December 2000

Service Composition for Electronic Commerce

David Edmond
Queensland University of Technology

A. ter Hofstede
Queensland University of Technology

Service Composition for Electronic Commerce

David Edmond and Arthur H.M. ter Hofstede

Cooperative Information Systems Research Centre
Queensland University of Technology
GPO Box 2434, Brisbane, Queensland 4001
Australia
e-mail: {davee,arthur}@icis.qut.edu.au

We investigate the possibility of some kind of automated and *ad hoc* trading based on the formulation of specific commercial arrangements derived from libraries of commonly-used trading patterns. Against this background, we discuss the issue of service composition, and introduce some basic forms. We then discuss the issue of composition as a process in its own right, and how that process may be extended with new forms.

Keywords: task structures, service composition

1 Introduction

The Internet and the Web have opened new ways of doing business cheaply and more effectively. The development of loosely-coupled commercial undertakings, in the form of *virtual enterprises*,[1] seems a likely outcome from this connectivity. Deals will be made, and settlement arranged, in a highly-automated fashion. But more than that, such deals will be triggered by discovery mechanisms that are continuously scanning, on a global basis, for opportunities to trade. In a recent report to the US Congress on this emerging digital economy, Margherio, Henry et al. (1998) forecast that, by the year 2002, $US300 billion worth of Internet transactions being performed annually. The globalisation triggered by the Internet has naturally spurred the development of tools and aids to navigate and share information on the Internet. Organisations of all sizes are keen to sell products to a wider customer base by transacting on the Internet (Lee 1998).

The dynamism, volatility and sheer unpredictability of this environment will require the ability to *adapt* to previously unknown situations, possibly while actively trading. We believe that some form of enhanced features from *reflection* and *process modelling* will provide the right ingredients to enable adaptability.

Reflection (Kiczales, des Rivières & Bobrow 1991) is the act of revealing a system's implementation, and allowing changes to that implementation in a controlled manner. By inspecting internal aspects of a system, its competence may be improved, either through better performance or greater adaptability. The concepts of reflection have been applied in a great many areas of computing and information technology, where it has

[1]A virtual enterprise is a pragmatic inter-organisational tool used by otherwise competing companies in an *opportunistic* way. Its aim is to enable such companies to respond quickly to frequent demands for complex products (Goldman, Nagel & Preiss 1995).

been employed as a means of constructing flexible and extensible computer systems – ones that can evolve and adapt to changing circumstances and expectations (Maes 1988).

A scenario of automated or semi-automated dealing depends greatly upon the availability of software components that can effect the necessary processes, and on being able to combine these processes in a way that suits the particular requirements of the deal being considered. One area for investigation is the possibility of a library of fairly generic activities, for example:

- Working on a project – where the work is possibly performed as a number of tasks spread over a period of time.
- Trading stock – where there is a need to negotiate on price and volume over some limited period of time.
- Settlement – where an amount of money is to be paid off, according to set rules.

After suitable *instantiation* and *extension*, these commonly-used patterns may then be *composed* in some way – resulting in some specific commercial arrangement that enables a particular deal to go ahead. This kind of ad hoc composition of task templates may require one or both of the following:

- Recognisable *hooks* within a template that allow it to be articulated with other templates.
- Programming constructs that provide the necessary *glue* to support any such articulation.

In this paper, we begin by reviewing a processing modelling language called *Task Structures.* This notation will be used to describe not only the activites upon which a deal is based, but also the deal itself. Then we look at different ways in which tasks may be composed. Then we discuss the kind of meta-knowledge that composition requires, and how reflection might be used to extend the compositional process, in general.

2 Task structures

Generally speaking, processes focus on the *coordination* of tasks. Any process specification language should at least be capable of capturing moments of choice, sequential composition, parallel execution, and various forms of synchronisation. *Task Structures* (Hofstede & Nieuwland 1993) is a notation for describing the various tasks within a process, and their interdependencies. An individual task structure may be composed from one or more of five particular building blocks, or process patterns, as is shown in Figure 1.

