On the mapping of business contracts to executable choreography

A. Berry
<andyb@whyanbeel.net>
Brisbane QLD
Australia

Z. Milosevic
<zoran@dstc.edu.au>
Distributed Systems Technology Centre
University of Queensland
Brisbane QLD 4072
Australia

Abstract

In this paper we present a mapping from the concepts and terms of a business contract language (BCL) to the executable primitives of a fully-distributed choreography engine (Finesse). In doing so we demonstrate the feasibility of compiling a business contract specification into an executable program suitable for execution on emerging choreography platforms. Through this process we also identify minimal properties of a business contract specification necessary to deterministically transform the specification into an executable program, and similarly, we highlight the key semantic properties of a choreography engine necessary to implement business contracts. This mapping indicates how a business contract specification might be mapped onto a standard process execution environment for E-commerce.

1 Introduction

Business contract languages are approaching the point where specifications in such languages can be used directly in the execution and monitoring of electronic business contracts. Commercial products are available to manage contracts within an organisation for some common processes like procurement[6, 10, 25], and recent research has proposed more general languages for cross-organisational contract management[15, 17, 28, 8, 26].

In parallel with this activity in support of electronic contracts, the maturity of technology for the co-ordination of activities or processes spanning multiple autonomous participants is improving. Recent work in the W3C standards body is focusing on so-called *choreography* specifications[13] and the OASIS consortium is working to ensure that the ebXML standards[7] are capable of describing and supporting choreography. In parallel with these efforts, there are a number of organisations that have announced plans to support choreography standards[4]. The primitives in choreography languages have been guided by previous work on co-ordination languages[5, 2] and the semantics of specification languages for distributed processes, particularly pi calculus[19].

This paper examines the likely convergence of these technologies in a concrete fashion by presenting a mapping from BCL[17, 23] to the executable primitives of the Finesse choreography engine[1]. These technologies share a common heritage in work on the RM-ODP standards[11] and subsequent collaborative work[21, 27]. While both BCL and Finesse can operate capably in an environment with centralised control, this paper is focused on distributed, autonomous, business-to-business activities because such activities give rise to a unique set of issues that are not present in systems with centralised control. These issues will be discussed in more detail in subsequent sections.

A mapping between a business contract specification and a process definition indicates how rules and policies in business contracts can be used in the design of processes for E-commerce and E-business platforms. We envisage that such platforms will be the target infrastructure for implementation and monitoring of electronic contracts in the future.

The following section of the paper describes the technologies in more detail, providing a basis for the mapping of BCL terms to Finesse primitives. In section 3, a mapping between BCL and Finesse is presented using an example contract for a purchasing process. This is followed in section 4 by a discussion of the issues uncovered, in particular, the capabilities required of both a choreography engine and a business contract language.

2 Technology Overview

Finesse and BCL are technologies emanating from the Distributed Systems Technology Centre (DSTC), both directly and through sponsored PhD research programs[1, 20]. They were both influenced in the early stages by the

participation of DSTC in the RM-ODP[11] standards effort of the mid-90s, and the relationship has been continued in BCL through its adoption of community models from the ODP enterprise language[12]. A number of joint papers involving the authors[21, 27] have also influenced the subsequent development of the technologies. The following subsections describe the technologies in more detail. A discussion of the relationship between these technologies and other technologies and standards is also presented.

2.1 BCL

Business Contract Language (BCL) has been developed for the purpose of specifying contract conditions so that the contract execution can be monitored in an event-based fashion. This contract execution ultimately refers to various activities of the signatories to the contract, and whether these activities signify fulfillment or violation of policies agreed in the contract. A special case of policy violation refers to situations in which a required activity of a signatory to the contract has not been carried out—thus the monitoring needs to detect these activity non-execution cases.

Events are the key behavioural modelling concept in BCL. A single event can be used to signify an action of a signatory or to signify a temporal occurrence such as a deadline. Multiple events can be combined and used to describe the execution of more complex activities. The BCL concept of *event pattern* is used to specify events, relationships between events and properties of events that are of relevance to business contracts. Examples are logical relationships between events (*AND, OR, NOT*), temporal relationships between events (e.g. before and after), temporal constraints on event patterns (e.g. absolute and relative deadlines and sliding time windows[24]), event causality, and certain special kinds of of singleton event pattern (e.g. contract violation and state change events). The event pattern approach has many similarities to the work by Luckham on complex event processing[18].

Event patterns are a key component in checking policies related to a contract. Policies define behavioural constraints in terms of event patterns and are associated with the roles that execute activities. Policy checking consists of identifying event patterns in activities of parties filling a role and ensuring that they satisfy the policies.

