International Journal of Cooperative Information Systems
© World Scientific Publishing Company

EXTENDING CHOREOGRAPHY WITH BUSINESS CONTRACT CONSTRAINTS

ANDREW BERRY
<andyb@whyanbeel.net>
Brisbane QLD
Australia

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

Business contracts play a central role in governing commercial interactions between organizations. It is increasingly recognized that business contract conditions need to be closely linked to internal and external business processes, both to reduce the risk of contract violations and to ensure compliance with legislative regimes. Recent research has proposed contract languages allowing the specification of obligations, permissions and prohibitions in business contracts. Business processes that cross organizational boundaries can be specified in choreography and coordination languages but these do not provide appropriate abstractions for contract constraints. In this paper, we examine the transformation of contract constraints in a business contract language into expressions in a choreography language. An example cross-organizational process is presented, along with a specification of the process in a choreography language and a specification of a set of contract conditions for the process in a business contract language. The contract terms are then translated into choreography expressions that govern the process to ensure compliance. Subsequent discussion explores a number of business and technology issues related to the results. We conclude that cross-organizational business processes can be monitored and enforced according to business contract specifications through the transformation of a contract definition to constraints on process behavior.

Keywords: Choreography; Business contracts; Business events; Policy Modelling.

1. Introduction

Business contracts play a central role in governing commercial interactions between organizations. It is increasingly recognized that business contract conditions need to be closely linked to internal and external business processes in E-business applications, both to reduce the risk of contract violations and to ensure compliance with legislation[23,39,38]. Recent research has proposed business contract languages allowing the specification of obligations, permissions and prohibitions from business contracts[15,35,29,42]. These languages facilitate integration of business contract specifications with cross-enterprise interaction models. The aim is to allow specifications in such contract languages to be used more directly in the execution and monitoring of business processes. Contract languages provide domain-specific abstractions that enable organizations to precisely describe contract conditions. This precise capture of contracts supports the deployment of contract management systems for any cross-organizational process and any vertical domain. By comparison, the current commercial applications are predominantly document-oriented contract management systems for common business processes like procurement[11,17,34,40]. These systems are also concerned primarily with the perspective of a single organization.

In parallel with this work in support of electronic contracts, the maturity of technology for the co-ordination of activities or processes spanning multiple organizations is improving. The term *choreography* is being used to describe the description and implementation of processes without centralized control, typically spanning organizations[24]. Recent work in the W3C consortium is focusing on so-called *web services choreography* (WS-CDL)[18] and the OASIS consortium is working to ensure that the ebXML standards[14,12] are capable of describing and supporting choreography. The primitives in choreography languages have been guided by previous work on co-ordination languages[8,4] and the semantics of specification languages for distributed processes, particularly pi calculus[26]. While these languages provide abstractions for describing cross-organizational interaction, the abstractions are not aligned with the needs of business people describing contract constraints.

This paper examines how contract terms in business contract languages can be transformed into expressions in choreography languages. This is done in a concrete fashion by presenting a sample cross-organizational purchasing process, along with a specification of the process in the Finesse choreography language[3] and a specification of the contract terms for the process in Business Contract Language (BCL)[23,29]. The contract terms are then translated into Finesse expressions that govern the process to ensure compliance and identify contract violations. Such a translation between a business contract specification and a process definition illustrates how rules and policies in business contracts can be used in the design of processes for E-commerce and E-business platforms. We contend that translation from contract to process terms will be an important tool in electronically-enabled businesses of the future.

In the remainder of the paper, section 2 describes a purchasing process and defines a set of contract terms for that process in plain English. The Finesse and BCL technologies are introduced and used to specify this purchasing process and its contract terms respectively. We then describe a mapping from the BCL contract terms to equivalent Finesse language expressions in section 3, combining these expressions with the original Finesse process definition to produce a Finesse process augmented with contract constraints. The result is an augmented process definition that monitors and controls the execution to ensure that the contract terms are satisfied. This is followed in section 4 by a discussion of the issues uncovered, including the semantic differences between contract terms and choreography expressions and the need to choose between in-band and out-of-band mechanisms to ensure contract compliance. Since the paper is intended to highlight issues in transforming business contract languages with choreography in the general case, section 5 identifies technologies similar to BCL and Finesse and discusses their strengths and weaknesses in supporting this transformation.

2. Finesse and BCL

This section informally describes a purchasing process involving a purchaser, supplier and freight company. A contract governing that process is presented in English, and the need for monitoring and enforcement of that contract justified. The Finesse and BCL technologies are described, and these informal process and contract definitions are presented in Finesse and BCL respectively. These code examples are used to drive the transformation described in section 3.

2.1. *Informal process definition*

The purchasing process is summarized as follows:

(1) The purchaser places a purchase order.
(2) The supplier receives a copy of the purchase order.
(3) The supplier fills the order and notifies the freighter that it is ready for shipment.
(4) The freighter collects the goods from the supplier and delivers them to the purchaser.
(5) The supplier invoices the purchaser for the goods delivered
(6) The purchaser pays for the goods.

Note that only activities relevant to the cross-enterprise process are identified in this description: internal activities not visible or relevant to the collaboration are omitted. This definition of externally visible activities and their relationships is the domain of choreography definition languages like Finesse and WS-CDL. Business contracts are similarly defined in terms of the externally visible activities and their relationships.

2.2. *Plain English contract*

The contract for the purchasing process involves three parties: purchaser, supplier and freighter based on the process definition presented in the preceding section. The following terms apply to the contract:

(1) The purchaser has an agreed credit limit with the supplier. 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 an invoice immediately after goods delivery.
(3) The purchaser is obliged to pay the balance of the invoice 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 being notified that the shipment is ready.

2.3. *The need for contract monitoring and enforcement*

The process definition of section 2.1 describes an expected sequence of activities in the cross-organizational, collaborative business process. These are intended to be governed by the contract defined in section 2.2. There are many situations in the execution of business activities that could lead to the violation of policies stated in the contract. On one hand, a party to the contract might choose not to fulfill their contractual obligations because they can obtain greater value through involvement in some other collaboration, despite penalties for a breach. On the other hand there might be circumstances beyond the control of any party to the contract that can lead to violations of contract conditions, for example communication failure. In order to protect parties to the contract, there needs to be monitoring and enforcement mechanisms associated with the process to ensure the fulfillment of obligations and other policies, preferably in real-time. Subsequent sections introduced the Finesse and BCL languages then demonstrate how BCL can be used to specify contract conditions, and how Finesse can be used to define the business process and embed monitoring and enforcement mechanisms based on the BCL contract specification.

