications on each of these:

user-defined property preservation The first is to have high-level properties of desired behavior, and ensure that individual cofigurations match these desired global properties. While this remains a problem in "legacy" (i.e., current) systems, if individual configurations are generated directly from a declarative specification, this problem effectively disappears (or is subsumed in a proof that the generator preserves the semantics of the specification). Bravo!

smell tests The second is to check configurations against well-known properties. These would include both desirable properties (those found in systems that function well) and undesirable ones (those known to lead to faulty behavior). In a higher-level language specification, these would most probably correspond to types or other static analyses. Another win.

Let us pause momentarily to dig deeper.

Smell-tests are useful but fraught with difficulty. In a domain as messy as datacenter configuration, the clean logical niceties of "soundness" and "completeness" seem unattainable. In the absence of robust properties, the best we must hope for is that analyses will find situations frequently enough to be useful but infrequently enough to not annoy. Designing such analyses—and their corresponding user-interfaces—is daunting.

The other validation scenario is not as watertight as it seems, either. When a datacenter configuration has thousands of parameters, operators would benefit from analyses that probe their specification instead of simply accepting it at face-value. Unfortunately, if smell-tests are difficult to design, obtaining properties seems even harder. The essence of any verification process is obtaining a *redundant* statement of the system's desired behavior. In our experience in a more limited domain (access-control), when the specification language is sufficiently declarative, users have great difficulty providing a duplicate statement of behavior.

The heart of this problem is obtaining a redundant specification. If a redundant one isn't available, is there any other?

Yes, indeed, except it describes a different system: the previous version of this one. In reality, the typical operator often has a configuration that is known to "work"; now he has a new configuration that represents a desired change, either in response to new features or, sometimes, as a result of an emergency (as when a security leak is identified). As Anderson and Scobie say [op. cit.], "Small configuration changes also occur very frequently in a complex environment". What the operator really needs to know is, *Will something break if I make this change?* That is, he cares not about correctness of the new system (which is too complex to comprehend authoritatively anyway), but rather about *transferring confidence*: i.e., how to gain confidence in the new version relative to the confidence placed in the old.

This idea is loosely manifest in Oppenheimer, et al. [op. cit.]:

change-impact analysis The third analysis is generalize their suggestion to "help operators understand [...] how their changes to one component's configuration will affect the service as a whole". Here, a declarative language has direct value: information that had to be reconstucted from lower-level specifications is now expressed directly, making the analysis richer, more tractable, and have fewer false-positives and/or false-negatives.

The Margrave Tool Computer systems of a scale that can benefit from a datacenter teem with policies that govern their behavior. These include:

- A configuration model with context-sensitive rules to determine what components can, should, and can't be combined.
- Access-control policies for the data in the datacenter.
- Firewall policies to determine which sub-networks may and may not communicate with each other.
- User-defined routing policies to administer network traffic for specific needs (e.g., QoS).
- Confidentiality and integrity requirements in systems such as Security-Enhanced Linux (SELinux).
- Hypervisor rules that describe desirable and unacceptable interoperation between compartments.

These policies are, in turn, often modular and even distributed.

Many of these uses have evolved their own domain-specific notation for expressing policies, thus enabling rapid system evolution along a critical vector. Some of these apply so broadly that they have evolved into industrial standards, such as XACML for access control. In general, these languages can be treated quite uniformly using standard languages such as first-order logic.

For several years we have been building Margrave [2], an analysis suite for policy languages. Margrave began as a tool specifically for XACML, but is evolving to support languages described more generally. Margrave has two components:

1. A *verification* engine for checking policies against formal properties. This is general enough to encompass both user-defined properties and smell tests. In the access-control domain, for instance, the useful smell tests describe standard domain metaphors such as least-privilege, conflict-of-interest, and separation-of-duty. These need not necessarily hold in a given policy, but the designer may to ensure that they are violated intentionally.
2. More usefully, Margrave offers *change-impact analysis* as a property-free analysis. I will focus on this aspect of the tool in the rest of this document.

Margrave presents changes as the set of inputs that yield a difference in output and, for each input, the corresponding output change. This is an especially concrete representation that users can immediately comprehend, matching a cognitive model called *surprise-explain-reward* [7].

In practice this change can be quite large, so the user needs tools to distinguish the important from the irrelevant. Margrave therefore enables the user to probe this output. For example:

- Is the new policy equivalent to the old one?
- What are the changes when restricted to a certain type? (For instance, restricting attention to those data formerly denied but now permitted access can identify inadvertent leakage.)
- What are the "hot-spots", e.g., one rule change that altered the effect of a large percentage of outcomes?
- What roles or resources had their permissions changed?

While some of these questions are specific to the access-control domain, others are universal to policy analysis.

The trained computer scientist will, of course, recognize that these correspond to *queries* and *views* over the changes, but the operator doesn't need to understand these issues. Even more intriguingly, an operator can ask:

- Confirm that role X did not gain privileges as a result of an edit.

This is, of course, a form of *verification*. But didn't we say operators have trouble expressing properties? In fact, what we've found from talking with users is that, while people often have difficulty expressing properties of a *system*, they have much less trouble stating properties of *changes*: at the very least, they can state the framing conditions that they expect to hold of their edit.

Change-impact analysis is, therefore, a particularly engaging application of formal methods. It weds the benefits of formality—soundness, coverage of a complex state space, etc.—to an informal, inquisitive usage style. The choice of representation of output particularly helps, because operators can more readily work with concrete, extensional representations than intensional ones. This analysis modality has many different "what-if" uses:

Upgrade checking "If I make this change, what will break?"

Upgrade choosing "Will these two different ways of altering the policy yield the same result?"

Refactoring checking "I intended only to improve the structure of my policy; did I accidentally change anything?"

Some users have even adopted Margrave as an oracle for mutation testing [4].

Looking Ahead Of course, this is only a beginning. There are numerous ways in which Margrave must grow to accommodate the needs of datacenters, such as:

- Analyzing these languages independently is one thing, but combining their analysis is an entirely separate challenge, and one we have not confronted. On the other hand, perhaps we won't need to if declarative languages for datacenters take off. Therefore, I view great potential for the synergistic development of such languages with these analyses.
- These languages operate in a very stateful environment. While we have done theoretical work to extend Margrave to accommodate state, I believe the more fruitful direction would be to run the process backwards: from the policies, derive monitors on the state that warn when changes are about to occur.
- The worlds of numbers and of symbols are currently distinct. We would like to integrate the logical power of these languages with the hard numbers we obtain from dynamic analyses of behavior to provide better analysis and prediction.

Ultimately, Oppenheimer, et al. [op. cit.] point to the predominance of misconfiguration errors and say about improving operator interfaces, "This does not mean a simple GUI wrapper around existing per-component command-line configuration mechanisms—we need fundamental advances in tools to help operators understand existing system configuration and component dependencies, and how their changes to one component's configuration will affect the service as a whole". We couldn't agree more: only thus armed can the operator declare victory on the problem of managing the datacenter. Margrave is our step in this direction.

Acknowledgments

I thank Don Batory, Dan Dougherty, Kathi Fisler, Leo Meyerovich, and Michael Tschantz. The ideas described here (only the good ones, that is) are the fruit of our collaborations. I appreciate support from the US National Science Foundation, Cisco, and Google.

References

[1] P. Anderson and A. Scobie. LCFG: The next generation. In *UKUUG Winter conference*, 2002.

[2] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *International Conference on Software Engineering*, pages 196–205, May 2005.

[3] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. *ACM SIGCOMM Computer Communication Review*, 32(4):3–16, 2002.

[4] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *International World Wide Web Conference*, pages 667–676, 2007.

[5] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Usenix Symposium on Internet Technologies and Systems*, 2003.

[6] D. A. Patterson. Technical perspective: the data center is the computer. *Communications of the ACM*, 51(1):105–105, 2008.

[7] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. Harnessing curiosity to increase correctness in end-user programming. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 305–312, 2003.

Component-Oriented Programming and Datacenter Applications

Venkatesh-Prasad Ranganath
Microsoft Research India
rvprasad@microsoft.com

Abstract

Component-oriented programming enables the *development* of easily deployable and manageable applications while datacenters supports easy *deployment* and *management* of applications (via provisioning). So, why not consider component-oriented programming as the paradigm to program datacenter applications?

1. Introduction

Recently, datacenters have been gaining focus as an enterprise application platform because of (1) the decreased cost and increased speed of provisioning virtual machines and networks and (2) the catering of basic services (ranging from large scale storage to e-mail) as integral features of the runtime environment. Consequently, there will be a need for approaches to develop *datacenter applications (DAs)*; specifically, approaches that enable aggressive use of provisioning capabilities of datacenters to easily deploy and manage DAs.

For over a decade, component-based applications have been prevalent in the form of applications based on COM, .NET, EJB, and Web services. A *component-based application (CBA)* is a composition of services published by various components along with various component specific and application specific provisioning requirements/constraints. The paradigm of programming CBAs through *component development, composition creation, component deployment,* and *composition assembly* is commonly referred to as *Component-oriented programming (COP)* (2). Some of the key features of COP, such as statelessness of components, crisp separation between interface and implementation and between development and deployment, and the declarative specification of components and compositions, enable the use of *Model Driven Development (MDD)* techniques to develop CBAs.

As the features of COP complement the capabilities of datacenters, we opine that component-oriented programming is a good paradigm to program datacenter applications. In this paper, we justify our opinion by comparing the tasks (and artifacts) involved in COP and programming DAs.

2. Component-Oriented Programming (COP)

A *component* is a composable and transparent software element that provides and uses services via well-defined interfaces, supports independent versioning and deployment, and caters to multiple mutually independent simultaneous uses. These characteristics of a component enable high degree of reusability and implementation variability.[1] A *component-based applications (CBA)* is a composition of services published by various components along with various component specific and application specific provisioning requirements/constraints. Typically, any application logic specific code is encapsulated in one or more of the participating components (if needed, new components are created). A *container* defines the execution environment for components and shields them from the actual runtime environment. Typically, it hosts components, provides basic services (such as communication and persistence) to the hosted components, and manages the interaction between the components and the runtime environment.[2]

The above COP entities are constrained as follows: (1) a container can host more than one instance of a component, (2) a container can host instances of different components, (3) a container is confined to a single physical node/host, and (4) an instance of a component resides entirely in a container.

We shall now describe various tasks involved in COP.

2.1 Component Development

The task of developing a component entails *programming a component and packaging it for deployment in a container.* After identifying the collection of closely related services that constitute a component, following are the typical steps involved in developing a component.

- *Define interfaces to expose the provided services.* Most often, interfaces are defined in high level languages (such as IDL) to enable the interoperability of components independent of the underlying component implementation.
- *Program a component implementation to realize the services.* Besides the actual programming, a component implementation requires a mapping of the interface definition to implementation features that realize the interface. Typically, such a mapping is generated along with a skeletal implementation by a programming language and runtime specific interface definition language compiler. Hence, the actual activity in programming a component implementation is the addition of the business logic to the generated skeletal implementation.
- *Associate unique identifiers with the component and the exposed interfaces.* These identifiers enable the transparent publication and discovery of components and provided services (via the corresponding interfaces).
- *Identify any required external service dependences.* To ease component development, most of the basic services such as communication and persistence are externalized and provided by containers. Typically, these services are available as part of the controlled runtime sandbox provided by the containers to the hosted components.
- *Package the component implementation.* Besides the executable implementation, the package will also contain any required

[1] By *implementation variability*, we mean that an implementation X of component C can be replaced by an implementation Y of component C with almost no impact on the application.

[2] A component and its instance can be thought of as a type and a value of that type, respectively. We shall use the term *component* to mean a component type. When unambiguous, we shall abuse the term to mean a component instance.

libraries along with a *deployment descriptor* that contains the metadata required to deploy and publish the component and its interfaces in the runtime environment.

2.2 Composition Creation

As in programming an application by composing features of various libraries, a component-based application is programmed by composing services provided by various components. Hence, the composition creation tasks involves the following composition-oriented activities.

- *Identify components that provide necessary services.* This is usually specified in terms of the published component and interface identifiers. Identifier-based specification enables the runtime system to dynamically provision components that cater the requested services. Consequently, the composition is more manageable as a result of the increased flexibility in provisioning.
- *Connect the interfaces of components to realize the composition.* Such a composition is merely an inter-service data and control flow network that realizes the intent of the application. Typically, any application logic not present in any of the off-the-shelf components are bundled into application specific components and used in the composition.
- *Reason about the correctness of the composition.* Similar to programs in C and other languages, compositions are prone to structural (syntactic) and behavioral (semantic) errors. Simple compile-time techniques such as type checking are employed to weed out basic structural errors while more sophisticated techniques are employed to perform deeper reason about richer or domain specific properties (such as connectivity between services and connectivity guarantees) of the composition. Such deeper reasoning most often depends on properties of the runtime environment and can generate new constraints on the runtime environment.