It is often the case that contract related events can change the state of certain variables associated with the contract. To this end, BCL introduces the notion of a state (of a specific contract variable) and the value of this state can be either determined explicitly in response to an event, or on request when the state value is needed. A contract can have many states that are changing to reflect the corresponding events.

A contract consists of a set of policies that apply to the behaviour of signatories to the contract. Thus policies take a form of modal constraints such as obligations, permissions and prohibitions. These modal constraints in a contract specification reflect their English-language meaning: obligations identify activities that must occur, permissions identify activities that may occur, and prohibitions identify activities that must not occur. In all cases, these constraints can be conditional, for example, if payment is not made then the supplier is permitted to charge interest on the outstanding amount.

BCL introduces the concept of a *community*, which can be regarded as an overarching concept for the specification of objects that collaborate to achieve a certain goal. These objects fill the roles of a community. So, a community is defined as a set of roles, policies, states and related event patterns that apply to the community. A community is a general concept for describing collaboration and one specific kind of community is a business contract.

2.2 Finesse

Finesse provides a model, operational semantics, language and execution engine for co-ordination of behaviour across distributed, autonomous participants. A Finesse `Binding` specification defines a set of roles, the role behaviours, and the interactions between roles. Role and interaction behaviours are specified separately, capturing the key semantic distinction between local and distributed behaviour.

Behaviour is specified in terms of events templates and their relationships. Events templates are templates for the execution of events and are explicitly associated with a role. Roles are bound to locations at instantiation time and the event templates of a role can then be executed by the location(s) bound to that role. Relationships between event templates are declaratively expressed as causal dependencies. For example, if we have event templates `E1` and `E2`, a causal dependency declaring that E2 causally depends on E1 is expressed in the language as `E1 -> E2`. The implication of this declaration for execution is that an execution of E2 cannot occur until the location at which E2 occurs is aware that an execution of E1 has occurred. If E1 and E2 are co-located, then this is immediate and requires no further action. If E1 and E2 are not co-located, then messaging is required to ensure the location of E2 is aware that E1 has occurred.

Data flow is also defined declaratively through functional relationships between event parameters and can only exist in the presence of a causal relationship. For example, `E1 -> E2 {E2.z = f(E1.x, E1.y)}`, specifies that `E2` depends on `E1` and its parameter `z` is equal to the result of evaluating the function `f(E1.x, E1.y)`.

Roles in Finesse have cardinality, that is, a single role can be filled by more than one participant in a *Binding* in-

stance. Role behaviour defines the behaviour of a single participant in isolation. Any co-ordination across participants filling the same role must be specified as interaction behaviour. This maintains the distinction between local and remote behaviour.

Interaction behaviour defines event relationships for event templates executed in distinct roles or between distinct participants filling the same role. The existence of a causal relationship between event templates executed at distinct participants implies messaging between the participants. The implicit nature of this messaging allows compilation to optimise messaging semantics if desired. For example, notifications of two local event occurrences can be combined if both events are required to satisfy causal dependencies at a remote location. The presence of explicit parameter relationships also allows optimisation of message payload, that is, the execution engine need only send those parameter values required to evaluate parameter relationships.

The Finesse model, semantics, language and the execution engine are described in detail in chapters 4, 5 and 6 of [1] respectively. There are also published papers describing the language[2] and the semantic model[3] in isolation. The operational semantics is formally defined in [1].

2.3 Relationships with Other Technologies

While there are no standards comparable with BCL, some aspects of its behavioural specifications overlap with available business process definition and execution technologies like BPEL[29] and ebXML[7]. BCL allows a business-oriented specification of policies or contract terms to be applied to existing process definitions. An event pattern is a means for describing a state of affairs. A state of affairs can range from the elementary, such as the occurrence of a particular action performed by a party or the passing of a deadline, to the more complex, such as ”more than three sets of down time in any a one week period” and ”one of the contract conditions has been violated”. Event patterns are used within policies to specify policy constraints in a contract. The goal is not to specify complete behaviour or business processes, but to define contract-related constraints over that behaviour. As an analogy, BCL can be thought of as an aspect-oriented language[14] that expresses business policies for processes.

One of the key targets for BCL is the specification of policies for business-to-business interactions. Early process technologies for workflow and most existing business process management products are restricted to definitions of the local process. There is no capability to express constraints on interactions between autonomous business processes. The emerging choreography technologies and standards[4, 13] address this deficiency. As with local processes, BCL can be seen as complementary to these technologies.