2.4. *Finesse*

Finesse is a system that provides an operational semantics, language and execution engine for co-ordination of behavior across distributed, autonomous participants[3]. The concepts embodied in these elements are taken from the A1$\surd$ architecture model[5]. This model describes distributed computations in terms of the autonomous objects or participants that execute local behavior, and a binding that defines the interactions between visible behaviors of participants. A binding can have pre-defined semantics, for example remote procedure call, or can be programmable. This use of the term binding is also present in the ISO/IEC Reference Model for Open Distributed Processing (RM-ODP)[36]. The Finesse language is a high-level language for programming bindings, and the the operational semantics formally defines a general model for the execution of bindings in a distributed environment with no shared state.

The Finesse language is used in the remainder of this paper to describe processes and choreography and the term Finesse is used in those sections to refer to the language rather than the execution engine or operational semantics. The following subsections provide an overview of the language syntax and semantics, and the use of this syntax is illustrated by defining our example process in section 2.5. In the following text, Finesse samples are highlighted through the use of a fixed-width font like `this`.

The Finesse language is perhaps best described as a coordination language and a description of the language has previously been published in that research sphere[4]. The syntax and structuring concepts of the language, however, have grown from the needs of open distributed processing[5]. The language is intended to describe the behavior and coordination of autonomous software components in an open, distributed environment. As such, it bears only a minimal resemblance to existing programming language syntax, including interface definition languages like CORBA IDL or WSDL (Web Services Definition Language).

A Finesse program or `Binding` is defined in terms of a set of roles capturing visible behavior of participants and a set of interactions, defining the interactions between roles. Both role and interaction behavior is defined in terms of events and their causal, temporal and parameter relationships. Finesse has some key characteristics that distinguish it from other languages:

(1) Finesse abstracts over communication, allowing transformation of data and compiler or run-time optimization of message passing between components.
(2) Finesse includes a representation of time, allowing the specification of temporal properties.
(3) Finesse is independent of the programming language and infrastructure used to implement distributed components. Participants in a Finesse `Binding` could be, for example, process engines, custom software components or web services.
(4) Finesse provides an abstract model for multicast and other group behavior through roles having non-unary cardinality.
(5) Finesse explicitly distinguishes between co-located and distributed behavior through separate specification of roles and interactions respectively. It uses a common semantic model for those behaviors, however.

2.4.1. *Basic Syntax and Structure*

A Finesse binding has an outer scope introduced by the keyword `Binding` and the name of the binding. This outer scope defines the program boundary: all behavior present in the binding must be described in this scope. A set of `Import` statements can appear at the beginning of this scope. An `Import` statement identifies another binding program whose behavior definitions can be referenced and re-used in this binding.

This opening is followed by two sections defining roles and interactions. The `Roles` section defines the required behavior of participating components, and the `Interactions` section defines the relationship between events at different roles. Braces ({...}) are used to delimit the scope of definitions. Note that in the following examples, ellipses (...) are used to avoid including unnecessary detail and are not a syntactic construct. The basic structure is thus:

```
Binding Example {
  Import ...;
  Roles {
    ...
  }
  Interactions {
    ...
  }
}
```

2.4.2. *Describing Roles*

A binding has one or more role definitions in the `Roles` section, each introduced by a role name. A role definition can be prefixed with a cardinality constraint enclosed in square braces `[]`, which constrains the number of participants that can fill a role in a single binding instance (process execution). The place-holder `#` represents the actual cardinality. Where no cardinality constraint is given, the default cardinality is exactly one, for example:

```
Roles {
  Client { ... }
  [#>=1] Server { ... }
}
```

This specifies that there are two roles, *Client* and *Server* and that there is exactly one *Client* participant and at least one *Server* participant in the binding.

Roles define local or co-located behavior. Behavior described within a role takes the form of event relationship specifications. In terms of the Finesse semantic model[3], these specifications connect event template definitions (see 2.4.4) using a number of relationship operators and modifiers. These operators and modifiers are discussed further in section 2.4.5. Where multiple participants can fill the role, each participants executes the described behavior independently except where linked by specifications in the `Interactions` section.

Roles can contain named behaviors that group together a set of event templates and allow the `Interactions` section to refer to some subset of the role when defining interaction behavior, for example the `read` and `write` behaviors in the following code fragment:

```
Binding {
  Roles {
    Client {
       read { ... } AND
       write { ... }
    }
    ...
  }
  Interactions {
    Client.read ...
  }
}
```

Named behaviors define a scope for event template names, allowing the role definitions to re-use template names in a different context. Reference to such event templates in the interactions section must include the role and any scope names, e.g. using the example above `Client.read.send`.

2.4.3. *Describing Interactions*

The `Interactions` specification defines relationships between event templates occurring at distinct role instances, that is, behavior that spans locations and implies messaging. In the roles specification, event templates are introduced with a name and a direction indicator. In the interactions specification, event templates are referred to by the role name, followed by a period '.', then the event template name. This reference to an event template can also have a cardinality constraint to deal with situations where multiple components fill the role. For example:

```
Binding Example {
  Import ...;
  Roles {
    Client { send! }
    [#>=1] Server { receive? }
  }
  Interactions {
    Client.send -> [#=all] Server.receive
  }
}
```

The place-holder `#` in the `Interactions` specification refers to the number of components executing the event template, while the place-holder `all` refers to the number of components filling the role in the binding instance. In the above example, the client role executes a `send` event followed by all servers executing the `receive` event. This binding is a high-level description of reliable multicast. Behavior described in the interactions section cannot introduce new event templates: it can only use event template names defined in the roles section.

2.4.4. *Event Templates*

The behavior within roles and interactions is defined by event templates and their relationships. An event template describes the essential characteristics of an event that can be executed at runtime, or in other words, a set of declarative constraints on the event execution. Events in Finesse are considered to be atomic, immutable and instantaneous occurrences with a specific location in both space and time.

An event template is introduced in the roles section by a name, a direction indicator, and a parameter list, for example:

```
e!(x:t1, y:t2)
```

where `e` is the event template name, `!` indicates that it is an output event, `x,y` are the event parameters, and `t1,t2` are the data types of the parameters. Events are uni-directional, that is, they can be input events or output events but not both. The `?` character is used in place of the `!` to indicate an input event. Input and output are relative to the role, that is, an output event implies that the component filling the role is providing parameter values, while an input event implies that binding program execution is providing parameter values.

The direction indicator and parameter list are only included in the first specification of an event template. This means they can only appear in role definitions. Where a template name appears more than once in a role definition or named block (nested context), only the first can include these annotations.

2.4.5. *Behavioral Model*

Event relationships provide the basis for describing behavior in both the role and interaction definitions of a binding. Event relationships capture the dependencies between event templates in a binding. Three distinct types of event relationship are identified:

Causal relationships which describe the causal dependencies between events;

Parameter relationships which describe the relationships between parameters of causally related events. Parameter relationships imply the content of messages passed between interacting components, but in a declarative, application-oriented manner;

Timing relationships which describe any real-time relationships between events. These relationships can be used to describe, for example, timeouts or quality of service requirements of interactions.