  Also, such reasoning can depend on the specifications of the internals of the participating components. If such specifications are exposed, then model driven development can be employed to iteratively refine models of components and compositions and, consequently, alleviate the reasoning burdens associated with large composition and/or rich property spaces (1).
- *Identify provisioning constraints.* Non-functional features of the participating components such as availability, response time, and resource requirements cannot be programmed into components. Instead, the requirements to realize these features are identified as provisioning constraints on the runtime environment.
- *Create an assembly descriptor.* Unlike a component package, an assembly descriptor is created as the final artifact of composition creation. Typically, an assembly descriptor declaratively specifies the required components (via identifiers), the connection between various interfaces of components, and any provisioning constraints on the runtime environment either in the context of specific participating components or the composition.

2.3 Component Deployment

The deployment of a component involves the installation of the contents of a component package into a container that provides the requested basic services and publication of the identifiers of the component and its published services with the discovery service of the runtime system.

2.4 Composition Assembly

Unlike the deployment of the component, assembling a composition is more involved. Once the components involved in an assembly are located in the runtime environment, they are connected in accordance with the assembly and the application is available for use.

However, it may be impossible to locate required component-container combinations that satisfy required provisioning constraints. In such situations, a static runtime system can either abort the assembling of the composition while a dynamic runtime system can try to identify or even create appropriate containers, which satisy the provisioning constraints, to deploy the components and then retry to assemble the composition. With MDD, these situations can be avoided by reasoning about the combination of the models of the composition and the runtime environment.

3. Programming Datacenter Applications (DAs)

A *datacenter* can be perceived as a computing system composed of a large number of well connected computers. Given this view, datacenter applications can be perceived as a distributed application that executes on multiple computers.

Beyond the above perception, a datacenter can be easily configured into smaller dedicated virtual computing systems by means of virtual networking and virtualization technologies. Given this capability of a datacenter, a datacenter application can be viewed as a composition of *application fragments (AFs)* hosted on *virtual machines (VMs)* in a dedicated virtual network. Hence, the deployment of a datacenter application can be perceived as the deployment of the virtual machines that host the application fragments, setting up the virtual network, and connecting the application fragments to other application fragments and the services provided by the datacenter (in accordance to the application composition). Depending on the provisioning constraints of the application fragments, the deployment may involve the selection of existing instances of AFs, existing VMs to host new instances of AFs, powered-up computers to host corresponding VMs, or even power-up computers to host corresponding VMs. The provisioning capability of the datacenter enables this approach of deploying and managing applications.

4. Relevance of COP to Program DAs

Based on the above description of programming DAs and earlier description of COP, we can draw parallels between concepts/entities in COP and in programming DAs: component and application fragments, composition and application, and containers and virtual machines. Similar parallels can be drawn between the approach to develop and deploy a datacenter application and the COP approach to create and assemble a composition.

Given the strong correlation of entities and tasks, we opine that it may be worthwhile to explore component-oriented programming (1) as a paradigm to program datacenter applications and (2) to identify and leverage (avoid) approaches and techniques that can facilitate (inhibit) the programming of datacenter applications.

References

[1] John Hatcliff, William Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems." In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 2003.

[2] Clemens Szyperski. "Component Software: Beyond Object-Oriented Programming." Addison-Wesley, 2002.

Failure is an option

Andrew Rice Sherif Akoush Andy Hopper
Computer Laboratory, University of Cambridge
[firstname].[lastname]@cl.cam.ac.uk

Abstract

Reducing the level of redundancy in a datacentre's power and cooling infrastructure can have a big impact on reducing overall energy costs. One means to reduce this overhead is to expect failures and adapt to them rather than attempting to eliminate them at all costs. High-level specification of services within a datacentre combined with technologies such as migration might provide us with the flexibility to run closer to the wire and adapt to infrastructure failures if they occur.

We argue that Service Level Agreements (SLAs) currently act as a disincentive to exploiting this flexibility and we suggest a more customer-centric specification which gives service providers the freedom to provision adaptively and provides incentives for both parties to work towards an appropriate service level for the client's business needs.

Specifying these agreements in machine-readable form is an important challenge and would provide two benefits: reaching, and relying, on these agreements can be made easier by modelling of the expected emergent behaviour of both parties; and integrating the specification of these new agreements into service declarations will allow us to begin to develop tools for orchestrating optimal adaption when failures occur.

1. Introduction

Our computing infrastructure is composed of a wide range of devices and infrastructure. Computing hardware such as servers, network infrastructure, storage devices, client machines and terminals are supported by infrastructure systems such as Uninterruptible Power Supplies (UPSs) and cooling systems. The energy consumption in the US due to servers and associated cooling systems alone has been estimated at 1.2% of total demand (Koomey 2007). Furthermore, the manufacture of microchips is a particularly energy intensive process. If we assume a three year operational lifetime of continuous use then a third of all the energy used by a server is in the manufacture (Williams 2004). Computing is consuming an ever increasing amount of energy. However, we believe there is significant scope for improving efficiency. We see the construction of an optimal digital infrastructure as a key research challenge of the future (Hopper and Rice 2008).

Modern datacentres have experienced rapid growth in size and energy consumption and now represent a significant proportion of our computing platform. In Section 2 we show that there is particular inefficiency in these large datacentres due to the support infrastructures (such as cooling and power systems) which support the high-reliability computing services we have come to expect and rely upon. Conventionally we justify this inefficiency because any failure of this infrastructure is viewed as a disaster scenario. However, we believe a datacentre could instead adapt to failures if they occur and therefore run closer to the wire. This adaption is already possible to a large extent using technologies such as virtualisation (Section 3). However, current service-level agreements present a key obstacle. We highlight shortcomings of conventional availability guarantees and suggest an alternative way for phrasing service agreements (Section 4). Quantifying, specifying, and analysing the emergent behaviour of these agreements is a vital first step towards an adaptive datacentre.

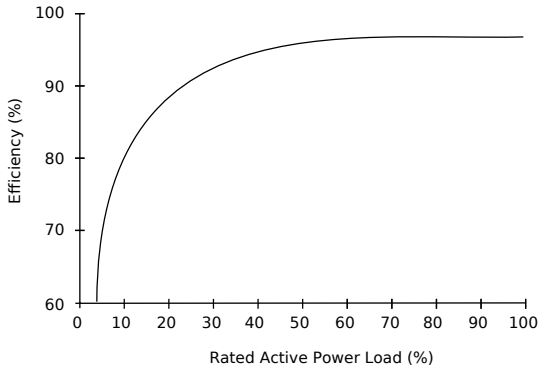


Figure 1. UPS Efficiency (high efficiency mode) (Ton and Fortenbury 2005).

2. Inefficient reliability

Datacentre reliability is often measured using the Tiering system (Turner et al.). A Tier 1 datacentre tolerates unavailability of the main electricity supply through the provision of an Uninterruptible Power Supply (UPS) and on-site generators. At the highest level, a Tier 4 datacentre provides redundancy at all points of the infrastructure through (independent) dual power supplies to every server on the machine floor. The two crucial design features of a Tier 4 facility are fault tolerance and maintainability. The facility can tolerate a failure of any single sub-system without any interruption in service and any single sub-system can be taken off-line for maintenance without interruption in service. Regularly servicing a datacentre's sub-systems provides a significant increase in reliability.

To highlight the inefficiency introduced, consider the efficiency curve of a UPS system (Figure 1). The vast majority of UPSs in use today are continually under active load and so the power distribution system is continually exposed to any inefficiency. We would ideally like to operate above 60% utilisation level due to the steep fall-off in efficiency at lower levels. This is problematic to achieve because simple over provisioning to cope with demand fluctuations or future growth can push us below this level. If one now adds a redundant live UPS for fault tolerance (a move from Tier 1 towards Tier 2) then we halve the utilisation—your UPS systems are in parallel and each must be capable of the full load if needed. If we move to a dual power system (Tier 4) then the utilisation of the UPSs halves again. The utilisation of each UPS

is now only 15% which in turn means 25% of our input power is wasted.

Achieving fault tolerance through added redundancy creates serious inefficiencies in our datacentres.

3. Adapting to faults

Rather than homogeneously provisioning capacity throughout the datacentre we could instead divide our systems up in to a number of independent units. The intention is that a fault in a single system should cause a reduction in capacity rather than a total failure. This gives the opportunity for adaption. Machines providing services with low availability guarantees might simply be switched off, diverting capacity to key services. Alternatively, services might be migrated to a smaller working set of machines perhaps compromising response time but maintaining availability.

Many key technologies with which to achieve this are already available. Recovery Oriented Computing (Patterson et al. 2002) considers how to minimise the recovery time after failure has occurred through techniques such as recursive restartability (Candia and Fox 2001) in which the system understands service dependencies. Virtualisation technologies such as Xen support the live migration of a running service from one physical machine to another (Clark et al. 2005). The migration approach can also be exploited to efficiently maintain a replica of a running virtual machine (Cully et al. 2008). This option provides redundancy at the logical level.

4. Smooth cost SLAs

Having highlighted efficiency issues and indicated the possibilities available for smooth adaption to failures we now discuss the impact this has on Service Level Agreements (SLAs). In light of their current growth and popularity we consider a web-services or Remote Procedure Call application in which a user makes an asynchronous request to a service which subsequently replies with a response.

We consider three principle actors:

- **The service provider** offers a computing platform for executing a service;
- **The users** make requests to the service and expects responses;
- **The client** contracts a service provider to provide a particular service for users.

The key goal of an SLA is to permit the client to specify the service level which should be provided to users. This is commonly an economic tradeoff in which the client chooses the right balance between the business benefit of a high service level and the high cost of sourcing it from a service provider.

As a simple example we consider the scenario in which a service provider is contracted by a client to host a website which will be selling tickets for an event. A conventional SLA might include:

- an availability guarantee;
- the maximum number of simultaneous users the site will support; and
- details of financial penalties and remedial actions for violation of the agreement.

This form of agreement would seem to suit the service provider because it makes provisioning straightforward but it does not suit the client. We imagine that the client might be happier specifying that:

- for each user who successfully requests the website the site the service provider gets paid some amount;
- for each user who requests the site and it fails to respond appropriately the service provider is fined some amount.

This model is closer to the cost analysis done by the business—what is each user worth to us? and what is the cost of not providing the service when they need it?

An agreement in this form has a number of benefits. Firstly, it aligns economic incentives with expected behaviour. For example, under the original agreement, if the site becomes overloaded the service provider simply incurs the penalty clause. In the second agreement the service provider is given an incentive to provide a good service to as many users as possible and no service to the remainder rather than almost no service to all users. Secondly, it allows the service provider to provision more efficiently. The cost of providing a certain level of fault tolerance in the datacentre can be compared with the cost of a particular adaption strategy when failure occurs.

Assigning a single price to a serviced request and a single fine for an unserviced request might be insufficient to express the client's needs. Clients might wish to penalise repeated failure for a particular user with a higher penalty. Both parties might with to limit their financial exposure either by capping the request rate or by varying the price curves depending on the number of requests per second.

5. Research Directions

We are seeking a new way of describing Service Level Agreements between clients and service providers. Changing the form of these agreements should give an incentive for service providers to reduce overheads in the infrastructure. More closely meeting the business needs of the client also provides opportunity for efficiency improvements—if the client no longer pays for a service level which he doesn't need, the service provider need not provide a service level which is unnecessary.

5.1 Specifying agreements

Languages such as Baltic (Bhargavan et al. 2007) can express the interactions between services and make static checks for correctness. Extending such a declaration to incorporate an SLA would mean that this information can then be used to inform the decisions of scheduling software to most appropriately allocate the resources available.

Research into ontologies and the Semantic Web attempt to codify real-world relationships and hierarchies. These approaches might be applied to describing a service levels for different applications.

5.2 Emergent behaviour

Integrating numerous costs and rewards into an agreement may well create complex emergent behaviour. Parties might wish to model check an agreement to determine expected outcomes under different traffic and fault models.

In the simple example agreement mentioned above a service provider might decide to run no services at all overnight due to the small number of expected requests. A client can respond to this strategy by generating synthetic traffic and thus imposing an additional fine on the service provider. However, the service provider can respond in turn by running the service overnight and allowing the synthetic traffic to run up a bill which does not correlate with any expected business revenue.

A model often used in game theory for the expected behaviour of the service provider and the client is that they will always act to maximise their own revenue. In the example above the service provider would not consider switching off the service overnight if the client's response will cause a loss. However, for this reasoning to be valid the client must actually have a possible response available to it.

6. Conclusion

Much research has focussed on reducing the energy consumption of computing itself. However, there are significant reductions to be found when considering the infrastructure which supports computing. Technologies such as migration and automated recovery strategies mean that we could build adaptive datacentres which run closer to the wire and provision their workload according to the available resources.