Finesse pre-dates the recent work on choreography technologies by several years but provides implemented technology and formal operational semantics satisfying the key goal of choreography: to co-ordinate the behaviours of distributed, autonomous participants in a business-to-business interaction. The semantic model provides expressiveness equivalent to or exceeding the capabilities of major process models and technologies like pi calculus[19] and ebXML[7]. The explicit model of locality and use of a truly concurrent operational semantics are key distinguishing features that make it ideal for describing processes that span distributed, autonomous participants. The Finesse engine can be thought of as a distributed, asynchronous and programmable process engine. The engine does not, however, provide any means of storing state beyond the state embodied in the run-time history of executed events.

In the context of business contracts and choreography, we expect that the Finesse engine or an equivalent choreography engine would be embedded in a web services environment to control and monitor the execution of business-to-business interactions, with existing BPM and workflow technology used to manage processes behind the web services interface within each organisation.

3 Mapping BCL to Finesse

The common heritage of BCL and Finesse discussed in section 2 provides us with an opportunity to explore the mapping between contract languages and choreography engines without the effort required to resolve significant semantic differences that might have occurred for entirely distinct technologies. The key issues lie primarily in the need to map a domain-specific language (BCL) onto the generic, technology-focused operational semantics implemented by the Finesse engine.

The mapping explicitly identifies the relationship between BCL terms defined in [17, 23, 22] and the Finesse language presented in [1]. In subsequent sections, the name *Finesse* can be understood to refer to this language rather than the engine or semantic model. The mapping of the Finesse language to the operational semantics implemented by the Finesse engine is described in detail in chapter 5 of [1]. A compiler would most likely use the Finesse operational semantics to map BCL concepts directly onto the internal form used by the engine for execution, but this internal form is not human readable and so the language syntax is used in this paper.

This following subsections define the mapping between key semantic concepts and syntactic elements in BCL and corresponding Finesse semantic concepts and syntactic elements. The mapping is driven by a purchasing exam-

ple, with fragments of language expressions introduced progressively and the mapping discussed alongside these fragments. As such, the mapping defines the compilation of key elements from the BCL language elements in the example to Finesse language elements.

Note that the resulting Finesse behaviour will not necessarily be a complete specification of the behaviour required to implement a purchasing process. The BCL fragments are contract-oriented constraints over the purchasing process. The Finesse behaviour specifications generated by the mapping need to be conjoined with the procedural behaviour specifications for the purchasing process to generate a complete process definition.

In the following text, BCL terms are highlighted through use of italicised sans-serif font like *this* and Finesse terms are highlighted through the use of a fixed-width font like `this`.

3.1 Example Overview

The example defines a relatively simple contract for a purchasing process involving three roles: purchaser, supplier and freighter. The purchaser places orders with the supplier, the supplier is obliged to fill those orders within an agreed time frame and the freighter is obliged to transport the goods from supplier to purchaser within an agreed time frame. Beyond these basic roles, the following terms apply to the contract:

1. The purchaser has a credit limit with the supplier as part of the agreement. The credit limit is a maximum outstanding amount with no particular time limit. The purchaser is not permitted to exceed the credit limit.
2. The supplier is permitted to provide a bill every 30 days from contract start (a point in time).
3. The purchaser is obliged to pay the balance of the monthly bill within 30 days of the billing date and time.
4. The supplier is obliged to have goods ready for shipment within 5 days of receipt of a purchase order.
5. The freighter is obliged to deliver goods within 5 days of the goods being ready for shipment.

The following subsections walk through a BCL definition of this contract and its terms, defining Finesse code fragments to match each BCL fragment. Discussion of each transformation is presented to highlight relevant issues and semantic difficulties.

3.2 CommunityTemplate

A BCL contract definition is a *CommunityTemplate*, introduced with the following syntax:

CommunityTemplate: PurchasingContract

A *CommunityTemplate* is the context for defining contract behaviour and is introduced by a name. The definition of roles, states, policies and event patterns is contained within the template, as discussed in subsequent sections. The corresponding term in the Finesse language is a `Binding` which defines a context for choreography behaviour specification. In both cases, these are the top level components of a specification. The matching Finesse syntax is thus:

```
Binding PurchasingContract { ... }
```

The binding specification is defined by code appearing within the braces {}, indicated by the ellipses (`...`) in the fragment above.