Relating these back to our purchasing process, causal relationships capture process flow, parameter relationships capture data flow, and timing relationships allow us to express temporal constraints that affect the process. These event relationship concepts, combined with the notions of *binding*, *role* and *interactions*, provide a powerful technique for the description of processes and choreography.

Causal relationships are defined using the `->` operator between event template names as used in both preceding and following code fragments. Parameter relationships are defined through declarative specification of event parameter values as illustrated in the following code fragment:

```
e1!(x:t1, y:t2) -> e2?(z:t3) {z = f(e1.x)}
```

This specifies that the parameter `z` of event `e2` is assigned the value resulting from applying the function `f` to the value of `e1.x`. Parameter relationship specifications can refer to any parameter of an identifiable, causally preceding event. There is no requirement that all parameters of any output event must be consumed by an input event, and the parameters of an output event can be used many times. Due to its common use in remote procedure call, Finesse has shorthand syntax for name equivalence of parameters, that is:

```
e1!(x:t1, y:t2) -> e2?(x:t1, y:t2) {*= e1}
```

This specifies all parameters of `e2` with names matching parameters in `e1` are assigned the value of that same-named parameter. There is no requirement that all parameters in either `e1` or `e2` be assigned by the operator.

Temporal constraints are introduced through an explicit time attribute of all events indicating the local time of execution. These time attributes can be referenced in guards as discussed in the following subsection.

2.4.6. *Guards*

A guard is a logical expression that must be satisfied before an event can be executed. This is in addition to any cardinality or causal predecessor constraints. They are introduced using the following syntax:

```
[guard] e1!(x1, y:t2)
```

The guard is a logical expression and can refer to parameters of causally preceding events in the same manner as parameter relationships. Where an event template has both a guard and a cardinality constraint, they must be contained within the same square braces and joined by a logical AND operator.

2.4.7. *Control Expressions*

Finesse supports three logical operators that can be used to join behavior expressions, namely `AND`, `OR` and `XOR`. These operators have semantics consistent with their use in other process logic. The language also has two looping expressions introduced by the keywords `while` for guarded iteration and `loop` for unguarded iteration.

2.5. *The purchasing process in Finesse*

The purchasing process defined in section 2.1 is presented below in the Finesse language. Note that the keyword `prev` is used to identify the immediately preceding event template in the specification below. This shorthand is included in Finesse as a syntactic convenience because many parameter relationships refer to the immediately preceding event template.

```
Binding PurchasingProcess {
  Roles {
    Purchaser {
      purchaseOrder!(o:EdifactPO) -> goodsDelivered?()
        -> invoiceReceived(i:EdifactInvoice)
        -> payment()
    }
    Freighter {
      goodsReady?() -> pickupGoods!() -> deliverGoods!()
    }
    Supplier {
      receiveOrder?(o:EdifactPO) -> fillOrder!()
         -> orderFilled!()
         -> deliveryConfirmed?()
         -> invoicePurchaser!(i:EdifactInvoice)
         -> paymentReceived?()
    }
  }
  Interactions {
    Purchaser.purchaseOrder ->
      Supplier.receiveOrder{o = prev.o} AND
    Supplier.orderFilled -> Freighter.goodsReady AND
    Freighter.deliverGoods -> Purchaser.goodsDelivered AND
    Freighter.deliverGoods  -> Supplier.deliveryConfirmed AND
    Supplier.invoicePurchaser ->
      Purchaser.invoiceReceived{i = prev.i} AND
    Purchaser.payment -> Supplier.paymentReceived
  }
}
```

This Finesse binding program describes the three roles in our example, capturing the local processes through causal relationships between event templates. The interactions between the those processes are also captured using causal relationships. Note that the use of a logical `AND` in the interactions to compose the event relationships can also be used in the same manner to compose event relationships in a role definition.

2.6. *BCL*

This section briefly describes each of the major constructs in BCL[23,29] through text and sample code fragments. The concrete syntax of BCL is an XML dialect. In this paper we use an English pseudo-syntax for readability.

2.6.1. *BCL overview*

BCL has been developed for the specification of contract conditions so that contract execution can be monitored against these conditions. This contract execution ultimately refers to various activities of the signatories to the contract (and possibly their agents). The monitoring is carried out in an event-based fashion and its aim is to check whether these activities signify fulfillment or violation of policies agreed in the contract.

Key characteristics of BCL are:

(1) BCL is a language providing abstractions related to the domain of business contracts. In this respect BCL is similar to other languages specifically developed to capture the semantics of specific business domains such as business processes, business documents and so on.
(2) BCL syntax closely resembles natural language expression of contracts, namely the expressions of deontic constraints[21,41] such as obligations, permissions and prohibitions. The BCL semantics is defined in terms of event pattern matching, which can include expressions of state changes, temporal constraints and event creation rules. This semantics is similar to that used in complex event processing[25].
(3) BCL is declarative language allowing domain experts to express contract conditions and the system interprets these expressions to determine how contract conditions should be monitored.
(4) BCL is an event-oriented and policy-based language where individual events or their combinations are used as a basis for determining valid set of contract conditions and these in turn determine the fulfillment or violations of policies.

Although the primary purpose of BCL was to support the expression of contract monitoring conditions, its concepts could be used for describing contract semantics for other contract automation purposes such as reasoning about contract structure and behavior. BCL can be thought of as an aspect-oriented language[19] that expresses business policies for processes.

BCL is an event-oriented language where an event is *something that happens*[36]. A single event can be used to signify an action of a signatory or some change arising from the environment that has relevance to the contract. This could be, for example, a change in regulatory or market conditions or a temporal occurrence such as a deadline. An event can also be generated within the contract management system to facilitate other monitoring or enforcement activities: this is known in BCL as an event creation rule. An event is characterized in BCL by a type definition. This definition includes:

- common parameters in its header, such as the timestamp when it was generated and the timestamp when it is ended, if it has duration;
- a list of its causal predecessors (if any); and
- a body which contains information about what occurrence this event signifies, for example, a purchase order document.

The following code fragment, for example, specifies that a purchase order event is signified by the existence of an XML document conforming to the EDIFACT PurchaseOrder XML schema. In this and subsequent code fragments, BCL terms are highlighted through use of italicized courier font like *`this`*. Ellipses (`...`) are used to avoid including unnecessary detail and are not a syntactic construct. We also use descriptive tokens enclosed in <> to identify the nature of the content without specifying detail.

```
Event typeID = PurchaseOrder
  HeaderInfo
    TimeStampStart
    TimeStampEnd
    ...
  BodyInfo
    <XMLSchema for EDIFACT PurchaseOrder>
```

2.6.2. *Event Pattern*

A BCL *event pattern* is a means for describing a state of affairs of significance for the monitoring of contract constraints. A state of affairs captures both the effects of actions of parties involved in contract-related activities and environmental effects such as change in regulatory policies. The effects are captured using the concept of event as described in the previous section. A state of affairs can range from elementary conditions, such as the occurrence of a particular action performed by a party or the passing of a deadline, to more complex conditions, such as "more than three system failures in any a one week period" or "one of the contract conditions has been violated". More specifically, event patterns express relationships between events and properties of events. As events occur, they *activate* event patterns, with the aim of determining whether a specific relationship between events (state of affairs) has ocurred. The primary use of event patterns in BCL to define policy checking behavior as described in section 2.6.4 below. Examples of event relationships are:

(1) Logical relationships between events, specifically `AND`, `OR` and `NOT`.
(2) Temporal relationships between events, for example `before` and `after`.
(3) Temporal constraints on event patterns, for example absolute and relative deadlines and sliding time windows.
(4) Event causality, including causality implied by temporal relationships and explicit definition of causal relationships.
(5) Certain special kinds of single events, for example contract violation and state change events.