Unfortunately, current SLAs do not provide much freedom or incentive to do this. We have argued for a more client-centric specification of service level. This in turn gives service providers more freedom to minimize their overheads and helps to ensure that the level of service provided more closely matches the level required by the client's business.

We wish to investigate how these SLAs might be incorporated in languages currently used for specifying the interaction of services within a datacentre and how this information might be used by a scheduling service to optimally provision infrastructure and computing resource.

References

Karthikeyan Bhargavan, Andrew D. Gordon, and Iman Narasamdya. Service combinators for farming virtual machines. Technical Report MSR-TR-2007-165, Microsoft Research, 2007.

George Candia and Armando Fox. Designing for high availability and measurability. In *Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability*, 2001.

Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, 2005.

Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, , and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

Andy Hopper and Andrew Rice. Computing for the future of the planet. *Philosophical Transactions of the Royal Society A*, To Appear, 2008.

Jonathan G. Koomey. Estimating total power consumption by servers in the U.S. and the world. February 2007.

David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kcman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, UC Berkeley, March 2002.

My Ton and Brian Fortenbury. High performance buildings: Data centers uninterruptible power supplies (UPS). Technical report, Lawrence Berkeley National Laboratory, 2005.

W. Pitt Turner, John H. Seader, and Kenneth G. Brill. Tier classifications define site infrastructure performance. Technical report, The Uptime Institute.

Eric Williams. Energy intensity of computer manufacturing: Hybrid assessment combining process and economic input–output methods. *Environmental Science and Technology*, 38(22):6166–6174, 2004.

Formal-logical models of norms and institutions

Marek Sergot
Department of Computing, Imperial College London
mjs@doc.ic.ac.uk

Abstract

This is a summary of ongoing work in the formal-logical modelling of norms, institutions and (virtual) organisations, with an indication of how it might relate to the topics of the workshop. This is a partial account. There is extensive activity elsewhere in the Department of Computing at Imperial College on related topics, such as policy languages, autonomous systems, workflow models, and automated dispute resolution mechanisms. These activities are not covered here.

1. Introduction

I have a long standing interest in the formal-logical modelling of norms, institutions, and (virtual) organisations. We want to be able to represent and reason about patterns of agent interaction in organised (human and computational) societies, and the rules and policies governing such interactions. Such rules are ordinarily formulated in terms of agents' obligations, permissions, and rights, their responsibilities in regard to the achievement of specific goals, and their powers to initiate changes, to authorise, and to delegate. The term 'agents' is intended to cover both human and artificial (computer) agents, deliberative agents with perceptual and reasoning capabilities, simple reactive agents with no reasoning capabilities, and systems composed of simple computational units in interaction.

It cannot always be assumed that agents will behave as they are supposed to behave. Agents may fail to comply with norms governing their behaviour for several reasons. They might do so deliberately, in open agent systems or other competitive settings, or unintentionally, in unreliable environments because of factors beyond their control, or even inadvertently because of the actions of other agents acting independently. In addition to analysing system properties that hold if specifications/norms are followed correctly, it is also necessary to predict, test, and verify the properties that hold if norms are violated, and to test the effectiveness of introducing proposed control, enforcement, and recovery mechanisms.

This is a summary of recent and ongoing activities in this general area. These include theoretical contributions to the logic of norms and to formal theories of complex normative concepts such as duty, right, delegation, and authority, the computer representation of laws, rules, regulations, technical standards, and codes of practice of various kinds, and applications in areas such as multi-agent systems, contracts and service level agreements (Daskalopulu and Sergot 1995, 1997; Farrell et al. 2005), trading procedures (Daskalopulu and Sergot 2002), workflows and business processes (Farrell 2008), and computer security (Sadighi Firozabadi et al. 2002).

This is a partial account. The references to published works are incomplete; I have merely picked out some representative examples, and there are very few references to related work by other groups. These may be found in the papers that are cited. I should also make it clear that there is extensive activity elsewhere in the Department of Computing at Imperial College on related topics, such as policy languages, autonomous systems, workflow models, and automated negotiation and dispute resolution mechanisms. These activities are not covered here.

2. Obligation and permission

The principal formal tool for the analysis of normative systems is deontic logic, the logic of expressions pertaining to obligation and permission. The field has its origins in philosophical logic, applied modal logic, and ethical and legal theory. In recent years, interest has also grown in the value of deontic logic as a formal tool in a range of problems in computer science. There are numerous references in the literature; the topic is the subject of a series of specialist conferences (*DEON*) held biannually since 1991. There are now also specialist workshops concerned with norm-governed multi-agent systems specifically.

There are many systems of deontic logic. Most take their point of departure in the inadequacies of so-called Standard Deontic Logic, a modal logic of type *KD* in the standard classification. Semantically, Standard Deontic Logic is based on a simple partitioning of possible worlds or states into 'ideal' and 'sub-ideal' (terminology varies). Other forms of deontic logic are based on various conditional logics, temporal logics, logics of action, logics of agency, preference structures, speech acts, and other semantical devices.

In view of its origins, one obvious class of applications of deontic logic is the representation of rules and regulations for the construction of computer programs that can determine or advise on the consequences of some real or hypothetical state of affairs, or in systems that provide support for the preparation and drafting of new sets of regulations. Such applications are not limited to the domain of law. One application we have looked at in the past, for example, concerned chemotherapy protocols in the treatment of cancer (Hammond and Sergot 1996). These have essentially the same character.

Deontic logic is also used as a tool in the analysis and specification of normative aspects of computer systems, such as access control to databases, policy management in distributed systems, authorisation mechanisms, privacy maintenance, and digital rights management.

Another general class of applications relies on the observation that computer systems themselves, in particular systems consisting of human and computer agents in interaction, can sometimes be modelled productively, for certain purposes, as instances of normative systems. Several groups of researchers have employed deontic logic of one form or another in order to deal with elements of fault tolerance and the specification of corrective actions that are to be performed when the computer system or one of its components fails to behave as it should. The term 'biddable component' has recently begun to be used in Software Engineering, and one can find refer-

ences to notions such *inadvertent* and *malicious* violation. Further distinctions can be made (see below).

3. Complex normative concepts

Deontic logic, in one form or another, is one ingredient in formal theories that attempt to give a precise characterisation of complex normative concepts and their properties. These complex normative concepts include concepts such as duty, right, privilege, immunity (the 'Hohfeldian' concepts) which feature prominently (but very imprecisely) in everyday legal discourse, as well as concepts commonly used informally when describing organisational structures, such as responsibility, accountability, entitlement, authority, authorisation, and delegation.

The Hohfeldian concepts were so named in recognition of the pioneering work of the legal scholar, Wesley Hohfeld, who at the beginning of the last century tried to introduce some precision and regularity into the language used in legal discourse. Hohfeld's writings have been highly influential both in the theory and practice of law.

The Kanger-Lindahl theories of 'normative positions' were long regarded as the most developed attempt at a formal characterisation of the Hohfeldian concepts. They combine a simple form of deontic logic with a logic of agency, that is, a logic dealing with expressions of the form 'agent x brings it about that such-and-such A is the case', or 'agent x is responsible for its being the case that A', or 'the actions of agent x are the cause of its being the case that A'. The study of such logics has a very long tradition in philosophical logic. In computer science the best known examples are perhaps the 'stit' ('seeing to it that') family of logics. (See e.g. (Horty 2001) for an extended account.)

In (Sergot 2001) we showed that the Kanger-Lindahl theories are a special case of a more general theory, and presented a new treatment, together with a set of inference methods and associated computational tools for applying the theory to practical problems. (Jones and Sergot 1992, 1993) discuss how such methods can be applied to an example in the computer security literature that was concerned with developing a precise specification of access rights (to medical data of patients in a mental hospital).

An essential ingredient missing from the the Kanger-Lindahl framework is the concept Hohfeld called '(legal) power'. It is also referred to as 'legal capacity' or 'competence' in legal theory. It is the characteristic feature of all institutions and organisations whereby a designated agent, often when acting in a particular role, is empowered by that institution to create or modify a relation that has recognised significance within that institution—such as when we say that a registrar is empowered by the state to create a state of marriage between a couple, or a judge declares a verdict in favour of the plaintiff in a court case, or a department manager signs a purchase order and thereby commits his or her employer to the purchase, or when a project manager assigns one of his or her team to a particular role. It is clear that institutional power (or capacity/competence) in this sense is not an exclusively legal phenomenon but a feature of all institutions and organisations.

Institutional power and the associated 'norms of competence' have been the subject of extensive study. In (Jones and Sergot 1996) we argued that institutional power is best understood in terms of more general statements, often called 'constitutive norms', of the form 'A counts, in institution I, as B'. On this account, agent x has power to create an institutional relation R means that the performance of a specified action or procedure by x counts as a means by which x sees to it that the relation R holds in the institution. The logic of 'counts as' conditionals is receiving much current attention.

It is instructive to look at the sort of terms that are ordinarily used to describe organisational structures from this perspective. The term 'authorised' for instance can have a wide variety of meanings, from permissible (not prohibited), explicitly permitted by some competent authority, empowered (in the Hohfeldian sense), or entitled (in the sense of a right of some kind), or some combination of these. The term 'responsibility' is used in at least two different ways. x 'is responsible for' A can mean that it is obligatory that x sees to it that A, as when we say that x is responsible for delivering a certain product to a client by a certain time. x 'is responsible for' A can also mean that x's action are the cause of its being A, as when we say that x is responsible for a product not being delivered to the client on time.

One concept that I think deserves special attention is that of entitlement. Until recently the emphasis in access control, for example, has been on the prevention of unauthorised access. In modern settings, one must also consider the other side of the coin, that an access control mechanism must also provide access to a resource when it is requested by a suitably authorised party. We might say that when x is entitled to a resource R, the agent y which controls access to R is obliged to grant access to R when requested, in some designated manner, by x: in Hohfeldian terminology that x has (institutional) power to create an obligation on y to give x access to resource R. In (Sadighi Firozabadi et al. 2004) we presented a framework for relating policies on entitlement to resources in virtual organisations to the local policies that govern individual member parties. I believe that various conceptions of entitlement are worth particular study.

It should not be assumed that the required representation and reasoning tools are necessarily too complex for practical application. For example, the formal theory of rights relations developed in (Sergot 2001) presents accompanying inference methods that are easily implemented in software tools to support practical application of the methods.

4. Action formalisms

Generally, the logic of norms and the logic of action have often been studied together. Many authors have argued that an adequate theory of norms must be underpinned by a precise theory of action (by which is often meant agency). Transition based treatments of action are not common in the philosophical logic literature.

In applications we use both the event calculus and the action language $\mathcal{C}+$ (Giunchiglia et al. 2004). The event calculus is a way of representing the effects of actions in a logic programming framework. Its main advantage is that it is easily and efficiently implemented for certain computational tasks, specifically for computing states from recorded event narratives. It is widely used, at Imperial College and elsewhere, for a wide variety of applications. (Farrell et al. 2005) used it in a system for tracking the evolving state of a contract as it is enacted. The contract state consists of the obligations, permissions, and powers ('capacities', 'competences') of the contracting parties and the values of various other state variables. The system monitors a stream of recorded event occurrences and maintains the current state of the contract as well as giving access to all past states if required, for instance for auditing purposes. Farrell's system uses a Java implementation of the event calculus with an XML encoding of contract terms and event occurrences. The system has been applied to a range of examples, mostly concerning Service Level Agreements in the context of Utility Computing.

A second example is the access control system described in (Sadighi Firozabadi et al. 2002). That was originally developed from a theoretical study of power/authority and delegation. Simplified forms of these concepts were encoded in a modified form of the event calculus, and subsequently re-coded in a low-level programming language. This system is now available as a commercial product from the Swedish Institute of Computer Science (SICS).

The language $\mathcal{C}+$ (Giunchiglia et al. 2004) is a (comparatively recent) member of the family of formalisms called 'causal action languages' in the AI literature. It is a formalism for specifying and reasoning about the effects of actions, default persistence ('inertia') of facts over time, non-deterministic and concurrent actions, and indirect effects of actions ('ramifications'). Although it is presented in terms of a general-purpose non-monotonic formalism called 'causal theories', it can also be regarded as a language for defining transition systems of a certain kind. Several implementations are available, notably the 'Causal Calculator' CCALC developed at the University of Texas[1] and made available as a means of performing a variety of computational tasks using $\mathcal{C}+$.

An advantage of the $\mathcal{C}+$ language, as we see it, is that is has an explicit semantics in terms of labelled transition systems, so providing a bridge between AI formalisms and a wide range of existing tools and techniques in other areas of computer science, such as model checkers in particular. We have our own implementation, ICCALC[2], which retains the core functionality of CCALC, and its core implementation, and adds a number of other features, such as the ability to pass (a symbolic representation) of the transition system defined to standard CTL model checking systems (specifically NuSMV) (Craven 2006). ICCALC also supports a number of extended forms of $\mathcal{C}+$. In particular $n\mathcal{C}+$ (Sergot and Craven 2006; Craven and Sergot 2007) is an extended form of $\mathcal{C}+$ designed specifically for representing simple normative and institutional concepts. The first extension is a means of expressing 'counts as' relations between actions, also referred to sometimes as 'conventional generation' of actions. The second is a way of specifying the permitted (acceptable, legal) states of a transition system and its permitted (acceptable, legal) transitions. Normative system properties expressed in temporal logics such as CTL can then be verified by means of standard model checking techniques (in ICCALC, using NuSMV) on a transition system defined using the $n\mathcal{C}+$ language.