AND-split:	*After* process A has been performed, both process B *and* process C will be performed. Process A is said to *trigger* the other two processes.
AND-join:	Process C cannot be performed *until* both A *and* B have been performed. This form of join is known as a *synchroniser.*
XOR-join:	Process C will be performed whenever either A or B, *but not both*, have been performed. This is known as a *discriminator.*
XOR-split:	*After* process A has been performed, either B *or* C *but not both* will be performed. This is known as a *decision point.*
OR-join:	Process C will be performed whenever A *or* B *or both* have been performed (and so, C may be performed twice).

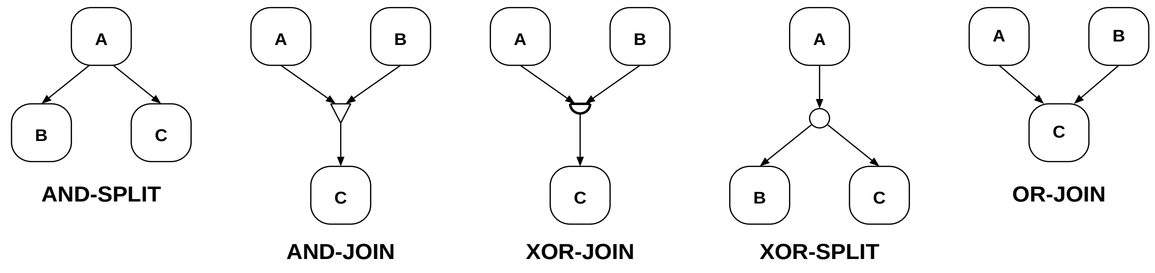


Figure 1: Building blocks for Task Structures

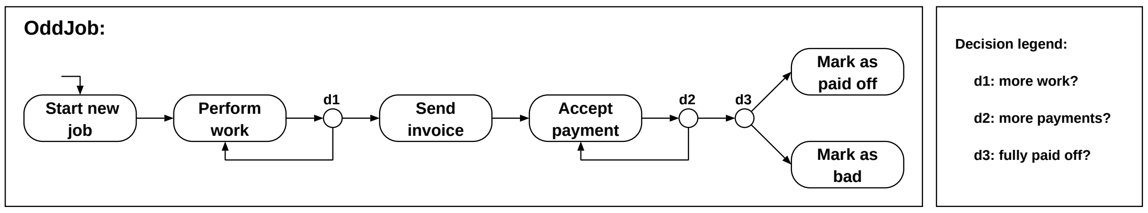


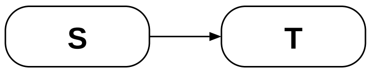
Figure 2: Finishing the entire job before being paid

Figure 2 is an example of a task structure. It represents a process in which, first of all, a new job is initiated, then the work is performed in a series of stages. An invoice is sent out when the job is completely finished. Next a cycle of payments commences. The process terminates when a decision is made that no further payments are expected.

3 Composition

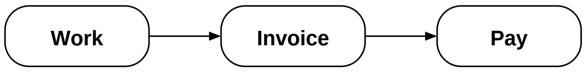
Composing atomic tasks:

Suppose we have two task structures $\mathcal{S}$ and $\mathcal{T}$, and we want to perform task $\mathcal{S}$ and then task $\mathcal{T}$. Their articulation may be effected by forming a new task structure, one that consists of an initial task $\mathcal{S}$ which, on completion, triggers task $\mathcal{T}$.



This would be a very straightforward kind of strictly sequential composition. Task $\mathcal{S}$ will need to be *entirely* finished before task $\mathcal{T}$ commences.

Consider the project work situation in its simplest form. Company A is to carry out some work on behalf of Company B: the work is carried out; a signal is then transmitted, in the form of an invoice, from company A to company B; finally, company B pays up.



The structure shown above is the simplest possible arrangement: only one job is peformed, only one invoice is prepared, and only one payment is necessary. These three basic activities might be composed in a number of different ways.

The structure shown in Figure 2 may also be viewed as the outcome of a composition process. A *project work* pattern is followed by a *settlement* pattern, with a *send invoice* task acting as an intermezzo, and the decision on the payment outcome acting as a coda. There are other ways in which the three tasks might be combined.