3.3 Role

A BCL *Role* is used to associate behaviour with a party to a contract at the specification level. BCL roles are names, with the expected behaviour of parties filling roles defined in subsequent *Policy* and embedded *EventPattern* definitions associated with a specific *Role* name. The syntax for role identification is as follows:

Role: Purchaser
Role: Supplier
Role: Freighter

The corresponding term in Finesse is also named `Role`. Finesse roles define the behaviour of participants in a `Binding`. Finesse `Role` definitions contain a complete definition of the visible behaviour associated with the `Role`. This behaviour is always local to the participant, that is, it defines the event templates and event relationships that a participant filling the role can execute. The role behaviour cannot identify or reference behaviour associated with other roles: this is remote behaviour. The corresponding syntax for the roles above is thus:

```
Roles {
  Purchaser {...}
  Supplier {...}
  Freighter {...}}
```

As we progress through the example, each of the policy statements will add behaviour to these Finesse roles. The Finesse role definition is defined as the logical `AND` of the behaviours corresponding to BCL policies.

Note that both BCL and Finesse roles have cardinality, that is, more than one party in a contract or participant in a binding can fulfill a role and some roles are optional. The default, however, is that a single participant fills each role. Other cardinalities are not used in the example.

3.4 Event Pattern

As discussed previously in section 2.1, event patterns are a key component in checking policies related to a contract. Policy checking consists of identifying event patterns in activities of parties filling a role and ensuring that they satisfy the policies. In most cases, events are matched with an event pattern by event type. The purchase order is the key data exchanged and its type is specified as follows:

EventType Id=PurchaseOrder
Defined by XMLSchema
for EDIFACT PurchaseOrder

This specifies that a purchase order event is signified by the existence of an XML document using the EDIFACT PurchaseOrder XML schema. Events in BCL can involve multiple *EventRoles*. The *EventRole* concept is a generic labelling mechanism for identifying roles in event execution that can be played by participants. An event with multiple roles is specified as follows:

Event typeId=PurchaseOrder
EventRole name=Buyer
EventRole name=Seller

Note that the event roles do not match the contract roles: by using generic event role names, the same event definition can be re-used in many contexts.

The multi-party nature of events in BCL implies that a messaging process or protocol is required to "execute" the event. BCL does not define this protocol or constrain the messaging semantics. For the purposes of this sample transformation, the first role identified in a BCL event specification is considered to be responsible for starting the execution, with all other roles requiring notification of that start. We also assert for this transformation that the data associated with the event is defined at the start and is immutable.

To transform the BCL for an event into Finesse code, we will use two fragments of code: the first defining a generic implementation of BCL events, and the second defining the *PurchaseOrder* event as a specialised version of the generic implementation.

```
Binding BCLEvent (DOCUMENT) {
  Roles {
    Initiator {initiate!(DOCUMENT)}
    [# >= 0] Receiver {receive?(DOCUMENT)}}
  Interactions {
    Initiator.initiate ->
      [#=all] Receiver.receive {*= prev}}
}
```

This is a parameterisable definition of event dissemination with multiple recipients. The `DOCUMENT` place holder can be replaced with any valid parameter expression. The default cardinality for roles is exactly one, so there is only one `Initiator`. The `[# >= 0]` cardinality specification on the `Receiver` role specifies that there can be zero or more receivers. The `Initiator` executes an output event `initiate` and this is followed by all `Receiver` participants executing a `Receive` event. The `*= prev` expression indicates that the `receive` event should have its parameters assigned to the value of the same-named parameters from the preceding event. A key point to note, however, is that this Finesse definition does not include any semantics for handling communication failure: the receiver of an event will wait indefinitely for the event notification to be delivered. This reflects the nature of the BCL specification, which also makes no provision for failure. If required to address legal concerns, communication failure can be addressed in the BCL specification by separating the sending and receipt of the purchase order and defining policies for non-receipt of a purchase order.

The second fragment describing a re-usable `PurchaseOrder` event is as follows:

```
Binding PurchaseOrder {
  Import BCLEvent;
  Roles {
    Buyer {
      BCLEvent.Initiator((doc:EdifactPO))}
    Seller {
      BCLEvent.Receiver((doc:EdifactPO))}}
  Interactions {
    BCLEvent(Buyer,Seller)}
```

In this definition, we have specialised the `Initiator` role definition to define the parameter list as containing a single `EdifactPO` parameter. There is an assumed translation between the data type systems of the two languages. Since Finesse uses a generic type system based on the notions of sets and functions this is a reasonable assumption, although it is something that deserves further exploration in future work.

Finally, we specify the binding in BCL of event roles to contract roles. This is achieved for the purchasing example using the following code:

Event typeId=PurchaseOrder
EventRole name=Buyer
RoleType name=Purchaser
EventRole name=Seller
RoleType name=Supplier

This applies to all purchase orders in the community template, meaning that the event pattern will only be matched if the *Purchaser* fills the *Buyer* event role and the Supplier fills the Seller role.