5. 'Open agent societies'

A particular kind of multi-agent system is one where the members are developed by different parties and have conflicting goals. Examples are negotiation protocols, dispute resolution protocols, rules of procedure, electronic marketplaces, and e-institutions; it has been argued that many practical applications in the future will be realised in terms of 'open agent systems' of this sort.

We have been developing methods for constructing executable specifications of 'open agent societies' (Artikis et al. 2002, 2008, 2007). The specification consists of four components: (1) the possible behaviours, causal relations, and physical capabilities of the member agents; (2) constitutive norms defining institutional concepts and relations (such as, in a contract setting, 'designated carrier', 'mode of delivery', 'supervising engineer', and so on) and the powers ('capacities', 'competences') of agents to create new instances of institutional relations or to effect changes; (3) permissions, prohibitions and obligations of the agents; and (4) sanctions, enforcement policies, and recovery procedures that deal with the performance of forbidden actions and non-compliance with obligations. We have used both the event calculus and the action languages $\mathcal{C}+$ and $n\mathcal{C}+$ to execute such specifications. Applications have included negotiation and resource allocation protocols, automated dispute resolution procedures, and voting mechanisms.

[1] `http://www.cs.utexas.edu/users/tag/cc`

[2] `http://www.doc.ic.ac.uk/~rac101/iccalc/`

6. Contracts, agreements, service compositions

Our group, led by my colleague Alessio Lomuscio, is participating in an collaborative EU project concerned with contracts and contract-based web service compositions. Our main technical aim is to integrate and then build on three strands of previous work: the event calculus based work on tracking evolving contract states mentioned above, the executable specifications of 'open agent societies' just described, and a third strand that has been developing model checking techniques for the verification of normative and temporal epistemic properties of multi-agent systems (Lomuscio and Raimondi 2006). This third strand is based on an extended form of the 'interpreted systems' framework (Fagin et al. 1995) used for analysing epistemic properties in multi-agent systems and distributed computer systems. 'Deontic interpreted systems' (Lomuscio and Sergot 2003) add a means of analysing epistemic system properties when components of the system fail to function according to their specification. We are exploring the application of these techniques to the formal modelling of contract-based web service compositions. The model checking of web service behaviour has remained limited to verifying simple termination, safety, and liveness properties. There is an argument that aspects of system composition can be analysed by considering the knowledge acquired by services during their interactions, and that the specification and verification of these epistemic properties will facilitate the analysis of a number of properties of the system, including Service Level Agreements and contracts which define the allowed (acceptable, legal) behaviours of the parties in the composition. A small example is presented in (Lomuscio et al. 2007). We use a specialised system description language (ISPL) paired with a symbolic model checker (MCMAS) optimised for the verification of temporal epistemic properties (Lomuscio and Raimondi 2006). This formalism can be seen as adding to the formalisms described above the concepts of an agent's local state and its local protocol/behaviour. Our aim is to integrate these three strands of work. We want to support both run-time monitoring of contract execution and implementation of the enabling infrastructure (e.g., web services), and off-line (design-time) verification of contract and system properties.

In regard to representation of contracts and agreements generally, my own view is that the emphasis on obligations and sanctions that pre-occupies most of the current published work is misplaced. Obligations and sanctions are relevant but are often comparatively peripheral. Our emphasis is rather on representing the terms under which a delivered product or service is to be regarded as meeting its agreed specification, and the powers and procedures by means of which changes to an agreement are requested, approved, and effected.

7. Agency and collective agency

Much of my own recent work has been on a new logic of agency, focussing on expressions of the form 'agent x brings it about that A', or 'agent x is responsible, perhaps unwittingly, for its being the case that A'. As has often been observed, the expression 'seeing to it that A' usually has a connotation of deliberate, intentional action. 'Bringing it about that A' does not have that connotation, and can be applied equally well to the unintentional as well as intentional (intended) consequences of one's actions. The agency modalities are of this latter 'brings it about' kind. The novel feature is the switch of attention away from talking about an agent's bringing it about that a certain state of affairs exists to talking about an agent's bringing it about that a transition has a certain property. Although the possibility of combining a logic of agency with a transition-based treatment of action has been mentioned from time to time, a detailed development has not been done before (to my knowledge).

This study was initially motivated by issues arising in the norm-governed regulation of multi-agent systems in computer science. An idea that has been gaining popularity in that field is that, in some cases, interactions among multiple, independently acting agents can best be regulated and managed by the use of norms. The term 'social laws' has also been used in this connection, usually with reference to 'artificial social systems'. A 'social law' has been described as a set of obligations and prohibitions on agents' actions, that, if respected, allow multiple, independently acting agents to co-exist in a shared environment.

We want to be able to say in this setting (and others) that in a system transition representing many concurrent actions by multiple agents and possibly the environment, it is specifically one agent's actions rather than another's that are in compliance or non-compliance with norms governing its behaviour, or more generally, that they are the cause of something being the case. We have been able to identify and characterise several different categories of non-compliant behaviour, for example, distinguishing between various forms of unavoidable or inadvertent non-compliance, behaviour where an agent does 'the best that it can' to comply with its individual norms but nevertheless fails to do so because of actions of other agents, and behaviour where an agent could have complied with its individual norms but did not. The aim, amongst other things, is to be able to investigate what kind of system properties emerge if we assume, for instance, that all agents of a certain class will do the best that they can to comply with their individual norms, or never act in such a way that they make non-compliance unavoidable for others. (Sergot 2008) shows how this can be done on some (small) examples.

The account generalises rather naturally to what we term 'unwitting collective agency'. It is clear that genuine collective or joint action involves a very wide range of issues, including joint intention, communication between agents, awareness of another agent's capabilities and intentions, and many others. We want to factor out all such considerations, and for the time being investigate what can be said about individual or collective agency when all such considerations are ignored. From the practical point of view, there is clearly a wide class of applications for multi-agent systems composed of agents with reasoning and deliberative capabilities. There is an even wider class of applications if we consider also simple 'lightweight' agents with no reasoning capabilities, or systems composed of simple computational units in interaction. We want to be able to consider this wider class of applications too.

References

A. Artikis, J. Pitt, and M. J. Sergot. Animated specification of computational societies. In C. Castelfranchi and W. L. Johnson, editors, *Proc. 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'02), Bologna*, pages 1053–1062. ACM Press, July 2002.

Alexander Artikis, Marek Sergot, and Jeremy Pitt. An executable specification of a formal argumentation protocol. *Artificial Intelligence*, 171 (10–15):776–804, July–October 2007.

Alexander Artikis, Marek Sergot, and Jeremy Pitt. Specifying Norm-governed Computational Societies. *ACM Transactions on Computational Logic*, 2008. To appear. Available at `http://tocl.acm.org/accepted/304artikis.pdf`.

Robert Craven. *Execution Mechanisms for the Action Language $\mathcal{C}+$*. PhD thesis, University of London, September 2006.

Robert Craven and Marek Sergot. Agent strands in the action language nC+. *Journal of Applied Logic*, 2007. To appear.

A. K. Daskalopulu and M. J. Sergot. A constraint-driven system for contract assembly. In *Proc. Fifth International Conference on Artificial Intelligence and Law (ICAIL'95), University of Maryland*, pages 62–70. ACM Press, 1995.

A. K. Daskalopulu and M. J. Sergot. The representation of legal contracts. *AI and Society*, 11:6–17, 1997.

A.K. Daskalopulu and M. J. Sergot. Computational aspects of the FLBC framework. *Decision Support Systems*, 33(3):267–290, July 2002.

R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, 1995. ISBN 0-262-06162-7.

Andrew D. H. Farrell. *Modelling Contracts and Workflows for Verification and Enactment*. PhD thesis, University of London, March 2008.

Andrew D. H. Farrell, Marek Sergot, Mathias Sallé, and Claudio Bartolini. Using the event calculus for tracking the normative state of contracts. *International Journal of Cooperative Information Systems*, 14(2–3):99–129, June & September 2005.

Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

P. Hammond and M. J. Sergot. Computer support for protocol-based treatment of cancer. *Journal of Logic Programming*, 26(2):93–111, 1996.

J. F. Horty. *Agency and Deontic Logic*. Oxford University Press, 2001.

A. J. I. Jones and M. J. Sergot. On the Characterisation of Law and Computer Systems: The Normative Systems Perspective. In J-J. Meyer and R. J. Wieringa, editors, *Deontic Logic in Computer Science: Normative System Specification*, chapter 12, pages 275–307. John Wiley & Sons, Chichester, England, 1993.

A. J. I. Jones and M. J. Sergot. Formal specification of security requirements using the Theory of Normative Positions. In Y. Deswarte, G. Eizenberg, and J.-J. Quisquater, editors, *Computer Security—ESORICS 92*, LNCS 648, pages 103–121. Springer-Verlag, Berlin Heidelberg, 1992.

A. J. I. Jones and M. J. Sergot. A formal characterisation of institutionalised power. *Journal of the IGPL*, 4(3):429–445, 1996. Reprinted in E. G. Valdés, W. Krawietz, G. H. von Wright, and R. Zimmerling, editors, *Normative Systems in Legal and Moral Theory. Festschrift for Carlos E. Alchourrón and Eugenio Bulygin*, pages 349–367. Duncker & Humboldt, Berlin, 1997.

A. Lomuscio and F. Raimondi. MCMAS: a tool for verifying multi-agent systems. In *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*. Springer-Verlag, 2006.

A. Lomuscio and M. J. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, October 2003.

A. Lomuscio, H. Qu, M. Sergot, and M. Solanki. Verifying temporal and epistemic properties of web service compositions. In *Proc. 5th International Conference on Service-Oriented Computing (ICSOC'07), Vienna, September 2007*, LNCS 4749, pages 456–461. Springer Verlag, 2007.

B. Sadighi Firozabadi, M. J. Sergot, and O. Bandemann. Using authority certificates to create management structures. In Bruce Christianson, Bruno Crispo, James A. Malcolm, and Michael Roe, editors, *Security Protocols. 9th International Workshop, Cambridge, April 2001*, LNCS 2467, pages 134–145. Springer, 2002.

B. Sadighi Firozabadi, M. J. Sergot, A. Squicciarini, and E. Bertino. A framework for contractual resource sharing in coalitions. In *Proc. 5th IEEE Workshop on Policies for Distributed Systems and Networks (Policy'04), IBM Yorktown Heights, June 2004*, pages 117–126. IEEE Computer Society, 2004.

M. J. Sergot. A computational theory of normative positions. *ACM Transactions on Computational Logic*, 2(4):581–622, October 2001.

Marek Sergot. Action and agency in norm-governed multi-agent systems. In A. Artikis, G. O'Hare, K. Stathis, and G. Vouros, editors, *8th Annual International Workshop "Engineering Societies in the Agents World" (ESAW'07), Athens, October 2007.*, LNAI 4995. Springer, 2008.

Marek Sergot and Robert Craven. The deontic component of action language nC+. In L. Goble and J-J. Ch. Meyer, editors, *Deontic Logic and Artificial Normative Systems. Proc. 8th International Workshop on Deontic Logic in Computer Science (DEON'06), Utrecht, July 2006*, LNAI 4048, pages 222–237. Springer Verlag, 2006.

Verifying Overlay Networks for Relocatable Computations

(or: Nomadic Pict, relocated)

Peter Sewell
University of Cambridge
http://www.cl.cam.ac.uk/users/pes20

Paweł T. Wojciechowski
Poznań University of Technology
http://www.cs.put.poznan.pl/pawelw/

Abstract

In the late 1990s we developed a calculus, Nomadic Pict, in which to express and verify overlay networks, for reliable communication between relocatable computations. Then, efficient system support for relocation was rare, and the calculus was reified in a prototype high-level programming language. Now, relocatable computation is a pervasive reality, though at the level of virtual machines rather than high-level languages. One can ask whether the semantic theory and algorithms developed for Nomadic Pict can be applied (or adapted) to infrastructure for communication between these virtual machines.