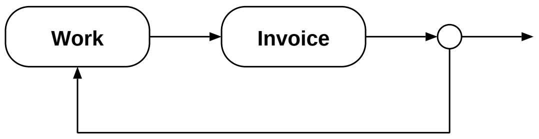


Figure 3: A simple, but restrictive, composition of the *work* and *invoice* tasks

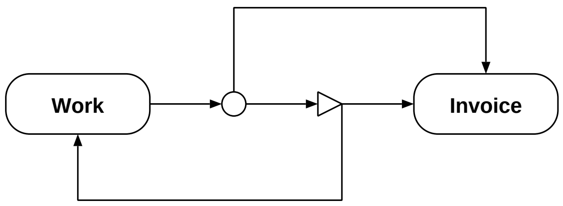


Figure 4: A composition that allows some parallelism of the tasks

Suppose that the work takes some time to perform, and, by its nature, consists of a number of stages. Then, it may be desirable to send an invoice at every stage. These two tasks could be combined in the way shown in Figure 3, where there is a cycle of work- then-invoice which is repeated until no further work remains to be performed. The problems here are that an invoice is sent *every* time work is performed, and that no further work can take place until the invoice for the previous stage has been prepared.

We can improve the situation by coupling them in a different manner, as shown in Figure 4. There, after some work has been performed, a decision is made as to whether or not there is more work to be undertaken. If there is not, then the invoice is prepared, and the whole task ceases. If there is still more work to do, then a synchroniser is triggered. This will result in two paths being followed: an invoice will be prepared *and* another unit of work will be executed. Thus, some parallelism is enabled.

Organic composition:

Alternatively, the composition may be more *organic* with the contents of the two structures being exposed, followed by some kind of attempt to "grow" the two structures together. For example, the terminal tasks of $\mathcal{S}$ may be matched up in some way with the initial tasks of $\mathcal{T}$.

Perhaps these two options are really just at the extreme ends of some kind of spectrum of possibilities. A decision, however, to reveal the internal structure of the component task structures seems an important distinction. Similarly a terminating decision may feed, instead, into some initial task of T.

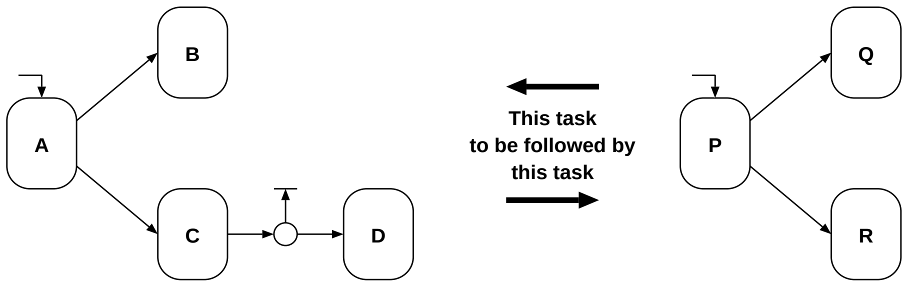


Figure 5: Two tasks to be composed

The animation of a task structure will probably give rise to more than one execution thread. Should *all* threads in one task structure be complete before commencing the next structure? Two basic approaches seem possible:

- An *eager* philosophy will be one in which the second task is started as soon as any thread in the first task has died.
- A *cautious* philosophy will require that *all* threads be complete before the second task commences.

With an eager approach, the conclusion of each thread will trigger the commencement of the ensuing structure. For example, as may be seen in Figure 6, there will be three different ways in which the threads of the first task may conclude, and the second task fired up. The second task, however, will only be fired up twice, but even twice may be too often. The problem with this approach is that it will lead to multiple executions of the second task. However, although being "eager" might have its disadvantages, it allows the execution of the second task to be expedited.

With a cautious or conservative approach, we will wait until the first task has entirely finished, that is, all its threads have terminated. A naive way to accomplish this would, as shown in Figure 7, be simply to interpose a synchroniser between all termination points of the first task and all initial tasks of the second structure.