The mapping of event roles to contract roles is matched in Finesse using the following code in the top-level binding specification:

```
Import PurchaseOrder;
Roles {
  ...
  Purchaser {
    loop {PO {PurchaseOrder.Buyer}}
    AND ... }
  Supplier {
    loop {PO {PurchaseOrder.Seller}}
    AND ... }}
Interactions {
  PurchaseOrder(Purchaser.PO,
                Supplier.PO)
  AND ... }
```

Our previous fragments have been complete binding specifications suitable for re-use whereas this is a fragment of the purchasing contract binding specification. The `loop` iteration constructs signifies that the enclosed role behaviour can be repeated indefinitely: without the loop, the `Purchaser` would be permitted to initiate only one `PurchaseOrder`. This iteration is the default behaviour in BCL, with explicit policies required to limit the number of occurrences of an event. Since we need to describe interactions between specific fragments of behaviour in participants, we must identify those fragments unambiguously, hence the introduction of the name `PO` for the behaviour in each role definition. The interaction behaviour binds the `PurchaseOrder.Buyer` and `PurchaseOrder.Seller` behaviours of the `Purchaser` and `Supplier` respectively. Iteration constructs are not used in the definition of interaction behaviour: iteration is implied by role behaviour.

Event specifications in the same style as the *PurchaseOrder* specification are assumed for the other events referenced in the example, specifically, *GoodsReady, GoodsDelivered, InvoiceSent, Payment, GoodsShipped*.

3.5 State

BCL has a *State* construct for defining data values shared by the participants in the *CommunityTemplate*. This is used to maintain running totals, counters and other state required to evaluate policy. Such state defines a set of update actions and is introduced with the following syntax:

```
State: OutstandingDebt
  CalculationExpression
    UpdateOn: Payment
    UpdateSpecification:
      return this - Payment.Amount
  CalculationExpression
    UpdateOn: GoodsDelivery
    UpdateSpecification:
      return this + GoodsDelivery.Amount
```

This defines the outstanding debt of a purchaser, which is updated whenever an order is delivered and whenever a payment is made. State changes are bound to event patterns and are deterministic, that is, the value of a state can only be modified through the matching of visible event patterns. While such state is relatively easy to maintain consistently in an environment with centralised control, maintaining state in a distributed context is considerably more difficult.

In Finesse, the only state maintained is the event history and this is maintained at all participants as a partial view of the complete event history. The local event history is updated only by local event execution and the delivery of remote event notifications. This avoids any need for explicit synchronisation between distributed engines but also forces a programmer or compiler to explicitly define the semantics of shared state if required. The current implementation of Finesse also does not have any ability to provide persistent storage of state at participants.

For this mapping, we will take the approach of defining an extra Finesse role responsible for shared state. This extra role is a separate role to avoid static specification of the participant "owning" the state. The choice of participant to maintain this state can be made when the binding is instantiated or deployed, noting that a single participant is permitted to fill multiple roles in a run-time Finesse binding. The operations to update the state are omitted from the Finesse specification because it does not support persistent state. To maintain the state according to the contract specification a compiler could, for example, generate a J2EE entity bean implementing the update semantics against a database and deploy this as required at one of the participants. Assuming the existence of an remote procedure call (RPC) binding fragment[1], the necessary code is thus defined as follows:

```
Import RPC;
Roles {
  ...
  State {
    outstandingDebt {
      loop {
        debit{RPC.Server((x:Real)())} OR
        credit{RPC.Server((x:Real)())} OR
        balance{RPC.Server(()(y:Real))}}}}
Interactions {
  ...
  Freighter.GoodsDelivered.Sender.initiate
    -> State.outstandingDebt.debit.receive
      {x=prev.amount}
  AND Purchaser.Payment.Sender.initiate
    -> State.outstandingDebt.credit.receive
        {x=prev.amount}}
```

In this fragment, a new `State` role has three RPC operations on `outstandingDebt` for `debit`, `credit` and `balance`. The `debit` behaviour is fired by the freighter initiating a `GoodsDelivered` event. Note that Finesse

does not require that an RPC server response event is attached to a corresponding client receive: it will be silently discarded if not required. Similarly, the `credit` behaviour is fired by the purchaser initiating a `Payment` event. The `balance` behaviour will be used by any policy requiring the current balance. If other state is required in a BCL contract, it can be added to this Finesse role in a similar manner or alternatively, a separate Finesse role for each BCL state can be defined.

There are many performance, scalability and robustness implications in the above code. For example, one of our policies is that a purchase order is only permitted if the outstanding debt is less than the credit limit. This means that all purchase orders are preceded by an RPC to retrieve the balance. If the participant holding the balance is unavailable, then the purchase order cannot proceed. Other solutions can remove this problem by changing the synchronisation behaviour or moving the state to the place where it is required, but all generic solutions will be susceptible to dirty reads, failures or starvation (consistent failure of updates because of synchronisation requirements) when consistent state is required in more than one location.