1. Introduction

In the 1990s, with the promise of commodity computation dimly visible on the horizon, there was considerable interest in *mobile computation*: systems in which running computations could be moved from one physical machine to another. Many motivating scenarios were put forward, including mangagement of the physical machines (e.g. moving a server away from a machine which needed rebooting), management of network resource (e.g. moving computations 'close' to their communication partners), and managing intermittently connected devices (e.g. moving computations to and from PDAs or laptops). Much of this work was in terms of high-level programming language support for moving computations, and some drew together research on functional programming languages such as Standard ML (Milner et al. 1997), and process calculi such as the π-calculus (Milner et al. 1992). Several languages, including Obliq (Cardelli 1995), Facile (Thomsen et al. 1996), and the Distributed Join Calculus (Fournet et al. 1996), supported not just mobility but also *location-independent* communication between these mobile computations, with distributed infrastructure in the language implementation, which today one might term an overlay network, to reliably deliver messages irrespective of any relocations.

It was clear then that the design of these overlay networks was a challenging problem:

- The distributed algorithms involved are delicate and error-prone, highly concurrent, and with potential races between message delivery and relocation of computations; they are hard to reason about informally.
- The languages cited above have particular algorithms hard-coded into their implementations, but in general the choice of an infrastructure algorithm must be somewhat application-specific: any given overlay algorithm will only have satisfactory performance for some range of migration and communication behaviour; it should be matched to the expected properties (and robustness and security demands) of applications, and of the underlying network.

To address this we developed a small calculus, Nomadic Pict, to permit such algorithms to be described concisely and with mathematical precision (Sewell et al. 1998, 1999; Wojciechowski and Sewell 1999, 2000; Wojciechowski 2001, 2006). The basic primitives included fine-grained concurrency and asynchronous message passing, taken from the π-calculus, together with constructs to create a new named computation (potentially multithreaded), to relocate such a computation from one machine to another, and to send asynchronous messages between these computations. All these are simple to realize, with at most one inter-machine communication required for each transition of the calculus. An overlay network for reliable location-independent communication could then be expressed as a translation of an extended calculus, with that added, into the basic calculus. We implemented the eponymous programming language based on the Nomadic Pict calculi, and experimented with a variety of overlay networks, variously centralised or P2P, with more or less caching, replication, and so on. Our experience was that the level of abstraction of these calculi was a good fit, making it relatively easy to design and understand the overlay network algorithms. Moreover, together with Unyapoth, we developed semantic techniques to support *verification* of the correctness of these algorithms (Unyapoth and Sewell 2001). The key issue here was observational congruence reasoning in the presence of assumptions under which particular computations could be guaranteed (temporarily) *not* to relocate, thus controlling the races between message delivery and relocation.

Ten years later, in 2008, relocatable computation is finally becoming a commonplace reality. This is happening not at the programming-language level we envisioned before, but via checkpointing and movement of virtual machine images, which provides a pervasive (and narrow) API at which to cut the software stack. However, when it comes to looking at communication between virtual machines, this may not be a significant difference. In this position paper we therefore ask whether the Nomadic Pict abstractions could be directly applied, or be adapted, to solve problems in this new setting.

2. The Nomadic π Calculi, Relocated

In this section we recall the Nomadic π calculi, shifting terminology to match the hypothesised virtual machine application.

The main entities are *sites* s and *virtual machines* a. Sites represent physical machines; each site has a unique name. Virtual machines are units of running computation. Each has a unique name and a body consisting of some Nomadic Pict concurrent process P (modelling whatever multi-threaded programs are running in that virtual machine); at any moment it is located at a particular site. For simplicity we do not model nested virtual machines.

A virtual machine can *relocate*, at any point in time, to any other site (identified by name), new virtual machines can be *created* (with the system synthesising a new unique name, bound to a lexically

2008/4/17

P	::=	$\texttt{create}^Z\ a = P\ \texttt{in}\ Q$	spawn VM a with body P, on local site
	|	$\texttt{relocate to}\ s \rightarrow P$	relocate this VM to site s
	|	$\texttt{iflocal}\ \langle a\rangle c!v\ \texttt{then}\ P\ \texttt{else}\ Q$	send $c!v$ to VM a if it is co-located here, and run P, otherwise run Q
	|	$\langle a\rangle c!v$ (sugar)	send $c!v$ to VM a if it is co-located here
	|	$\langle a@s\rangle c!v$ (sugar)	send $c!v$ to VM a if it is at site s
	|	$\mathbf{0}$	empty process
	|	$P \mid Q$	parallel composition of processes P and Q
	|	$\texttt{new}\ c : {}^I T\ \texttt{in}\ P$	declare a new channel c
	|	$c!v$	output of v on channel c in current VM
	|	$c?p \rightarrow P$	input on channel c in current VM
	|	$*c?p \rightarrow P$	replicated input
	|	$\texttt{if}\ v\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional
	|	$\texttt{let}\ p = ev\ \texttt{in}\ P$	local declaration

Figure 1. Nomadic π-calculus: Syntax

$$
\begin{array}{lcll}
\Gamma \Vdash @_a \texttt{create}^Z\ b = P\ \texttt{in}\ Q & \rightarrow & \Gamma \Vdash \texttt{new}\ b : \texttt{VM}^Z @s\ \texttt{in}\ (@_b P \mid @_a Q) & \text{if } \Gamma \vdash a@s \\
\Gamma \Vdash @_a \texttt{relocate to}\ s \rightarrow P & \rightarrow & (\Gamma \oplus a \mapsto s) \Vdash @_a P & \\
\Gamma \Vdash @_a\ (c!v | c?p \rightarrow P) & \rightarrow & \Gamma \Vdash @_a \mathsf{match}(p, v) P & \\
\Gamma \Vdash @_a \texttt{iflocal}\ \langle b\rangle c!v\ \texttt{then}\ P\ \texttt{else}\ Q & \rightarrow & \Gamma \Vdash @_a P \mid @_b c!v & \text{if } \Gamma \vdash a@s \wedge \Gamma \vdash b@s \\
\Gamma \Vdash @_a \texttt{iflocal}\ \langle b\rangle c!v\ \texttt{then}\ P\ \texttt{else}\ Q & \rightarrow & \Gamma \Vdash @_a Q & \text{if } \Gamma \vdash a@s \wedge \Gamma \vdash b@s' \wedge s \neq s'
\end{array}
$$

Figure 2. Nomadic π-calculus: Selected Reduction Rules

scoped identifier) and virtual machines can *interact* by sending messages to each other.

A key point in the design of the low-level calculus is to make it easy to understand the behaviour of the system in the presence of partial failure. To do so, we chose interaction primitives that can be directly implemented above the real-world network (the Sockets API and TCP or UDP), without requiring a sophisticated distributed infrastructure. Our guiding principle is that each reduction step of the low-level calculus should be implementable using at most one inter-site asynchronous communication. [1]

To provide an expressive language for local computation within each virtual machine body, but keep the calculus concise, we include the constructs of a standard asynchronous π-calculus. The Nomadic Pict concurrent process of a virtual machine body can involve parallel composition, new channel creation, and asynchronous messaging on those channels within the virtual machine.

In the rest of this section we give the syntax of processes, and the key points of their reduction semantics.

2.1 Processes of the Low-Level Calculus

The syntax of a low-level core calculus is given in Fig. 1, grouped into the three virtual machine primitives, two useful communication forms that are expressible as syntactic sugar, and the local asynchronous pi-calculus. Executing the construct $\texttt{create}^Z\ b = P\ \texttt{in}\ Q$ spawns a new virtual machine, with body P, on the current site. After the creation, Q commences execution, in parallel with the rest of of the body of the spawning virtual machine. The new virtual machine has a unique name which may be referred to with b, both in its body and in the spawning virtual machine (b is binding in P and Q). The Z is a mobility capability, either $\texttt{s}$, requiring this virtual machine to be static, or $\texttt{m}$, allowing it to be mobile.

Virtual machines can relocate to named sites: the execution of $\texttt{relocate to}\ s \rightarrow P$ as part of a virtual machine results in the whole of that virtual machine migrating to site s. After the migration, P commences execution in parallel with the rest of the body of the virtual machine.

There is a single primitive for interaction between virtual machines, allowing an atomic delivery of an asynchronous message between two virtual machines that are co-located on the same site. The execution of $\texttt{iflocal}\ \langle a\rangle c!v\ \texttt{then}\ P\ \texttt{else}\ Q$ in the body of virtual machine b has two possible outcomes. If the virtual machine a is on the same site as virtual machine b then the message $c!v$ will be delivered to a (where it may later interact with an input) and P will commence execution in parallel with the rest of the body of b; otherwise the message will not be delivered and Q will execute as part of b. This is analogous to test-and-set operations in shared memory systems—delivering the message and starting P, or discarding it and starting Q, atomically. It can greatly simplify algorithms that involve communication with virtual machines that may relocate away at any time, yet is still implementable locally, by the VM implementation on a single site.

Two other useful constructs can be expressed as sugar: $\langle a\rangle c!v$ and $\langle a@s\rangle c!v$ attempt to deliver $c!v$ (an output of v on channel c), to virtual machine a, on the current site and on s, respectively. They fail silently if a is not where it is expected to be, and so are usually used only in a context where a is predictable.The first is implementable simply as $\texttt{iflocal}\ \langle a\rangle c!v\ \texttt{then}\ \mathbf{0}\ \texttt{else}\ \mathbf{0}$; the second as $\texttt{create}^{\texttt{m}}\ b = \texttt{relocate to}\ s \rightarrow \langle a\rangle c!v \texttt{in}\ \mathbf{0}$, for a fresh name b that does not occur in s, a, c, or v.

[1] This choice may not be appropriate in the virtual machine setting, where one would presumably like to relocate VMs while retaining whatever network connections and connectivity they possess.

Turning to the π-calculus constructs, the body of a virtual machine may be empty (**0**) or a parallel composition $P|Q$ of processes.

Execution of **new** $c : \text{^}^I T$ **in** P creates a new unique channel name for carrying values of type T; c is binding in P. The I is a capability: as in (Pierce and Sangiorgi 1996), channels can be used for input only `r`, output only `w`, or both `rw`; these induce a subtype order.

An output $c!\,v$ (of value v on channel c) and an input $c?p \to P$ in the same virtual machine may synchronise, resulting in P with the appropriate parts of the value v bound to the formal parameters in the pattern p. Note that, as in other asynchronous π-calculi, outputs do not have continuation processes. A replicated input $*c?p \to P$ behaves similarly except that it persists after the synchronisation, and so might receive another value.

Finally, we have conditionals **if** v **then** P **else** Q, and local declarations **let** $p = ev$ **in** P, assigning the result of evaluating a simple value expression ev to a pattern p. In $c?p \to P$, $*c?p \to P$ and **let** $p = ev$ **in** P the names in pattern p are binding in P.

For a simple example program in the low-level calculus, consider the following VM server.

$$\begin{array}{l} *getVM?[a\ s] \to \\ \quad \textbf{create}^{\mathtt{m}}\ b = \\ \quad\quad \textbf{relocate to}\ s \to \\ \quad\quad\quad (\langle a@s' \rangle ack!\,b \mid B) \\ \quad \textbf{in}\ \mathbf{0} \end{array}$$

It can receive (on the channel named $getVM$) requests for an virtual machine. This is a replicated input ($*getVM?[a\ s] \to \ldots$) so the server persists and can repeatedly grant requests. The requests contain a pair (bound to the tuple $[a\ s]$ of a and s) consisting of the name of the requesting virtual machine and the name of the site for the new VM to go to. When a request is received the server creates a virtual machine with a new name bound to b. This virtual machine immediately relocates to site s. It then sends an acknowledgement to the requesting virtual machine a (which here is assumed to be on site s') containing its name. In parallel, the body B of the served VM commences execution.

2.2 Processes of the High-Level Calculus

The high-level calculus is obtained by extending the low-level language with a single location-independent communication primitive.

$$\begin{array}{lcll} P & ::= & \ldots & \\ & \mid & \langle a@? \rangle c!\,v & \text{send } c!\,v \text{ to virtual machine } a \\ & & & \text{whereever it is} \end{array}$$

The intended semantics is that this will reliably deliver the message $c!\,v$ to virtual machine a, irrespective of the current site of a and of any relocations. The high-level calculus includes all the low-level constructs, so those low-level communication primitives are also available for interaction with application virtual machines whose locations are predictable.

2.3 Outline of the Reduction Semantics

2.3.1 Located Processes and Located Type Contexts

The basic process terms given above only allow the source code of the body of a single virtual machine to be expressed. During computation, this virtual machine may evolve into a system of many virtual machines, distributed over many sites. To denote such systems, we define *located processes*

$$LP ::= @_a P \mid LP|LQ \mid \textbf{new}\ x : T@s\ \textbf{in}\ LP$$

Here the body of a virtual machine a may be split into many parts, for example written $@_a P_1 | \ldots | @_a P_n$. The construct **new** $x : T@s$ **in** LP declares a new name x (binding in LP); if this is a virtual machine name, with $T = \mathtt{VM}^Z$, we have an annotation $@s$ giving the name s of the site where the virtual machine is currently located. Channels, on the other hand, are not located – if $T = \text{^}^I T'$ then the annotation is omitted.