The most primitive event pattern is a singleton event pattern. Individual events are matched by event type and possibly some additional constraints on attributes of the event. A singleton event pattern is thus:

```
Event typeId = PurchaseOrder
```

A singleton event pattern in BCL can also involve multiple `EventRoles` using the following syntax:

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

Event roles provide a mechanism for abstraction and re-use. They abstract over synchronization behavior, specifying that the identified roles must synchronize on the nominated event. They can also be re-used in multiple contexts. For example in the code above, we can infer the condition that the event matches both the sending and receipt of a message in a process and that the pattern will match only if the sender has the *Buyer* role and the nominated receiver has the *Seller* role. In business modelling, this construct provides a re-usable approach to checking the association of event roles with community roles (see 2.6.6), for example, to ensure that role-based security policies are satisfied.

Event patterns matching multiple related events are used to identify more complex contract conditions such as temporal dependencies, causal dependencies and parallelism. For example, an event pattern defining the maximum time between the *OrderFilled* event and the *PurchaseOrder* event in a purchasing contract looks like:

```
OrderFilled before (PurchaseOrder add 5 days)
```

A further example presents a more complex event pattern specifying how to identify payments received within an acceptable timeframe:

```
Payment before (Invoice add 30 days)
OR
(Payment after (Invoice add 30 days)
   AND NOT HasOccured(OverDuePayment,
     Range(LowerBound(Now() less 6 months),
           UpperBound(Now()))))
```

In this case, the normal behavior is to pay within 30 days of the invoice, but we have allowed an overdue payment to occur at most once in a six month period.

2.6.3. *Event Creation Rule*

BCL allows an event pattern to be abstracted through an event creation rule (*ECR*). An event creation rule defines a new event that is created when an event pattern is matched. This can simplify policies and support re-use, allowing policies to refer to a single composite event. This definition of composite events is similar to the event-condition-action paradigm from active databases[43].

2.6.4. *Policy*

A BCL *Policy* defines behavioral constraints for the roles that execute activities associated with a contract. Policies are checked by comparing the current state of affairs with these constraints. The constraints are defined through identifying a role, a modality and a condition expressed as an event pattern. Thus, BCL policy checking consists of matching event patterns in a system and determining whether they satisfy the policies. Note that one event can activate the checking of multiple policies.

The modality of a policy is expressed using the deontic modalities of *obligation*, *permission* and *prohibition*[41]. These modal constraints reflect their English-language meaning: obligations identify activities that must occur, permissions identify activities that are are allowed to occur, and prohibitions identify activities that must not occur. The role associated with a policy identifies the participant that is subject to the policy, or in other words *who* is obliged, permitted or prohibited. For example, if payment is not made then the *supplier* is permitted to charge interest on the outstanding amount. There can be dependencies between policies, for example, the violation of one policy can activate another policy that expresses remedial actions. The syntax of policies is illustrated in the following fragment:

```
Policy: FillPuchaseOrder
  Role: Supplier
  Modality: Obligation
  Condition:
    OrderFilled before (PurchaseOrder add 5 days)
```

This policy specifies that the supplier is obliged to have the goods ready for shipment within 5 days of the purchase order.

Policies are defined in the context in which they apply. This context is a `Community`, as will be explained in section 2.6.6. Relating these concepts to our contract monitoring and enforcement problem, the terms of a contract can be expressed in BCL using policies: a contract consists of a set of policies that apply to the behavior of signatories to the contract.

2.6.5. *State*

It is often the case that contract related events can change the state of certain variables associated with the contract. BCL introduces the notion of a `State` variable to define an updateable data value shared by participants in a `Community` (see 2.6.6). State variables can be used, for example, to maintain running totals, counters and other data values required to evaluate policies. The value of a state variable can be either determined explicitly in response to an event, or on request when the state value is needed. A contract can have many state variables that are changing to reflect the corresponding events.

The definition of a state variable identifies a set of update actions or calculations triggered in response to corresponding event patterns, as illustrated in the following example:

```
State: OutstandingDebt
  CalculationExpression
    UpdateOn: Payment
    UpdateSpecification:
      return (this less Payment.Amount)
  CalculationExpression
    UpdateOn: GoodsDelivery
    UpdateSpecification:
      return (this add InvoicePurchaser.Amount)
```

This defines the outstanding debt of a purchaser, which is updated whenever an order is delivered and whenever a payment is made. State variable changes are bound to event patterns and are deterministic, that is, the value of a state variable can only be modified through the matching of visible event patterns.

2.6.6. *Community*

BCL introduces the concept of a *community* based on the ODP community concept[36]. A community is a context for the specification of entities that collaborate to achieve the specified goal. A BCL community definition includes a set of roles, policies that apply to the roles, state variables and related event patterns that are used as part of policy expressions. One entity can play multiple roles, although this can be constrained by policies such as separation of duty. A community can also be regarded as an entity that can fill a role of some higher level community.

A community is introduced with the `Community` keyword followed by the community name and an identifier. It is typically followed by role definitions and static (immutable) values significant to the contract, for example:

```
Community:
  PurchasingContract Id = ABCD

Value: StartDate
Value: EndDate
Value: CreditLimit

Role: Purchaser
Role: Supplier
Role: Freighter
```

Note that the concept of a community is a general concept for describing collaboration and can be used as a basis for defining various organizational structures such as a company, supply chain, extended enterprise and so on. One specific kind of community is a business contract as discussed in this paper.

It is worth noting that we apply an event-based paradigm to support both the management and evolutionary aspects of a community, in a similar manner to the autonomic computing paradigm[2]. For example, any attempt to change the structure or composition of community can be treated as an event and such an event can in turn be the subject of a separate policy, that is, the management policy. This allows the possibility of changing the structure of a community in a controlled way as the corresponding management policies allow. There are many issues, however, that require caution when trying to dynamically change the structure of community and its constituent parts. Examples include possible policy conflicts that could arise, creation of inconsistent paths in business processes and timing issues in a distributed context (when does the new definition become active). These issues need to be addressed as part of the underlying architecture as is done, for example, in the Business Contract Architecture (BCA)[27].