Unfortunately, such an approach, in the example shown, is guaranteed to deadlock: the synchroniser will wait for signals from the termination option of the terminating

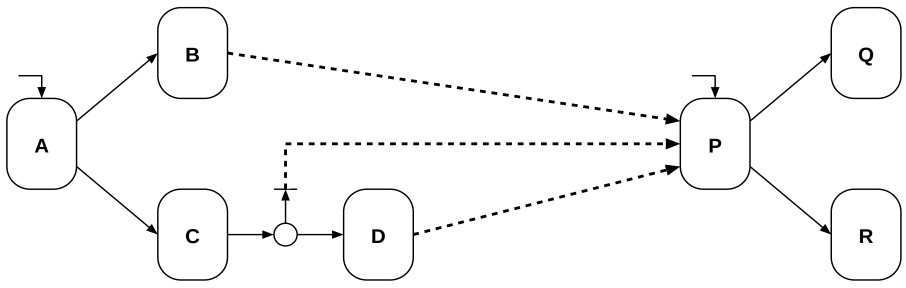


Figure 6: Two tasks composed in an eager fashion

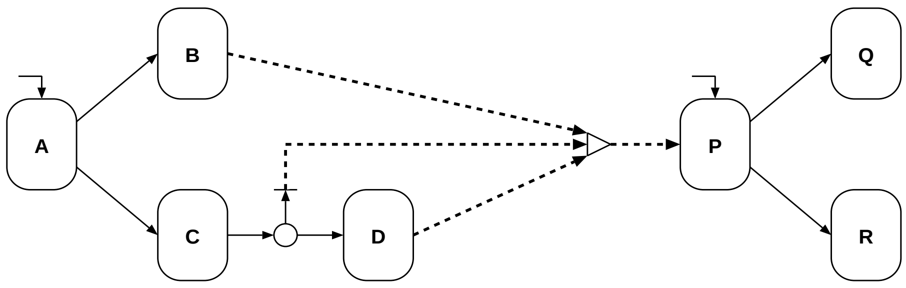


Figure 7: Two tasks composed in a naïvely cautious fashion

decision point *and* the conclusion of subtask D. By the very nature of a decision point, only one of these events can occur.

It is clear that the sequential composition of two or more structures is a kind of synchronisation, and that the standard synchroniser is inadequate. The difficulty may be resolved by allowing a different form of synchronisation – known as a **discriminator**.[2] Its use is shown in Figure 8, where it is simply interposed between the two structures. The discriminator accepts the *first* signal from $t2$, $t4$ or $t5$. It then triggers the initial task or tasks of the ensuing structure. Any later signals from the other two task items are recognised by the discriminator, but no onward triggering occurs. Thus, we have the kind of synchronisation that we sought in Figure 7.

4 Knowledge about composition

Composing two task structures $\mathcal{S}$ and $\mathcal{T}$ may be viewed as a process of taking structure $\mathcal{S}$ and decorating it in some way with task $\mathcal{T}$. The process will involve the addition of new subtasks, decision points, synchronisers and discriminators. We can represent the manner in which the composition takes place as yet another task structure, as may be seen in Figure 9.

There are two task structures in this figure. *Task composition* shows the general *process of task composition*, that is, it shows how the process of composing tasks is controlled. The composition process consists, firstly, of a decision as to which particular

[2]The concept is similar to that of the discriminator in the Verve Workflow product (Verve 1999).