Correspondingly, we add location information to type contexts. *Located type contexts* Γ include data specifying the site where each declared virtual machine is located; the operational semantics updates this when virtual machines move.

$$\Gamma ::= \bullet \mid \Gamma, X \mid \Gamma, x : \mathtt{VM}^Z@s \mid \Gamma, x : T \quad T \neq \mathtt{VM}^Z$$

For example, the located type context below declares two sites, s and s', and a channel c, which can be used for sending or receiving integers. It also declares a mobile virtual machine a, located at s, and a static virtual machine b, located at s'.

$$s : \mathtt{Site},\ s' : \mathtt{Site},\ c : \text{^}^{\mathtt{rw}}\mathtt{Int},\ a : \mathtt{VM}^{\mathtt{m}}@s,\ b : \mathtt{VM}^{\mathtt{s}}@s'$$

2.3.2 Reductions

To capture our informal understanding of the calculus in as lightweight a way as possible, we give a reduction semantics. It is defined with a structural congruence and reduction axioms, extending that for the π-calculus (Milner 1993). Reductions are over *configurations*, which are pairs $\Gamma \Vdash LP$ of a located type context Γ and a located process LP. We use a judgement $\Gamma \vdash a@s$, meaning that a virtual machine a is located at s in the located type context Γ. We shall give some examples of reductions, illustrating the new primitives. The most interesting axioms for the low-level calculus are given in Figure 2.

A virtual machine a can spawn a new virtual machine b, with body P, and continues with Q. The new virtual machine is located at the same site as a (say s, with $\Gamma \vdash a@s$). The virtual machine b is initially bound and the scope is over the process Q in a and the whole of the new virtual machine.

$$\begin{array}{rl} & \Gamma \Vdash @_a(R \mid \textbf{create}^{\mathtt{m}}\ b = P\ \textbf{in}\ Q) \\ \to & \Gamma \Vdash @_a R \mid \textbf{new}\ b : \mathtt{VM}^{\mathtt{m}}@s\ \textbf{in}\ (@_a Q \mid @_b P) \end{array}$$

When a virtual machine a relocates to a new site s, we simply update the located type context.

$$\begin{array}{rl} & \Gamma \Vdash @_a(R \mid \textbf{relocate to}\ s \to Q) \\ \to & \Gamma \oplus a \mapsto s \Vdash @_a(R \mid Q) \end{array}$$

A **new**-bound virtual machine may also relocate; in this case, we simply update the location annotation.

$$\begin{array}{rl} & \Gamma \Vdash @_a R \mid \textbf{new}\ b : \mathtt{VM}^{\mathtt{m}}@s'\ \textbf{in}\ @_b \textbf{relocate to}\ s \to Q \\ \to & \Gamma \Vdash @_a R \mid \textbf{new}\ b : \mathtt{VM}^{\mathtt{m}}@s'\ \textbf{in}\ @_b Q \end{array}$$

A virtual machine a may send a location-dependent message to a virtual machine b if they are on the same site. The message, once delivered may then react with an input in b. Assuming that $\Gamma \vdash a@s$ and $\Gamma \vdash b@s$.

$$\begin{array}{rl} & \Gamma \Vdash @_a(\textbf{iflocal}\ \langle b \rangle c![]\ \textbf{then}\ P\ \textbf{else}\ Q) \mid @_b(c?[] \to R) \\ \to & \Gamma \Vdash @_a P \mid @_b(c![] \mid c?[] \to R) \\ \to & \Gamma \Vdash @_a P \mid @_b R \end{array}$$

If a and b are at different sites then the message will get lost.

$$\begin{array}{rl} & \Gamma \Vdash @_a(\textbf{iflocal}\ \langle b \rangle c![]\ \textbf{then}\ P\ \textbf{else}\ Q) \mid @_b(c?[] \to R) \\ \to & \Gamma \Vdash @_a Q \mid @_b(c?[] \to R) \end{array}$$

Synchronisation of a local output $c!\,v$ and an input $c?x \to P$ only occurs within a virtual machine, but in the execution of **iflocal** a new channel name can escape the virtual machine where it was created, to be used elsewhere for output and/or input. Consider for example the process below, executing as the body of a virtual

machine a.

$$\begin{array}{l}\textbf{create}^{\mathtt{m}}\ b = \\ \quad c\textbf{?}x \rightarrow (x\textbf{!}3 | x\textbf{?}n \rightarrow \textbf{0}) \\ \textbf{in} \\ \quad \textbf{new}\ d : {}^{\wedge\mathtt{rw}}\mathtt{int}\ \textbf{in} \\ \qquad \textbf{iflocal}\ \langle b\rangle c\textbf{!}d\ \textbf{then}\ \textbf{0}\ \textbf{else}\ \textbf{0} \\ \qquad | \ d\textbf{!}7\end{array}$$

It has a reduction for the creation of virtual machine b, a reduction for the **iflocal** that delivers the output $c\textbf{!}d$ to b, and then a local synchronisation of this output with the input on c. Virtual machine a then has body $d\textbf{!}7$ and virtual machine b has body $d\textbf{!}3 | d\textbf{?}n \rightarrow \textbf{0}$. Only the latter output on d can synchronise with b's input $d\textbf{?}n \rightarrow \textbf{0}$. For each channel name there is therefore effectively a π-calculus-style channel in each virtual machine. The channels are distinct, in that outputs and inputs can only interact if they are in the same virtual machine. This provides a limited form of dynamic binding, with the semantics of a channel name (i.e., the set of partners that a communication on that channel might synchronise with) dependent on the virtual machine in which it is used; it proves very useful in the infrastructure algorithms that we develop.

The high-level calculus has one additional axiom, allowing location-independent communication between virtual machines.

$$\Gamma \Vdash @_a\langle b@?\rangle c\textbf{!}v \ \rightarrow\ \Gamma \Vdash @_b c\textbf{!}v$$

This delivers the message $c\textbf{!}v$ to virtual machine b irrespective of where b (and the sender a) are located. For example, below an empty tuple message on channel c is delivered to a virtual machine b with a waiting input on c.

$$\begin{array}{rl} & \Gamma \Vdash @_a(P \mid \langle b@?\rangle c\textbf{!}[]) \mid @_b(c\textbf{?}[] \rightarrow R) \\ \rightarrow & \Gamma \Vdash @_a P \mid @_b(c\textbf{!}[] \mid c\textbf{?}[] \rightarrow R)\end{array}$$

3. The Questions

So, are these calculi (or something similar) a level of abstraction that would be useful in managing datacentres, with widespread virtualization? Are there system design problems whose solutions would be best expressed at this level?

Insofar as there are problems involving the interaction of VM relocation and inter-VM communication, the answer seems (plausibly, to us) yes, but we are not in a position to know. We look forward to finding out.

References

1996. *CONCUR '96: Concurrency Theory, 7th International Conference*. LNCS, vol. 1119. Springer-Verlag, Pisa, Italy.

CARDELLI, L. 1995. A language with distributed scope. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, ACM, Ed. ACM Press, New York, NY, USA, 286–297.

FOURNET, C., GONTHIER, G., LÉVY, J.-J., MARANGET, L., AND RÉMY, D. 1996. A calculus of mobile agents. See (CON 1996), 406–421.

MILNER, R. 1993. The polyadic π-calculus: A tutorial. Series F: Computer and System Sciences, vol. 94. Springer. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

MILNER, R., PARROW, J., AND WALKER, D. 1992. A calculus of mobile processes, part I/II. *Information and Computation 100*, 1–77.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. The MIT Press.

PIERCE, B. C. AND SANGIORGI, D. 1996. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science 6*, 5, 409–454. An extract appeared in *Proc. LICS '93*: 376–385.

SEWELL, P., WOJCIECHOWSKI, P. T., AND PIERCE, B. C. 1998. Location independence for mobile agents. In *Proceedings of IFL 98: the Workshop on Internet Programming Languages (Chicago), in conjunction with ICCL*. 6pp.

SEWELL, P., WOJCIECHOWSKI, P. T., AND PIERCE, B. C. 1999. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*. Springer Verlag, 1–31.

THOMSEN, B., LETH, L., AND KUO, T.-M. 1996. A Facile tutorial. See (CON 1996), 278–298.

UNYAPOTH, A. AND SEWELL, P. 2001. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (London)*. 116–127.

WOJCIECHOWSKI, P. T. 2001. Algorithms for location-independent communication between mobile agents. In *Proceedings of AISB '01 Symposium on Software Mobility and Adaptive Behaviour (York, UK)*. Also published as Technical Report IC-2001-13, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL).

WOJCIECHOWSKI, P. T. 2006. Scalable message routing for mobile software assistants. In *Proceedings of EUC '06: the 2006 IFIP International Conferenc e on Embedded And Ubiquitous Computing*. Lecture Notes in Computer Science, vol. 4096. Springer, 355–364.

WOJCIECHOWSKI, P. T. AND SEWELL, P. 1999. Nomadic Pict: Language and infrastructure design for mobile agents. In *Proceedings of ASA/MA '99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents), Palm Springs, CA, USA*.

WOJCIECHOWSKI, P. T. AND SEWELL, P. 2000. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency 8*, 2 (April–June), 42–52.

2008/4/17

Practical structural and performance model checking

Eno Thereska
Microsoft Research, Cambridge UK
etheres@microsoft.com

Abstract

Performance management, tuning and debugging is a major source of headache for administrators of any large system. When a user complains (e.g., "why is my performance slow?") administrators need to perform two tasks: 1) localize the problem to a handful of resources and 2) find the root-cause of the problem. This position paper makes the case that the first task can be fully automated and the second task can be partially automated through sound system design. Lightweight, practical performance models should be first-class citizens and programmed inside a system. These models should continuously check expected and observed structural (how are requests flowing through the system) and performance (how long are requests taking at each module) behavior and flag unexpected deviations.

A preliminary study done within a cluster-based storage system shows that out of 29 performance anomalies encountered during a 3 year period, built-in models can correctly localize the source of the problem in 23 cases, identifying 18 system bugs and 5 model bugs. For all the 5 model bugs, the models were able to retrain the models correctly with minimal human interaction. More case studies, at different system levels (e.g., request flow through C# routines and through nodes in a distributed system) are needed to understand to make generalizations.

1. Extended abstract

To catch performance anomalies one needs models of expected behavior for both structural and performance properties of a system and workload. Figure 1 illustrates one simple expectation in a hypothetical data center consisting of a database and a storage subsystem. A structural expectation is that, when 3-way replication is used, three storage nodes should be contacted on a write, and acks should be subsequently received. A performance expectation is that three times the original block size should be seen on the client's network card. Information on CPU and network demands can be automatically discovered (e.g., CPU data encoding/encrypting should use 0.02 ms and it should take 0.5 ms to send the data to the storage-nodes). A myriad of methods are available for creating performance expectations. For example, a CPU model might be based on direct measurements (i.e., model emulates the real CPU processing of data and reports on the time that takes). A network model might be an analytical formula that relates the request size and network speed to the time it takes to transmit the request. Cache and disk models might be based on simulation and might replay previously collected traces with a hypothetical cache size and disk type.

In a distributed system with multiple resource tiers, one needs a general framework to reason about the effects of a hypothetical change in any of the tiers on end-to-end performance. *Queuing analysis*, which models the system as a network of queues, is the building block of such a framework (Lazowska et al. 1984). Each queue model represents a resource. Customers' requests place demands on these resources. Fundamentally, queuing theory assumes

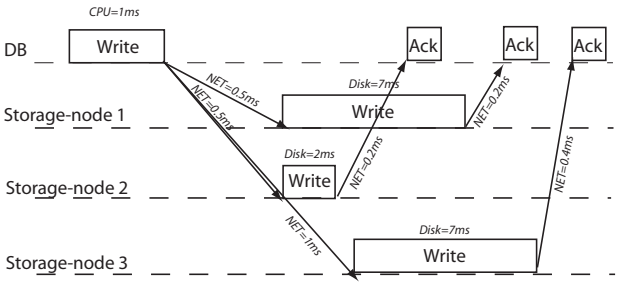


Figure 1: High-level structural and performance expectations for a storage sub-system in a data center.

knowledge of the way requests flow through the system's service centers (i.e., knowledge of the *queuing network*) and knowledge of the performance of the individual resources (i.e., *service centers* along the queuing network). For a sufficiently complex system, a main challenge is to update this knowledge constantly.

1.1 Challenges and lessons learned so far

We have implemented a prototype model checker in an experimental cluster-based storage system called Ursa Minor (Thereska and Ganger 2008). Ursa Minor is being developed for deployment in a university data center. On the software side, Ursa Minor consists of about 250,000 lines of code. Models in Ursa Minor are needed to predict performance consequences of data migration and resource upgrades.

Over the last 3 years, we learned several lessons from the model checking:

(1) Make models first-class citizens: Models are usually built by model designers, not system designers. Models are built about a system, not within the system. Each time the system changes, the models might become obsolete. The model designers would then need to consult the system designers as to what might have changed. The added communication overhead, coupled with tight project timelines often means that fixing the model (or system) has low-priority. We (the system designers) built models within the system from the start.