2.7. *BCL for the purchasing contract*

Specification of a contract in BCL uses the definition of the expected business process as a starting point, for example, that presented in section 2.1. With the english language contract as reference, we then identify:

- Key activities in the business process specification that are subject to contract terms using BCL `event patterns`. In our example these are represented as singleton event patterns, for example, `PlaceOrder` or `GoodsDelivered`;
- Fixed data items(`Values`) necessary for the definition of contract terms, for example, the purchaser `CreditLimit`;
- Variable data items (`State`) necessary for the definition of contract terms, for example, the current `OutstandinDebt` of the purchaser; and
- Temporal conditions defining the times at which certain contract conditions are to be checked, for example, a delivery deadline of five days. In this example all the points in time are expressed relative to events in the behavior.

The contract terms are specified as a set of policies, namely obligations, permissions and prohibitions, expressed in terms of the activities and state identified above. It is important to note that the context of the contract specification extends beyond a single purchasing binding instance. For example, the `OutstandingDebt` state is the summation of unpaid invoice values across all purchasing binding instances. Thus dynamic information about each active contract will often need to be stored in a manner that is independent of binding execution.

The BCL contract definition for the contract terms specified in 2.2 are defined as follows with the specific contract terms highlighted in the code through dashed comment lines.

```
Community:
  PurchasingContract id: ABCD

  InitialisationSpecification: PuchaseContractCreationEvent

  ActivationSpecification: StartDate

  Value: StartDate
  Value: EndDate
  Value: CreditLimit

  Role:  Purchaser
  Role:  Supplier
```

```
 Role:  Freighter
----------------------- policy 1 ------------------------------
 Policy: CreditLimitForPurchaser
   Role: Purchaser
   Modality: Prohibited
   Condition:
     PurchaseOrder ((OutstandingDebt add PurchaseOrder.value)
                     greaterthan CreditLimit)

 State: OutstandingDebt
   CalculationExpression
     UpdateOn: Payment
     UpdateSpecification: return (this subtract Payment.amount)
   CalculationExpression
     UpdateOn: InvoicePurchaser
     UpdateSpecification:
       return (this add InvoicePurchaser.amount)
----------------------- policy 2 ------------------------------
 Policy ProvideInvoice
   Role: Supplier
   Modality: Permitted
   Condition:
     (InvoicePurchaser after Freighter.DeliverGoods)
----------------------- policy 3 ------------------------------
 Policy: PromptPayment
   Role: Purchaser
   Modality: Obliged
   Condition:
     Payment before (InvoicePurchaser add 30 days)
----------------------- policy 4 ------------------------------
 Policy PromptOrderFulfillment
   Role: Supplier
   Modality: Obliged
   Condition:
     OrderFilled before (ReceiveOrder add 5 days)
----------------------- policy 5 ------------------------------
 Policy: PromptDelivery
   Role: Freighter
   Modality: Obliged
   Condition:
     DeliverGoods before (OrderFilled add 5 days)
```

3. Mapping BCL to Finesse

This section defines a mapping of BCL concepts to Finesse expressions. We begin by discussing high-level mapping issues in 3.1 and options for enforcement and monitoring in 3.2, then describe the transformation of specific concepts from BCL to Finesse in 3.4 through 2.6.3. The mapping is illustrated by the purchasing example, with fragments of language expressions introduced progressively and the mapping discussed alongside these fragments. Along with text and code fragments describing the transformations, we depict key relationships between relevant fragments of BCL and Finesse meta-models using a simplified version of the approach proposed by Akehurst et al[1]. A UML <<relation>> stereotype is used to identify transformation relationships between model elements in BCL and Finesse. In terms of generating an augmented Finesse specification, these relationships define the additional elements that will be added to the original Finesse specification to monitor contract conditions. The section concludes by illustrating the mapping in a complete definition of a Finesse purchasing binding augmented with contract controls in 3.12.

3.1. *High-level Mapping Issues*

Finesse and BCL are technologies emanating from the Distributed Systems Technology Centre (DSTC), both directly and through sponsored PhD research programs[3,27]. They were influenced in the early stages by the participation of DSTC in RM-ODP[36] standardization. The influence has been continued in BCL through DSTC involvement in ODP enterprise language[37] standardization, and collaboration with the University of Kent[23]. This common heritage provides us with an opportunity to explore the transformation between contract languages and choreography languages without the effort required to resolve significant semantic differences that might have occurred for solutions derived from entirely distinct approaches. The key issues lie primarily in the need to map a domain-specific language (BCL) onto the computation-oriented semantics of the Finesse language. More specifically, we need to determine how the specification of policies as stated in BCL can be expressed through behavioral constraints on activities in Finesse to ensure that expected behavior in a Finesse execution of a binding satisfies those policies.

There are a number of similarities between BCL and Finesse. Both languages are event-oriented and both languages use roles to distinguish participants and model notions of autonomy and responsibility. Roles provide the basis for defining geographic distribution, although this is more explicit in Finesse than in BCL. A feature that distinguishes both languages from other languages and notations is the explicit support for cardinality of roles, that is, the ability to have multiple participants fulfilling the same role. Both languages also define an explicit context for interaction behaviour, that is a `Binding` in Finesse and a ***`Community`*** in BCL.

There are also a number of differences between the languages, primarily in intent and abstractions. BCL was developed with the intent of checking constraints on business-level behavior and defines the behavior of a monitor observing events of relevance to a contract in an *out-of-band* manner as depicted in figure 1. BCL uses the abstractions of the ODP Enterprise Viewpoint[36]. On the other hand, Finesse was developed with the aim of specifying distributed processes spanning autonomous components using abstractions similar to that of the ODP Computational Viewpoint[36]. The Finesse language describes expected behavior at component in-

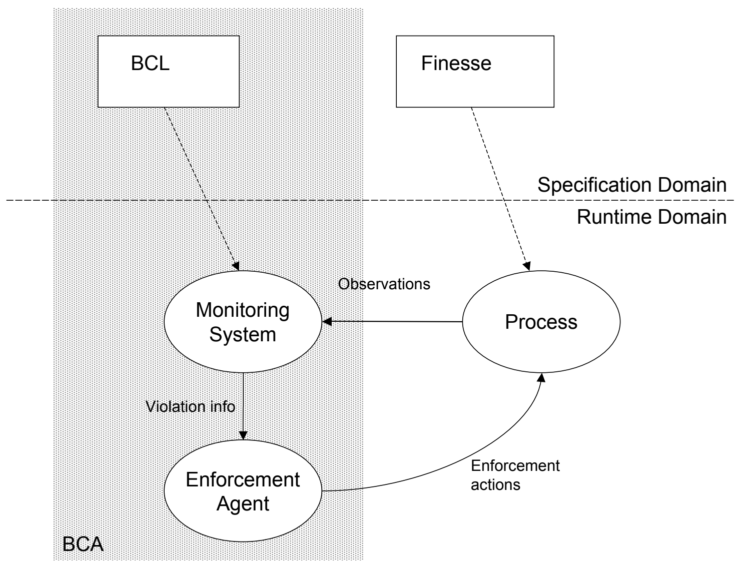

Fig. 1. Out-of-band monitoring and enforcement

terfaces (roles) and the relationships between those behaviors (interactions). The implication is that Finesse is more suited to defining *in-band* monitoring and constraints as discussed in more detail in section 3.2.