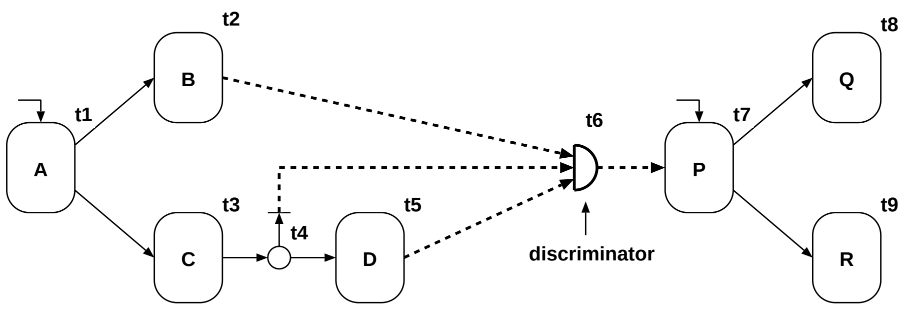


Figure 8: Using a discriminator to compose tasks

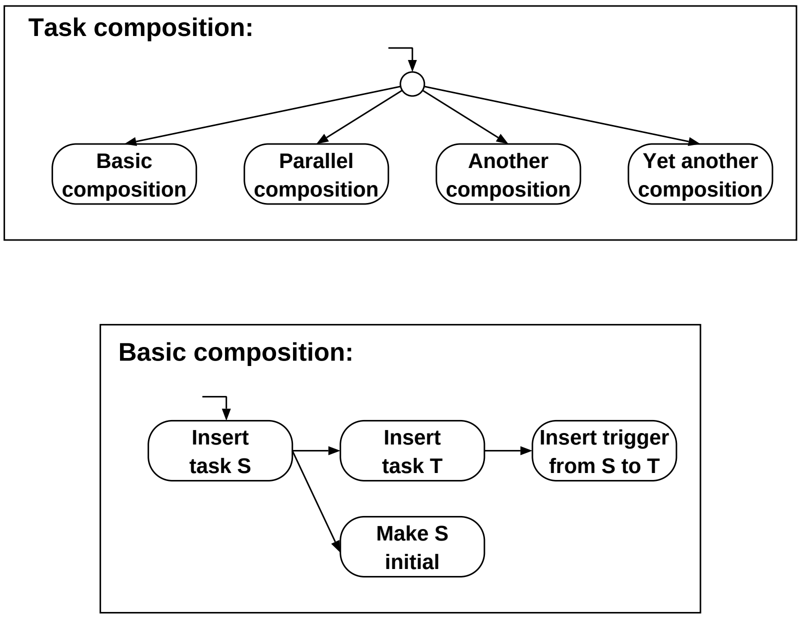


Figure 9: Controlling the composition of tasks

kind of composition is to be performed. The *Basic composition* subtask is also a structure, and it begins by inserting the task that is to be performed first. Then, in whatever order, that task is made initial, and the other task is inserted. Finally, a trigger is inserted that connects tasl $\mathcal{S}$ to task $\mathcal{T}$.

Three levels of task structure can be identified. Starting from the application level, and moving into the kernel of the system:

- At the *application-domain* level, we have task structures that describe application processes. Figure 2 shows an example.
- At the *composition* level, we have task structures that describe how application structures may be composed. Examples of such tasks are shown in the *Task composition* and *Basic composition* tasks of Figure 9
- A the *composition-adaptation* level, we have task structures that describe how com-

position (as a process in its own right) may be controlled and modified.

The task structures that we build at this innermost level will determine what kind of task composition we will be able to accomplish: one possibility is to use a *null* task structure, but this is a very limiting although safe strategy, since no changes would be allowed to the middle level at all. Thus, we would be allowed to make the kinds of composition that were envisaged at the time of construction, but no others. Another option is to have a task structure that allows carefully controlled modifications to the middle level. The modifications will be limited to ones that introduce new compositional styles.

5 Related work and conclusions

In attempting to formulate a research agenda for the rigorous composition of software systems, Nierstrasz, Schneider & Lumpe (1997) observe that:

- There are two complementary viewpoints of software engineering – the computational and the compositional. An important task will be to discover what kind of abstractions are necessary for software composition.
- If we hope to compose software in the same way hardware is composed, a (software) *component framework* must support the specification of component behaviour, protocols for intercomponent communication, and rules for component substitutability.

Workflow patterns: In the work of (Aalst, Barros, Hofstede & Kiepuszewski 2000), a systematic overview of process control constructors is provided. The patterns address business requirements in an imperative workflow style, but are independent of any particular workflow language. They encapsulate commonly used forms of complex workflow functionality. An example is the *N-out-of-M Join* where *M* parallel paths converge into one. The subsequent activity is activated once *N* paths become active. An instance of such a pattern would be when *three* reviewers are each sent a copy of a paper, but the review process ceases when *two* reports are returned.

Generic process models: In (Aalst 1999), each workflow schema is associated with a *family* of variants. A particular task in the schema may be viewed as the root of an extensible class hierarchy, with the hierarchy expressing allowable instantiations of that particular task.

Architecture Description Languages: Software architecture specification is intended to describe the structure of the components of a software system, their interrelationships, and principles and guidelines governing their design and evolution (C3DS 1997). Architecture Description Languages (ADLs) are high-level notations for expressing and representing architectural designs and styles. ADL descriptions are an ideal way of precisely describing modern software systems that are composed of components; furthermore such descriptions permit construction of system development environments that can map ADL specifications onto the base system services. ADL based system development environments hold the promise of abstracting away the differences between agent and workflow based systems (Clements 1996).