(2) Expect systems to be misconfigured or buggy: System components might have been implemented wrong or might be later misconfigured in the field.

(3) Expect models to be limited or buggy: Building models is not a flawless process either. We observe that models often have regions of system-workload interactions in which they work well and regions in which they do not. Reasons that such regions exist include non-linear behavior that is mathematically difficult to model and incomplete understanding by the model creator on how the system behaves under certain complex conditions.

For both (2) and (3) the models we built continuously validate the system's structural behavior (i.e., how requests flow through

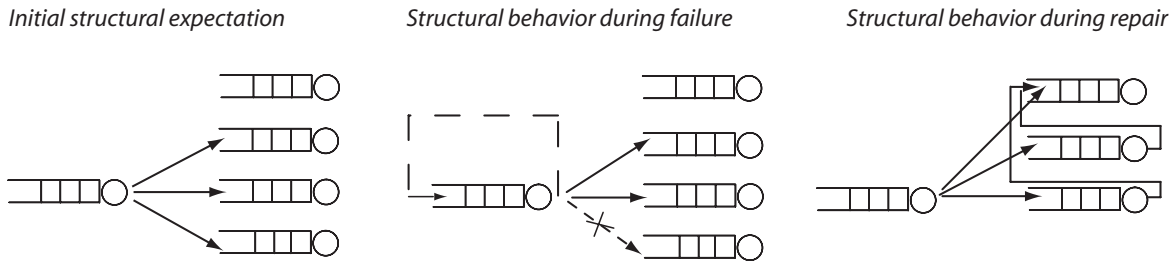


Figure 2: Structural deviations during failure and repair for 3-way replication. The initial structural expectation for write requests indicates that a write should contact three storage-nodes. If one of the storage-node fails, one of the writes times out and reenters the sending service center. During repair, a spare storage-node takes over from the failed one and is populated by the two remaining storage-nodes. Three graphs can be created and contrasted for each of these scenarios (nodes in such graphs would be entry and exit points from the service centers above and edges would represent the service center's response times).

it) and performance behavior (i.e., how long each request takes to be processed by each component). The redundancy of high-level system specifications described through the models and low-level implementations was used to identify the presence and help localize the source of a performance anomaly.

For example, Figure 2 shows causal graphs created for certain request in Ursa Minor. Those graphs can then be contrasted with the graph of structural expectation that the designer first inputs. Structural mismatches manifest themselves as a change in the fan-in/out of service center nodes. Performance mismatches manifest themselves as a change in the edge latency between service center nodes (a reasonable policy might define "change" as a one standard deviation from the expected average edge latency). To localize mismatches, a model must be built to continuously self-check its behavior and the behavior of the system.

(4) Building initial models is easy, refining them is hard: We explicitly defined the structural behavior of the system during the system design phase. This was not an easy task. However, the energy spent on it is a fraction of the energy already spent verifying the correctness of the various algorithms and protocols in the system. Concretely, several months were spent by several people designing and reviewing how requests would flow in Ursa Minor. Translating that work into a queuing network took one person less than a month. However, during the system operation, we quickly realized that as the system evolved, the models had to evolve as well. We coupled the queuing-based mathematical models with statistical components that give fidelity estimates by keeping track of historical information about their predictions. In many cases, these statistical models could continuously adjust queuing models' parameters for new workload-system attributes.

(5) End-to-end monitoring is a must: Good modeling requires good measurements, at all points in the system. We learned that, especially for detecting structural mismatches, a system must have a good measurement infrastructure in place that keeps track of requests at service centers' entry and exit points. Methods for designing such *end-to-end tracking* of requests have been an active area of research recently (e.g., Magpie (Barham et al. 2004)). The underlying technology for collecting traces efficiently is slowly being integrated into operating systems as well (e.g., ETW in the Windows operating system (Microsoft 2005)).

1.2 Ongoing work

Performance model checking should be done at all levels in a system. We have only experimented with model checking at the distributed communication level (i.e., RPC and above). It remains to be seen whether we can do similar model checking at lower levels (e.g., to check request flows through C# routines). A key reason the model checking worked well for Ursa Minor is that collecting end-to-end measurements is relatively cheap (when compared to the cost of accessing storage). Finding a good measurement infrastructure for lover levels of the system and analyzing the cost involved in collecting statistics is a natural next step.

Currently, our model checking prototype relies on system- and programmer-specific conventions for writing behavior expectations. Using a more formal language would be beneficial for portability reasons.

1.3 Related work

PSpec (Perl and Weihl 1993) and PIP (Reynolds et al. 2006) are most related to our work. PSpec is a language that allows system designers to write assertions about the performance behavior of their system. Once the assertions are written, they are continuously checked against the system. PIP augments PSpec by allowing designers to write assertions about the structural behavior of the system as well. Our approach builds on the above and generalizes them by trusting neither the model nor the system implementation as correct. We use a hybrid modeling scheme, where expectation-based models are augmented with statistical observation-based models to provide long-term fidelity in the model and system.

Acknowledgments

We thank the members of the Ursa Minor team for their support on this project.

References

Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Symposium on Operating Systems Design and Implementation*, pages 259–272, 2004.

E. Lazowska, J. Zahorjan, S. Graham, and K. Sevcik. *Quantitative system performance: computer system analysis using queuing network models*. Prentice Hall, 1984.

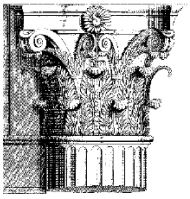
Microsoft. Event tracing, 2005. http://msdn.microsoft.com/.

Sharon E. Perl and William E. Weihl. Performance assertion checking. In *ACM Symposium on Operating System Principles*, pages 134–145, 1993.

Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Symposium on Networked Systems Design and Implementation*, pages 115–128, 2006.

Eno Thereska and Gregory R. Ganger. IRONModel: Robust performance models in the wild. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2008.

Traveling to Rome: a retrospective on the journey



john wilkes

HP Laboratories, Palo Alto, CA

john.wilkes@hp.com

Abstract

Starting in 1994/5, the Storage Systems Program at HP Labs embarked on a decade-long journey - to automate the management of enterprise storage systems by means of a technique we initially called attribute-managed storage. The key idea was to provide declarative specifications of workloads and their needs, and of storage devices and their capabilities, and to automate the mapping of one to the other. One of many outcomes of the project was a specification language we called Rome[1] – hence the title of this paper, which offers a short retrospective on the approach and some of the lessons we learned along the way.

Categories and Subject Descriptors D.4.2 Storage Management, D.4.5 Reliability, D.4.8 Performance, I.6.5 Model Development, K.4.3 [Organizational Impacts] automation, K.6.2 Installation Management, K.6.4 System Management.

General Terms Algorithms, Management, Measurement, Performance, Design, Economics, Reliability, Experimentation.

Keywords storage management; attribute-based storage; declarative system management; storage performance models; solvers.

1. Before the beginning

In the late 1980s, I'd worked on a scalable storage system called DataMesh [Wilkes1989], which advocated (about a decade too soon!) building a storage system out of intelligent building blocks containing a disk drive, some local processing power, and a high-speed network port. The idea was to connect these together into a mesh, and build a storage system that could be scaled to meet whatever performance or availability demands were placed on it. It quickly became obvious that such a beast would be a nightmare to control and configure if viewed a disk at a time, so we started to think about how you might delegate control of design choices to it, starting with failure recovery goals [Wilkes1990].

DataMesh never took off. But the seed of an interesting idea had been planted.

2. Setting out

In 1994, we were about to finish helping our colleagues on the HP AutoRAID project [Wilkes1996] and asked ourselves – "what if we could apply the same principles to an enterprise-scale storage system?" That is: what if users of large-scale storage systems didn't have to micro-manage the data placement, choice of RAID level, and kind and number of storage devices to purchase? What if the system could work these things out for itself, given a specification of what the customer wanted? The obvious motivations were offered: reduced system management costs; lower-cost system designs, faster (and more accurate) response to changing inputs; and fewer errors injected.

We wanted to separate the specification of what was desired from the process used to get to an answer – i.e., we were defining a declarative system for storage management. The name we chose was *attribute-managed storage* [Golding1995], by comparison to IBM's system-managed storage [Gelb1989]. Stores (data objects), streams (access patterns), and storage devices were each given attributes that specified their properties or behaviors. Streams were associated with stores. We called the process of assigning stores to devices the *mapping problem*, and proposed to solve it automatically.

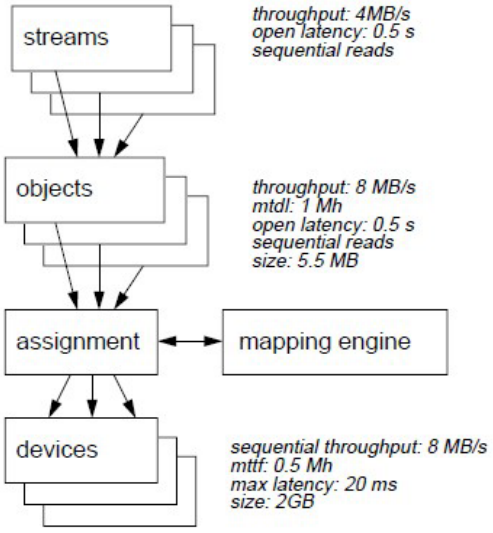


Figure 1. The attribute mapping problem.

Different aspects of the mapping problem included "how many devices are needed to support this load?"; "how much load can this set of devices support?" (which introduced a need for an early version of utility); and "half my data center has just burned down – which subset of the load can I still support?" In practice, we spent the majority of our time focused on the first question, on the

[1] The names chosen by the HPL Storage Systems program team for the various project components were derived from an architectural theme consistent with our logo – a Corinthian column. Over time, this progressed towards names with a generally classical bent. We apologize for none of them!

grounds that most users had a set of work they wanted to get done, and were interested in seeing how to support it.

3. Packing for the outward journey

It was pretty straightforward to generate a mathematical formulation of the mapping problem as a constraint-based optimization problem, with the constraints being things like "all workloads should be assigned exactly once", and "no capacity limit should be exceeded", and with objective functions of the form " minimize the cost of a complete solution", or "maximize the utility" [Shriver1996].

Two additional outcomes were observable at this stage: a first, clear specification of a set of parameters and attributes for workloads, stores, and storage devices; and the need for models to determine whether constraints were satisfied.

Adding up storage capacity to check a constraint was trivial; determining if the load imposed by placing a set of stores on a device would be too high was much trickier. We quickly ruled out simulations as being too costly, and resorted to simple analytic models for the expected behavior. Our background in simulation models for storage devices [Ruemmler1994] led us to a set of analytical models for disk devices that was more complete than most, and yet executed quickly [Shriver1998].

We had started down the path of analytical performance models that would occupy us for much of this leg of our journey.

To help ground our work, we picked the TPC-D benchmark as a representative sample of the kinds of application we would have to cope with. Taking I/O traces of a (non-audited!) system running this load showed us that there were several distinct phases in which one portion of the system was heavily used while another lay idle – and vice versa. Time-sharing the storage resources between different phases could save as much as a factor of six in storage system cost. Addressing this issue resulted in us developing a sophisticated set of performance models that could handle both short-term workload peaks and correlations between longer-term workload behaviors [Borowsky1998].

So far, we had just been modeling single disk drives. Our real target was disk arrays, which introduced a great many complications in the performance models for various RAID levels [Varki2004]. Hard work on analytical device models eventually addressed these [Uysal2001].

Nonetheless, the time required to generate a set of calibrated storage device models proved troubling – as did the fact that it took a set of highly competent people with PhDs to do it. An alternative approach was needed. We found it in a clever application of brute force. Instead of carefully crafting models that predicted the likely behavior of a storage device, we built models that extrapolated the likely behavior from sets of stored measurements – lots of them. We called this approach table-based modeling [Anderson2001]; using spline-based interpolations to fit the data, and being careful about unwarranted extrapolations gave us accuracies similar to – or better than – the analytic models, with comparable or better runtimes, and considerably less work on our part.

4. On the road, outbound

Early on, it became clear that the search space we wanted to explore was rather large: this is a variant of the multi-dimensional, multi-knapsack problem, which is (of course) NP complete – and the scale at which we were operating largely precluded exhaustive search.

We called the tools we used to explore alternative assignments of work to devices *solvers*. Our first attempt at a solver was called Forum; it handled the single-device models described above, and used greedy hill-climbing to select the best alternative [Borowsky1997]. A few simple heuristics for ordering the examination of alternatives were explored, including (repeated) randomization of the order to consider workloads for assignment; sorting the workloads on various attributes, and using the Toyoda algorithm for deciding which device to pack the next load onto [Toyoda1975].