Another key difference is one of scope. A contract has a scope that will often span many internal processes, for example, a supplier might have separate processes for supply and debt collection governed by a purchasing contract. Conversely, a process could be governed by many different contracts depending on dynamic factors, for example, a purchaser might have distinct contracts with each supplier but use the same process, or might be subject to internal policies and statuatory obligations that span all activities of the enterprise. In this paper, we focus on the semantic mapping between policies in a single contract and behavior in a single process related to that contract but consider the implications of the many-to-many relationship between contract and process in choosing an appropriate mapping.

It should be noted that the differences in abstraction and scope run deeper than just the mapping between BCL and Finesse. Business-level constraints are inevitably more abstract and must be amenable to changes reflecting the dynamic nature of the business environment. The implication for any runtime environment is that change is a necessary part of any process and that flexibility in the runtime environment is essential. We do not directly address the semantics of process change but identify key areas where change can and should be supported.

3.2. *Contract Monitoring and Enforcement*

Previous approaches to contract implementation and enforcement based on BCL[27,28,29] have used an approach based on out-of-band monitoring of processes to detect violations. Similarly, a separate component was used to implement enforcement mechanisms. In other words, an external contract monitor examines a stream of events that capture the activities in a process and identifies behavior violating the contract terms, and these violations are reported to a separate enforcement agent like the BCA *Contract Enforcer*[30]. This approach is depicted in figure 1.

The mapping of contract terms to choreography language expressions provides the opportunity to enforce contract compliance in several alternative ways, notably:

(1) Prevention of non-compliant behavior, that is, the choreography engine implicitly monitors and controls the local processes of each participant and does not allow execution of activities that violate the contract, as depicted in figure 2. It terminates the binding in cases where failing to perform an activity violates the contract. Note in these figures that the box labelled *P-Finesse* refers to a new Finesse binding definition augmented with the policy constraints.

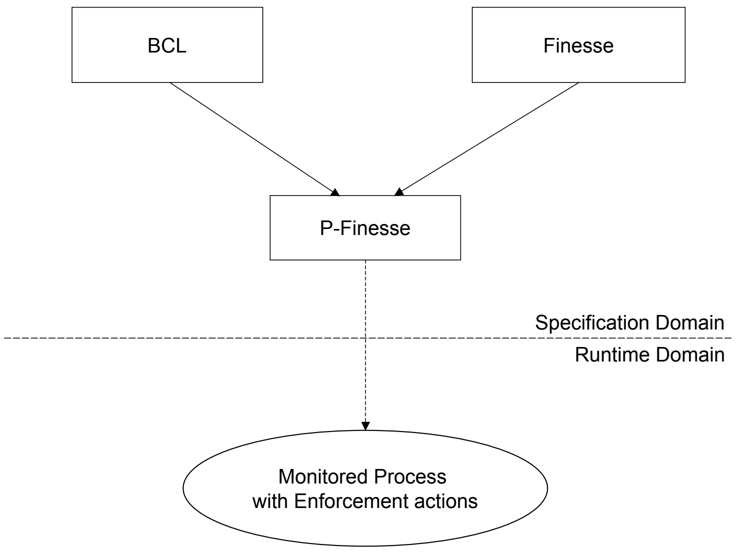

Fig. 2. In-band monitoring and enforcement

(2) In-band monitoring and enforcement, that is, the choreography engine immediately identifies non-compliant behavior and begins the execution of explicitly-

defined enforcement behavior for handling non-compliance. This could include, for example, terminating the binding or automatically applying financial penalties. This is conceptually equivalent to the previous approach, except that a distinct role and additional behavior is defined for enforcement.

(3) In-band monitoring with external enforcement, that is, the choreography engine identifies non-compliant behavior and notifies an external contract enforcement agency to begin contract enforcement processes. This hybrid approach is depicted in figure 3.

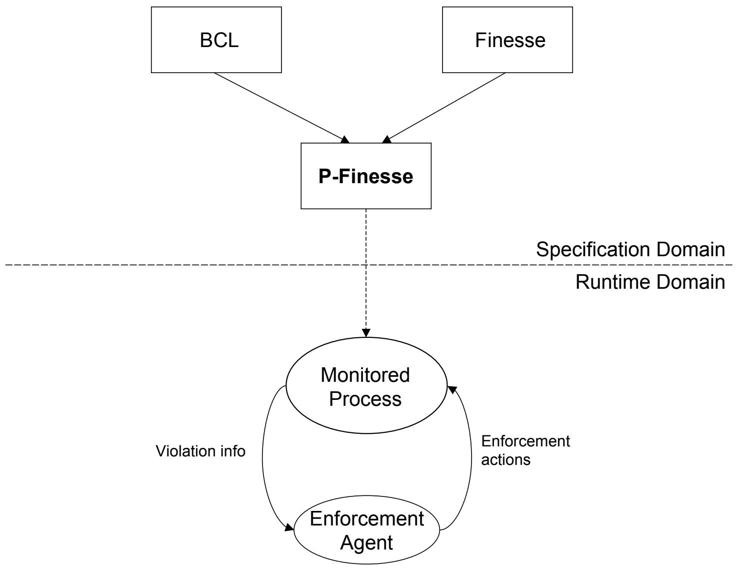

Fig. 3. In-band monitoring with out-of-band enforcement

Any combination of the original approach and these alternative approaches is possible and often desirable, thus providing increased flexibility in ensuring contract compliance. For example, service-level contract violations with small financial penalties might be automatically applied using in-band enforcement mechanisms, with other breach-of-contract issues dealt with using an external monitor and enforcer. There are also differing levels of enforcement as identified for BCA[30], and the in-band approach to enforcement typically corresponds to the notion of *non-discretionary* enforcement, that is, enforcement rules that can and will be applied regardless of the circumstances. *Discretionary* enforcement is often required to provide more flexibility in dealing with violations.

In this paper we have chosen to implement the first approach, that is, prevention of non-compliant behavior. This requires the most comprehensive mapping from BCL to Finesse behavior and also demonstrates that it is feasible to use the Finesse engine purely as an out-of-band monitoring engine. That is, existing process execution mechanisms are used to implement processes and a stream of events from processes is fed into the Finesse engine to monitor for contract violations.