Given this discussion, any *CommunityTemplate* that requires consistent state shared across participants must be handled with some care. For a cross-organisational contract, it will almost always be preferable to replace shared state with explicitly located state and an asynchronous state alignment protocol.

3.6 Policy

A *Policy* is used to specify business-level constraints in a BCL *CommunityTemplate*[17]. It is explicitly associated with a *Role* and has a *Modality* indicating whether it is an obligation, permission or prohibition. The behaviour associated with a policy is a conditional expression over events expressed as an event pattern. There are three modes of policy expression: obligation, permission and prohibition. The mapping of these policy expressions to Finesse language constructs is discussed in the following subsections.

3.7 Obligation

A BCL *Policy* can have an *Obligation* modality, indicating that the event pattern defined in the policy must occur. An obligation is specified as follows:

Policy: FillPuchaseOrder
Role: Supplier
Modality: Obligation
Condition:
$GoodsReady.date < PurchaseOrder.date + 5\ days$

The policy specifies the purchase order filling time condition as an event pattern. The policy specifies that the supplier is obliged to have the goods ready for shipment within 5 days of the purchase order. A simple mapping of this behaviour is as follows:

```
Roles {
  ...
  Supplier {
    loop { PO ->
        [timeless(prev, 5*3600*24)]
          GoodsReady XOR
        [not timeless(prev, 5*3600*24)]
          FillPOViolated }
    AND ... }}
```

From the BCL policy specification, we can infer that the *GoodsReady* event must have a causal dependency on a preceding *PurchaseOrder* event. In our translation to Finesse, we use guards and a logical XOR to specify that if the GoodsReady behaviour is not executed within 5 days, a `FillPOViolated` event is executed to indicate that the policy has been violated. This violation event is implicit in BCL, although if the violation event is referenced in other contract behaviour, it can be explicitly identified in the specification. Note that the `timeless(...)` function is a built-in guard function that evaluates to true when the time since the identified event (in this case, the keyword `prev` indicating the preceding event in the specification) is less than a specified number of seconds.

The definition of monitoring for this obligation is made easier by the explicit specification of a time period in the policy. If the obligation is not satisfied in the the time period, then a violation event is generated. A violation event can also be explicitly associated related policies to indicate non-fulfillment of the obligation in question. An obligation with no mechanism for identifying non-fulfillment is considered incomplete and cannot be adequately monitored or enforced.

In this mapping we have used the knowledge that both *PurchaseOrder* and *GoodsReady* event patterns are visible in the Supplier role, either as initiator or as receiver. Because both events are locally visible, the obligation can be locally monitored. If this is not the case, then we have to decide where to evaluate and monitor the obligation. The issue of where to monitor *Obligations* is an instance of the general problem of making consistent observations in distributed systems[9]. Consider our example: a supplier has an obligation to have goods ordered by a purchaser ready for shipment within 5 days. There are several ambiguities arising from this simple English statement:

- When does the 5-day period begin? In our example, we have specified that begins when the supplier receives the order. Without this explicit specification, there is considerable potential for ambiguity and dispute.

- Who monitors or enforces the 5-day period? If monitored by the purchaser, then the purchaser must have confirmation the order was received with an accurate and trusted timestamp indicating the time of receipt. What if the confirmation is lost or is timed out?
- If the purchaser is monitoring, how do they know that the goods have been shipped? They need a notification that the order has been shipped, also having an accurate and trusted timestamp indicating the time of shipping. What if that notification is lost or times out?

Notice that in all three items above, the location of actions is critical to the contract. If the time period is measured from the time the order is sent by the purchaser, the supplier becomes responsible for lost orders. If monitoring were to be carried out at the supplier, then we remove the problem of lost notifications but add the need for increased trust in the supplier to admit violations (which we have assumed in our solution above). If we use transactional messaging, the problem of lost notifications is removed, but at what point does the transactional messaging system stop attempting to send a failing message and notify the sender of failure? In the general case it is impossible to guarantee delivery of messages in a fixed time frame so this exception must be explicitly handled. The implication is that remote monitoring of obligations can only identify possible violations, with further evidence required from the source of monitored events to accurately establish the violation of an obligation in the event of message failure. More complex obligations can involve behaviour at multiple locations so using a generic rule to detect violations at the source cannot be universally applied.