In a distributed environment, applications must be more than just fault-tolerant, they must also be capable of dynamic reconfiguration. Ranno, Shrivastava & Wheater (1999) describe a *scripting* language that enables tasks to be composed, and inter-task dependencies to be specified. Two kinds of dependency are examined: a *notification* dependency from task A to B indicates that task B cannot commence until task A has finished; a *dataflow* dependency between the two tasks indicates that, in addition, task B requires some input from task A.

Coordination languages: This is a family of languages that are designed to glue applications together. Control-driven coordination is the term that is used to refer to models of coordination that are event- rather than data-driven (Papadopoulos & Arbab 1998). Concepts within the *Manifold* coordination language are the *process* which is a kind of black-box to or from which, information may be passed by means of named *ports*. These ports may have *streams* attached to them, thereby processes may be interconnected. As well as streams, there is an mechanism by which *events* are broadcast, to be picked up by any interested processes.

In this paper, we have introduced the idea of forming commercial arrangements *ad hoc*. These arrangements will arise by firstly instantiating and extending commonly-used patterns appropriately chosen from libraries of such activities. The results will then be composed in some way – resulting in some specific deal. We have also discussed how the general process of composition may be specified in such a way as to allow the incorporation of new forms of dealing, as required.

References

Aalst, W. v. d. (1999), How to handle dynamic change and capture management information, *in* 'Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems (CoopIS-99), Edinburgh, Scotland', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 115–126.

Aalst, W. v. d., Barros, A., Hofstede, A. t. & Kiepuszewski, B. (2000), Workflow patterns, Technical report, Queensland University of Technology. (Submitted for publication).

C3DS (1997), 'C3DS: Control and coordination of complex distributed services'. Downloaded from `http://www.tcd.research.ec.org/c3ds/programme.html` on `25-Jun-1999`.

Clements, P. C. (1996), A survey of architectural description languages, *in* 'Proceedings of the 8th International Workshop on Software Specification and Design, Paderborn, Germany', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 16–25.

Goldman, S. L., Nagel, R. N. & Preiss, K. (1995), *Agile competitors and virtual organizations : strategies for enriching the customer*, Van Nostrand Reinhold, New York, NY, USA.

Hofstede, A. t. & Nieuwland, E. (1993), 'Task structure semantics through process algebra', *Software Engineering Journal* **8**(1), 14–20.

Kiczales, G., des Rivières, J. & Bobrow, D. G. (1991), *The Art of the Metaobject Protocol*, The MIT Press, Cambridge, Mass., USA.

Lee, H. G. (1998), 'Do electronic marketplaces lower the price of goods?', *Communications of the ACM* **41**(1), 73–80.

Maes, P. (1988), ‘Computational reflection’, *The Knowledge Engineering Review* **3**(1), 1–19.

Margherio, L., Henry, D. et al. (1998), *The Emerging Digital Economy*, US Dept of Commerce.

Nierstrasz, O., Schneider, J.-G. & Lumpe, M. (1997), Formalizing composable software systems – a research agenda, *in* E. Najm & J.-B. Stefani, eds, ‘Proceedings of 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS-96), Paris, France’, Chapman & Hall, London, England, pp. 271–282.

Papadopoulos, G. A. & Arbab, F. (1998), Modelling electronic commerce activities using control-driven coordination, *in* A. Tjoa & R. Wagner, eds, ‘Proceedings, 9th International Conference on Database and Expert Systems Applications, Workshop on Coordination Technologies for Information Systems (CTIS-98), Vienna, Austria’, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 583–590.

Ranno, F., Shrivastava, S. K. & Wheater, S. M. (1999), A language for specifying the composition of reliable distributed applications, Technical Report C3DS TR No. 12, University of Newcastle upon Tyne, England. Downloaded from `http://www.laas.research.ec.org/c3ds/trs/index.html` on `13-Jul-1999`.

Verve (1999), *Process Editor*, Verve Inc. `http://www.verveinc.com/technology/processeditor.htm`.