3.3. *Implementing enforcement in Finesse*

Enforcement behavior is not currently defined either in the Finesse binding definition or in the BCL contract definition. For previous implementations of BCL, enforcement was implemented by a separate contract enforcement agent [30,27]. The decision to implement in-band controls in Finesse means that the enforcement behavior in the case of contract violations must be specified. Any enforcement behavior is possible, including the continuation of the binding with alternate constraints. Such behavior can be specified explicitly in the contract and mapped to Finesse using the behavior mappings specified in this section. For the purposes of the example, we will use the following approach to enforcement:

- a violation will result in the termination of the binding; and
- checks for violation will occur before proceeding with any subsequent behavior in the binding, or in other words, once a violation has occurred, no further behavior will be possible.

We will use a subordinate Finesse binding defining termination semantics, specifically:

```
Binding Termination {
  Roles {
    [#=1] Initiator { terminate!(reason:String) }
    [#>=1] Receivers { terminate?(reason:String) }
  }
  Interactions {
    Initiator.terminate ->
      [#=all] Receivers.terminate {reason = prev.reason}
  }
}
```

This contract enforcement behavior is quite inflexible and it is likely that a real system would soften the behavior with specific mediation or other remedial behaviors. In fact, it is often a business-level requirement that these behaviors should be flexible and expression-driven to support changes in contracts and enforcement approach during the life of a contract. The termination behavior above should thus be considered a placeholder, allowing us to highlight the points in the binding at which contract enforcement is necessary.

3.4. *Community*

A BCL `Community` is very similar to a Finesse `Binding` specification: it is a context for the definition of roles, behavior and states of affairs of interest in the community. While these concepts are similar, the scope of a `Community` is somewhat different from a Finesse `Binding` as discussed in section 3.1. The implications for the mapping of this difference can be summarized as follows:

- Community constants (`Values`) must be available to all binding instances associated with the community;
- Community state variables (`State`) must be available to and updateable by all binding instances associated with the community.
- The set of communities whose policies apply to a binding must be determined prior to binding instantiation;
- Where names associated with events and roles differ between the community and binding definitions, an explicit mapping of names will be necessary; and
- There might be activities related to the community that are not realized or visible in a binding definition.

The mapping of a community to a Finesse binding is depicted in figure 5. For the purposes of our mapping, we apply the policies of a single community to a single binding with `Values` and `State` appropriately abstracted from the binding.

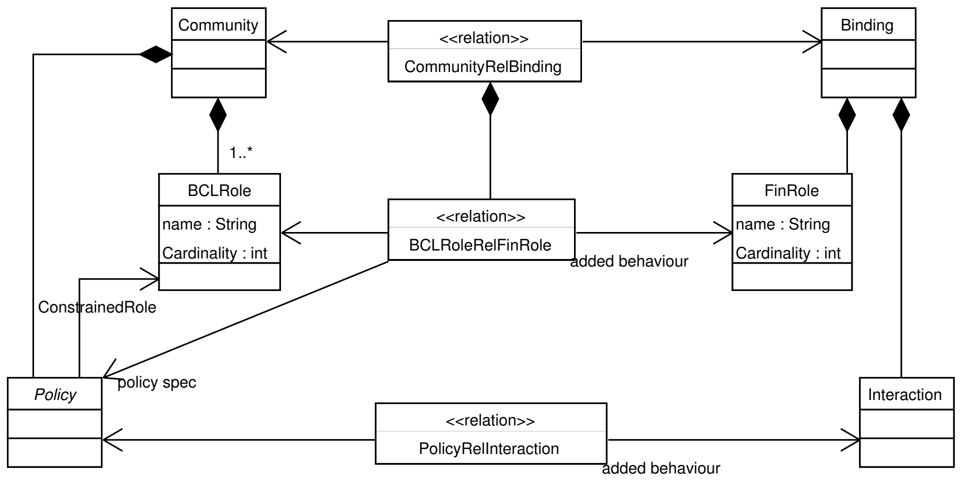

Fig. 4. Transforming community and roles

It is worth noting that the monitoring approach embodied in BCA[30] has a distinct notion of *activating* a community, which indicates the monitoring of the policies defined for a community should begin. Since we have chosen in-band monitoring for this initial mapping, we will assume that the community has been activated prior to binding instantiation and that monitoring is always required. More flexible approaches can be implemented if required, for example, providing an interface and appropriate guard conditions to enable monitoring on-the-fly.

3.5. *Role*

A BCL *`Role`* is used to identify behavior associated with a party to a contract. BCL roles are names, with the expected behavior of parties filling roles defined in subsequent *`Policy`* and thus the embedded *`EventPattern`* definitions associated with a specific *`Role`* name. The corresponding `Role` term in Finesse is similar: it defines the expected behavior of participants in a `Binding`. The key differences are firstly that Finesse `Role` definitions must contain a complete definition of the visible behavior associated with the role whereas BCL identifies only those behaviors that are significant to the contract. Secondly the Finesse role behavior is always local to the participant.

The approach to mapping BCL roles to Finesse roles is to treat each of the BCL policy statements as constraining or adding behavior to the Finesse roles. The augmented Finesse role definition becomes the logical `AND` of the behaviors corresponding to BCL policies and the existing Finesse binding definition. This ensures that the Finesse specification of expected behavior will also satisfy policies stated in the contract. So, the binding execution can enforce the rules stated in the policies of the contract. There will be cases, however, where the constraints expressed in a policy apply to interactions between participants. These must be mapped to constraints defined in the `Interactions` section of the Finesse program and joined with the existing interaction behavior as before using a logical `AND`. Examples of this mapping are illustrated subsequently in section 3.10 and the UML definition of this mapping is depicted in figure 4.

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 cardinality semantics are equivalent so must be compatible, specifically, the Finesse role cardinality specifications must not permit role cardinalities that are not permitted by the BCL specification. The approach to mapping behavior above applies in the same manner to roles with non-singular cardinality, that is, all parties fulfilling a role are subject to the policies associated with that role.

3.6. *Subcommunity*

BCL also supports the notion of subcommunities, where subcommunities can fill roles in higher-level communities. This provides modularity, visibility controls and separation of concerns in the language. There are two ways to implement subcommunities in Finesse:

(1) Through definition of a proxy that acts as a gateway between the distinct

bindings that correspond to subcommunity and the top-level community.

(2) Through flattening the community and subcommunity into a single Finesse specification, meaning roles of the subcommunity become explicit roles in the Finesse binding.

The first is most appropriate for subcommunities used to give visibility controls, for example, to define the internal enterprise behavior that implements a role in a cross-enterprise community. It is important to note that the use of a proxy resolves the issues associated with locality, remembering that events associated with a Finesse role must be co-located. The second is perhaps more efficient, but is typically inappropriate because it undermines the reason for introducing a subcommunity in the first place: we do not consider it further. Note that the Finesse `Import` functionality is a programming abstraction to support code re-use and it does not provide the semantics necessary to implement subcommunities.