We can resolve these issues in the general case by defining a monitoring `Role` in Finesse for each BCL *Obligation*. Roles in Finesse are bound to a location during binding establishment, so the monitoring location can be identified by the parties involved for each run-time instance and could potentially use a third-party monitor. Since Finesse roles can be filled by multiple participants, it is conceivable that more than one participant actively monitors the obligation. Note also that we must rely on accurate measurement and recording of time with events at all locations.

Similar specification of obligations in BCL and mappings to Finesse can be defined for the other obligations identified in the example overview of section 3.1.

3.8 Permission

A BCL *Policy* can have an *Permission* modality, indicating that the behaviour defined in the policy is allowed to occur. For example:

```
State: LastInvoice
  InitialisationSpecification: StartDate
  CalculationExpression
    UpdateOn: InvoiceSent
      UpdateSpecification:
        return InvoiceSent.date

Policy: MonthlyInvoice
  Role: Supplier
  Modality: Permission
  Condition:
    (InvoiceSent.date - LastInvoice) = 30 days
```

We use a State specification to capture the date of the last invoice which is initialised to the contract start date. The policy specifies that the supplier is permitted to send an invoice only when exactly 30 days have elapsed since contract start or the last invoice. We will assume that the *LastInvoice* state is mapped to Finesse in the same manner as the *OutstandingDebt*, in other words, through an interface with a `invoiceDate` RPC to retrieve the date. In this case, the policy is mapped to a guard including the condition and joined with existing behaviour using a logical `OR` as follows:

```
Roles {
Import RPC;
Roles {
  ...
  Supplier {
    loop {
      invoiceDate {RPC.Client(()(d:Date))}
      -> [now()-prev.d > (29*24*3600)
          AND now()-prev.d <= (30*24*3600)]
          InvoiceSent}
    OR ...}}
Interactions {
  RPC(Supplier.invoiceDate,
      State.LastInvoice.invoiceDate)
  AND ...}
```

This specifies that the supplier retrieves the last invoice date from the `State` role using an RPC and can then execute the `InvoiceSent` behaviour any time in the period from midnight on the 29th day to the end of the 30th day after the last invoice date. The `OR` in the role behaviour says this behaviour is always allowed if the condition is met. Note the implication that BCL permitted behaviours with no condition are always allowed. The presence of conflicts between permissions and prohibitions or obligations in BCL will, however, result in the prohibition or obligation taking precedence.

3.9 Prohibition

A BCL *Policy* can have an *Prohibition* modality, indicating that the behaviour defined in the policy must not occur,

for example:

Policy: CreditLimitForPurchaser
Role: Purchaser
Modality: Prohibition
Condition: PurchaseOrder(
OutstandingDebt $>$ *CreditLimit)*

This is the BCL realisation of our credit limit constraint, requiring that the outstanding debt is less than the credit limit. In Finesse, this maps to a guard on the behaviour that specifies a negation of the BCL conditional expression in the prohibition. This behaviour is joined with other binding behaviour using a logical `AND`:

```
Roles {
  ...
  Purchaser {
    loop {
      balance {RPC.Client(()(balance:Real))}
      -> [NOT balance > CreditLimit] PO}
    AND ...}}
Interactions {
  RPC(Purchaser.balance,
      State.outstandingDebt.balance)
  AND ...}
```

Note here that we must use the `balance` RPC operation on the `State` role to obtain the current outstanding debt using our model for shared state. When a prohibition is unguarded, the policy is mapped to a `false` guard on the behaviour meaning that it cannot occur.

4 Discussion and Conclusions

The mapping presented in the preceding section provides an indication of the work required to compile business contract specifications into executable choreography primitives for control, monitoring and notification. It highlights a number of key issues in the contract and choreography semantic models, specifically:

1. Time and locality are critically important in the accurate specification of policies.
2. The choreography environment must support the generation of events to signal potential contract violations.
3. The use of contact language event semantics requiring reliable notification for event completion is not scalable or robust in the face of failure. Defining failure semantics in terms of local, distinct execution and notification receipt events can be used, but templates for common failure behaviour would be useful to reduce complexity.
4. In many cases, it is necessary to collect notifications of events from many sources to satisfy event and policy semantics. This can imply delivery of event notifications to more than one destination. The Finesse model handles this elegantly, but it could add significant complexity for choreography models based on explicit point-to-point messaging.
5. The concept of non-static shared state in a business contract must be translated into a set of interactions used to retrieve and optionally update that state. If the concurrency and durability properties of the state are important to the contract, then these properties must be explicitly specified so that an appropriate transaction model can be applied. In general, shared state is difficult to manage in a truly distributed environment and alternative mechanisms are preferable.