Forum tackled performance for single storage devices. A completely separate tool, Corbel, was the first to tackle the conjoint design problem for availability and performance at the same time [Amiri1996]. Corbel synthesized RAID designs, tested their availability against the objectives associated with stores, and then selected a suitable design from the ones that were left. Unfortunately, Corbel was never integrated into our mainstream code base – partly because it relied on a somewhat hard-to-use Markov chain analysis tool that was written in Fortran. Corbel used a greed first-pass assignment process that took the raw hardware cost plus the cost of downtime into account, followed by a refinement pass that fixed up the solution by selective moving of a few stores that were not well matched to their placements.

5. Making good progress

Our second attempt to support disk arrays was a solver called Minerva [Alvarez2001]. That tackled the problem in two parts: it first tagged workloads with the recommended kind of RAID level that they should be assigned to, and then performed a Forum-like assignment. As with Corbel, a final optimization pass cleaned up a few stragglers – especially stores that ended up consuming an entire RAID group.

To make Minerva work well, we had to make good choices about deciding which RAID level to use for each store [Anderson2002]. Using simple rules of thumb – as a human might do – produced acceptable answers, but integrating the choice into the process of assigning stores to devices did much better, albeit at the cost of some additional computation time.

At one point we thought that genetic algorithms (GAs) seemed like an obvious approach to this problem: the species genotypes would represent the current sets of assignments of load to devices, and mutations and combinations would explore the space of alternatives in an efficient fashion. Having tried the experiment, we learned that the cost of evaluating each of the solutions was so high that it dominated the running time of the GA solver, even after aggressive memoization.

Initially we had been leery of trying to do performance- and availability-based assignments simultaneously because of the huge search space that it engenders. However, Eric Anderson was able to get around this problem by constructing a solver that used speculative exploration plus a tree-like representation of the design of a storage system and its assignments. The solver, called Ergastulum,[2] performed much faster than Minerva, and was able to explore a great many more alternatives [Anderson2005].

6. Arriving at the destination

The material so far has described how we developed solutions to the declarative design of a single storage system configuration. But our goal was always to develop a way to make the storage

[2] The name means a private prison attached to most Roman farms, where the slaves were made to work in chains. It was selected when Eric was a summer intern in our group – he claims that it seemed like a good idea at the time. Regrettably, the ACM TOCS reviewers took aversion to it, and we had to drop it from the published version.

system self managing – by which we meant self-configuring, self-optimizing, self-healing, and all of the other self-* objectives.

The approach was straightforward – at least in principle: (1) take a specification for what is wanted; (2) build a storage system that matches those needs; (3) deploy the application or workload on that system; (4) monitor it to see if it is meeting the actual needs of the workload; (5) re-design if necessary, and migrate the application to the new configuration – preferably while it is still running.

Hippodrome was the name of the system we devised to do all this [Anderson2002a]. It went one better: it didn't need a detailed specification of the performance requirements of the workload, just the capacity and availability needs. It would run the application, measure the result, design and deploy a system to meet those needs, and iterate until the result stabilized – typically in only 2-3 iterations. This was the system we had been aiming for all along.

7. Language barriers

Workload descriptions (streams plus stores), device capabilities, models, objective functions, and configuration settings for our tools all needed writing down. Some years before we had invented a way of marrying Tcl with a complicated simulation system [Golding1994], and we continued this approach as we developed ways to write down these various inputs. The result was a fairly flexible language for recording attributes and other specifications that naturally supported nesting of components and dynamic extensibility (by being interpreted, and making it easy to ignore elements that were not understood).

We christened this language Rome; it stood us in good stead for quite some time, but eventually became encrusted with hidden assumptions about the meanings of various elements and their relationships.

Rome 2 was an attempt to provide a clean specification for both the syntax and semantics of the language we were using. It was derived from the *de facto* version, and followed it quite closely in may ways.

We should have done this sooner; by the time Rome 2 was ready, it was too late – the team had moved on to other goals. Another lesson was the importance of separating the semantics of a language from its expression. Well-meaning people kept on pressing us (unhelpfully) to use XML – as if that would solve any of our problems. In practice, having a language that humans can manipulate, plus automated translations back and forth into a more "standard" representation like XML, is the right way to proceed – a lesson that has yet to be relearned by many groups, I fear.[3]

8. The journey back

One slightly troubling aspect of our approach that we had chosen to elide was how we were going to answer the question: "where do the requirements come from?" Hippodrome offered one way out (measure them), but that doesn't work for systems that don't exist yet, or for non-measurable metrics such as availability or reliability targets.

We never did come up with a better answer for the first problem, but we did make some headway on the latter, by taking a step back and realizing that availability and reliability requirements are ultimately driven by business needs. If we could extract those business needs, we reasoned, we could use them to drive decision-making about the right availability levels to push for.

In fact, we ended up realizing that we could go one better: if business objectives can be expressed in monetary terms – such as the hourly cost of an outage (unavailability) or data loss – we could add that to the [calculated] cost of achieving a particular level of availability or reliability, and treat the result as an optimization problem, with the objective of minimizing their sum. This turned out to work well; first for designing the storage system itself [Keeton2004], for evaluating how well the storage system will behave when things go wrong [Keeton2004a], and – best of all – for working out how to recover once things have started to go wrong [Keeton2006]. We suspect that the latter is particularly valuable, as the likelihood of making errors increases greatly when people are under stress.

9. Entertaining excursions

Hippodrome requires the ability to reconfigure a storage system between iterations, but there are plenty of other reasons to want to move data from one setup to another. We found that applying the same kind of declarative problem-specification plus an automated solver to the migration problem led to similar dividends. The setup here is simple: descriptions are provided of an initial data layout and a final one, and the goal is to derive a plan that moves the data from one to the other, while minimizing the number of spare staging areas needed, or the elapsed time, or both [Saia2001, Anderson2008]. The problem is by no means completely solved: our work didn't support changing the format of containers or taking performance effects such as network bottlenecks into account.

As described, the storage system design cycle is a long-lived one, operating at the timescale of provisioning decisions (hours or days). In order to cope with shorter-term fluctuations, it's necessary to provide a finer-grained control mechanism. One approach to this is to enforce quality of service at runtime [Karlsson2004, Wang2007]. Doing so requires a clear understanding of the specifications that it is intended to enforce – another example of the need for precise declarative specifications.

10. Returning home

What has all this taught us about declarative approaches? First: they can be made to work, at significant scale and complexity, and across a wide range of problems. The capabilities of the technology are exciting; and the use of goal-based declarative specifications seems much cleaner than rule-based or process-based ones such as workflows.

Second: that deploying such systems is much more than a technical problem. In fact, I believe that the single greatest barrier to adoption of such systems is not our ability to generate the technology to build such systems, but our ability to persuade the likely users that they should trust that the systems will do the right thing. To this end, we need to invest more in making our systems trustworthy – which means ensuring that they don't surprise people; making it easier to express what we want them to do; putting limits on what they can do without our consent; and explaining their decisions when requested.

Ultimately, we need to remind ourselves that we are building systems to serve people, and the success of our technical accomplishments will be dictated by how comfortable we can make those people with what we are accomplishing on their behalf.

[3] To press this point home, two forms of representation were provided for the Rome language: the native version (derived from the Tcl syntax) was called Latin; the alternative XML one was called Greek – and was typically 2-3 times as long.

Acknowledgments

The work described here was actually done by a talented pool of colleagues who I had the pleasure – and good luck – to work with over the last two decades. Space precludes listing them, but my thanks to them all!

References

[Alvarez2001] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems* 19(4):483-518, November 2001.

[Amiri1996] Khalil Amiri and John Wilkes. *Automatic design of storage systems to meet availability requirements*. Technical report HPL-SSP-96-17, HP Laboratories, August 1996.

[Anderson2001] Eric Anderson. *Simple table-based modeling of storage devices*. Technical report HPL-SSP-2001-4, HP Laboratories, July 2001.

[Anderson2002] Eric Anderson, Ram Swaminathan, Alistair Veitch, Guillermo A. Alvarez and John Wilkes. Selecting RAID levels for disk arrays. *File and Storage Technology* (*FAST'02*, Monterey, CA) pp. 189-201, January 2002.

[Anderson2002a] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. *File and Storage Technology* (*FAST'02*, Monterey, CA) pp. 175-188, January 2002.

[Anderson2005] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, Qian Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems* 23(4): 337-374, November 2005.

[Anderson2008] E. Anderson, J. Hartline, M. Hobbs, A. Karlin, J. Saia, R. Swaminathan and J. Wilkes. Algorithms for Data Migration. *Algorithmica*, to appear, 2008

[Borowsky1997] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E.Shriver, M.Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. *5th Intl. Workshop on Quality of Service* (*IWQoS*, Columbia Univ., New York, NY), June 1997, pp. 199-202.

[Borowsky1998] Elizabeth Borowsky, Richard Golding, Patricia Jacobson, Arif Merchant, Louis Schreier, Mirjana Spasojevic and John Wilkes. Capacity planning with phased workloads. *Workshop on Software and Performance* (*WOSP'98*, Santa Fe, NM), October 1998.

[Gelb1989] J. P. Gelb. System managed storage. *IBM Systems Journal* 28(1):77–103, 1989.

[Golding1994] Richard Golding, Carl Staelin, Tim Sullivan, John Wilkes. "Tcl cures 98.3% of all known simulation configuration problems" claims astonished researcher! *Tcl Workshop* (New Orleans), May 1994.

[Golding1995] Richard Golding, Elizabeth Shriver, Tim Sullivan, and John Wilkes. Attribute-managed storage. *Workshop on Modeling and Specification of I/O* (San Antonio, TX), 26 Oct. 1995.

[Karlsson2004] Magnus Karlsson, Christos Karamanolis and Xiaoyun Zhu. Triage: performance isolation and differentiation for storage systems. *International Workshop of Quality of Service* (*IWQoS'04*, Montreal, Canada), pp. 67-74, June 2004.

[Keeton2004] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase and John Wilkes. Designing for disasters. *File and Storage Technologies* (*FAST'04*, San Francisco, CA), March-April 2004.

[Keeton2004a] Kimberly Keeton and Arif Merchant. A framework for evaluating storage system dependability. *International Conference on Dependable Systems and Networks*, (*DSN'04*, Florence, Italy), June-July 2004.

[Keeton2006] Kimberly Keeton, Dirk Beyer, Ernesto Brau, Arif Merchant, Cipriano Santos and Alex Zhang. On the road to recovery: restoring data after disasters. *European Systems Conference* (*EuroSy'06s*, Leuven, Belgium), pp. 235-248, April 2006.

[Ruemmler1994] Chris Ruemmler and John Wilkes. An introduction to disk drive modelling. *IEEE Computer* 27(3):17-28, March 1994.

[Saia2001] Jared Saia, Eric Anderson, Joe Hall, Jason Hartline, Michael Hobbes, Anna Karlin, Ram Swaminathan, and John Wilkes. An experimental study of data migration algorithms. *Algorithm Engineering, the Proceedings of WAE 2001: 5th Workshop on Algorithm Engineering* (BRICS, University of Aarhus, Denmark), August 2001). Published as *Springer-Verlag Lecture Notes in Computer Science* 2141, pp. 145-158, August 2001.

[Shriver1996] Elizabeth Shriver. *A formalization of the attribute mapping problem*. Technical report HPL-SSP-95-10 revision D, HP Laboratories, July 1996.

[Shriver1998] E. Shriver, A. Merchant, and J. Wilkes. An analytical behavior model for disk drives with readahead caches and request reordering. *Int'l. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 182–91, June 1998.

[Toyoda1975] Y. Toyoda. A simplified algorithm for obtaining approximate solutions to zero-one programming problems. *Management Science*, 21(12):1417–27, August 1975.

[Varki2004] Elizabeth Varki, Arif Merchant, Jianzhang Xu and Xiaozhou Qiu. Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 15(6):559-574, June 2004.

[Wang2007] Yin Wang and Arif Merchant. Proportional share scheduling for distributed storage systems. *File and Storage Technologies* (*FAST '07*, San Jose, CA), February 2007.

[Wilkes1989] John Wilkes. *DataMesh --- scope and objectives*. Technical report HPL-DSD-89-37rev1, HP Laboratories, July 1989. http://www.hpl.hp.com/research/ssp/papers/#DataMesh

[Wilkes1990] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. *4th ACM-SIGOPS European Workshop* (Bologna, Italy), September 1990, published as *Operating Systems Review* 25(1):56-59, January 1991.

[Wilkes1996] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems* 14 (1):108-136, February 1996.

[Uysal2001] Mustafa Uysal, Guillermo A. Alvarez, and Arif Merchant. A modular, analytical throughput model for modern disk arrays. *9th Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems* (*MASCOTS'01,* Cincinnati, Ohio), pages 183-192, August 2001.

Microsoft Research Ltd
7 J J Thomson Avenue
Cambridge, CB3 0FB, UK
www.research.microsoft.com