While these issues are problematic, the exercise has indicated that a translation is feasible. We did not strike any particular limitations in the transformation for this example and expect that the mapping presented will work in a relatively generic fashion for the features used. The usefulness of such translation needs to be explored further through more complex examples. There are also features of BCL that have not yet been explored and further work is required to develop a complete and generic mapping.

In this paper we have used BCL and Finesse as vehicles to demonstrate the conversion from business contract to process definition and they have proven to be quite suitable for the task. We have previously shown that it is possible to monitor and enforce a contract using a language-specific infrastructure like that presented for BCL in [22], but we envisage that such contract specifications will be used in environments based on standard process execution technologies like BPEL[29] and CDL[13] in future. This paper has provided an insight into the issues that need to be addressed, and has identified several key semantic concepts in BCl and Finesse that might contribute to future standardisation efforts like OASIS eContracts[16] and standard process technologies respectively.

5 Acknowledgments

The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Industry, Science & Resources).

References

[1] A. Berry. *Describing and Supporting Complex Interactions in Distributed Systems*. PhD thesis, University

of Queensland, 2002.

[2] A. Berry and S. Kaplan. Open, distributed coordination with finesse. In *ACM Symposium on Applied Computing*, Feb. 1998.

[3] A. Berry and S. Kaplan. A distributed asynchronous execution semantics for programming the middleware machine. In *Fifth International Symposium on Autonomous Decentralized Systems*, Dallas, Texas, USA, Mar. 2001. IEEE.

[4] BEA, Intalio, SAP, Sun publish web services choreography interface, 2002. Press Release.

[5] P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models*. Springer, 1996.

[6] Dicarta. http://www.dicarta.com.

[7] The OASIS ebXML standards. http://www.ebxml.org.

[8] A. Farrell, M. Sergot, M. Salle, C. Bartolini, D. Trastour, and A. Christodoulou. Performance monitoring of service-level agreements for utility computing using the Event Calculus. In *First IEEE International Workshop on Electronic Contracting*, July 2004.

[9] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, Aug. 1991.

[10] iMany. http://www.imany.com.

[11] ISO/IEC 10746-1 10756-2 10746-3 10746-4 Basic Reference Model for Open Distributed Processing.

[12] ISO/IEC is 15415 Open Distributed Processing-Enterprise Language, 2002.

[13] N. Kavantzas, D. Burdett, and G. Ritzinger, editors. *Web Services Choreography Description Language Version 1.0*. W3C, 2004. http://www.w3.org/TR/ws-cdl-10/.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer Verlag, June 1997.

[15] R. Lee. A logic model for electronic contracting. *Decision Support Systems*, 4(1):27–44, 1988.

[16] Oasis legalxml econtracts. http://www.oasis-open.org/committees/legalxml-econtracts/charter.php.

[17] P. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal. A unified behavioural model and a contract for extended enterprise. To appear in Data Knowledge and Engineering Journal.

[18] D. Luckham. *The Power of Events*. Addison-Wesley, 2002.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. part i and ii. *Information and Computation*, 100(1), 1992.

[20] Z. Milosevic. *Enterprise Aspects of Open Distributed Systems*. PhD thesis, University of Queensland, 1995.

[21] Z. Milosevic, A. Berry, A. Bond, and K. Raymond. Supporting business contracts in open distributed systems. In *Proceedings of the Workshop on Services in Distributed and Networked Environments*. IEEE, 1995.

[22] Z. Milosevic, S. Gibson, P. Linington, J. Cole, and S. Kulkarni. On design and implementation of a contract monitoring facility. In *The first IEEE workshop on E-contracting (WEC04)*. IEEE, July 2004.

[23] Z. Milosevic, P. Linington, J. Cole, S. Gibson, and S. Kulkarni. Inter-organisational collaborations supported by e-contracts. In *The IFIP I3E Conference*, Aug. 2004.

[24] S. Neal, J. Cole, P. Linington, Z. Milosevic, S. Gibson, and S. Kulkarni. Identifying requirements for business contract language: a monitoring perspective. In *Proceedings of EDOC2003*. IEEE, Sept. 2003.

[25] Oracle Contracts. http://oracle.com/appsnet/products/contracts/.

[26] O. Perrin and C. Godart. An approach to implement contracts as trusted intermediaries. In *First IEEE International Workshop on Electronic Contracting*, July 2004.

[27] A. Rakotonirainy, A. Berry, S. Crawley, and Z. Milosevic. Describing open distributed systems: A foundation. *The Computer Journal*, 40(8), 1997.

[28] Web service level agreements (WSLA) project. http://www.research.ibm.com/wsla/.

[29] Business process execution language for web services, May 2003. http://www.ibm.com/developerworks/library/ws-bpel/.