The first mapping is depicted in figure 5. The behavior of a subcommunity fulfilling a role in a higher level community is encapsulated by a gateway object and a proxy role definition that identifies the behavior of the subcommunity that is exported to the higher-level community. Note that the gateway object depicted in figure model 5 cannot be directly modelled in the Finesse language syntax because the concept of participant or object is not present in the language. The use of two FinRole classes in this figure is a reflection of the Akehurst notation: this representation is used to explicitly distinguish the two Finesse role instances.

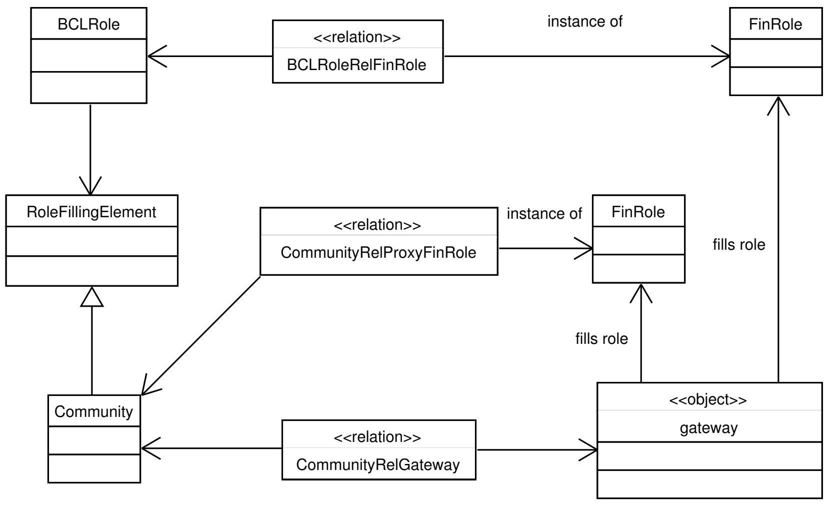

Fig. 5. Transforming subcommunities using a gateway

3.7. *State*

The BCL `State` construct defines data values of relevance to the contract and required to evaluate policy. This state is shared by the participants in the `Community` and defines a set of state update actions, triggered in response to corresponding event patterns, as illustrated in the BCL contract example of section 2.7. In the context of a BCL community specification, this state is global.

Finesse has no explicit state construct. State is implied by reference to causally preceding events in parameter relationships, or in other words, a partial view of the event history for a binding. This avoids any need for explicit synchronization between distributed engines but also forces a programmer or compiler to explicitly define the semantics of shared state if required. The value of any BCL shared state can thus be maintained in Finesse in three ways:

(1) By always retrieving and updating the state through an explicit role in the binding program.
(2) By defining the current value of the state in a parameter relationship as a function of the set of causally preceding state update events.
(3) Through explicitly defined state synchronization behavior, for example, use of an optimistic voting mechanism to maintain shared state across participants.

We will take the approach of defining an extra Finesse role responsible for persistent storage of each BCL state variable, reflecting option 1 above. This extra role is a separate role to avoid static specification of the participant "owning" the state. When we consider that the scope of state variables for a contract will often extend beyond a single binding, the idea of using a separate role for maintenance of state becomes even more compelling since the component implementing the role must be shared across bindinges. One can think of this role as a shared contract state repository. This transformation of BCL state is depicted in UML in figure 6. The behavior of a Finesse role for maintaining a specific state variable is captured by the `FinRoleBehavior` class in this figure.

In our example, the `FinRoleBehavior` for the outstanding debt is captured in the following code fragment. We assume the existence of a remote procedure call (RPC) binding fragment[3].

```
Import RPC;
Roles {
  ...
  OutstandingDebt {
      loop {
        debit{RPC.Server((x:Real)())} OR
        credit{RPC.Server((x:Real)())} OR
        balance{RPC.Server(()(y:Real))}
      }
  }
}
```

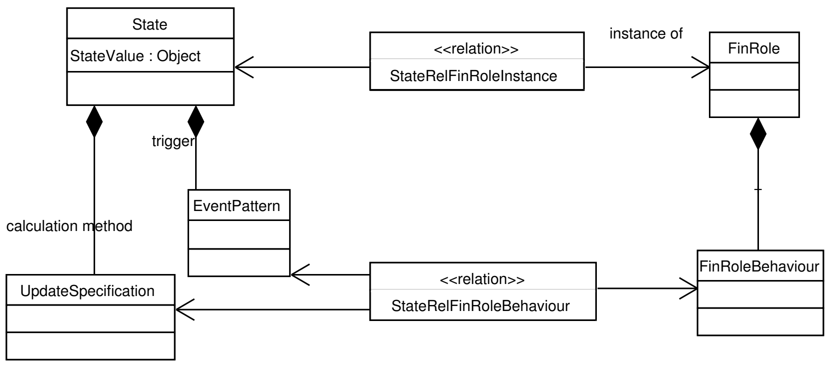

Fig. 6. Transforming state

```
Interactions {
  ...
  Freighter.goodsDelivered
    -> OutstandingDebt.debit.receive
      {x=prev.amount}
  AND Purchaser.payment
    -> OutstandingDebt.credit.receive
         {x=prev.amount}
}
```

In this fragment, a new `OutstandingDebt` role has three RPC operations for `debit`, `credit` and `balance`. The `debit` behavior is fired after 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` behavior is fired after the purchaser executes the `payment` event. The `balance` behavior will be used by any policy requiring the current balance. The `loop` construct is necessary to allow these operations to be used multiple times, remembering that iteration must be explicit in Finesse roles. It should be noted that the `->` operator in Finesse implies that the causal relationship *must* be satisfied and hence implies reliable messaging unless alternative failure behavior is explicitly specified. If other state is required in a BCL contract, it can be added in a similar manner.

3.8. *Values*

BCL has a *`Value`* concept that defines fixed data values initialized at contract activation time. This concept is missing from Finesse and for the purposes of the mapping, we will use the same approach as defined for BCL *`State`*, that is, define an interface for any values. This interface will not use RPC since the values are static and can be published just once, allowing any subsequent event to use the identified values.

The semantic model of Finesse is such that an explicit static value concept could be added with relative ease and would certainly be useful. In a mapping between BCl and Finesse, the values of such data items would be initialized at binding instantiation from the context of all relevant communities. Such a concept will be considered in future work on Finesse.

3.9. *Policies*

A `Policy` is used to specify business-level constraints in a BCL `Community`. These constraints are mapped to an existing Finesse binding definition by adding guards reflecting the policy constraints to existing behavior, and adding enforcement behavior as the alternate path if a guard indicates a violation. As discussed previously, the enforcement behavior in our mapping example will be to terminate the binding to reflect the decision to implement in-band enforcement. We will assume that all events identified in BCL event patterns are matched by equivalent events in Finesse: this is reasonable in our example when we consider that both the BCL and Finesse are derived from the same process definition.

The relationship between BCL policy and Finesse behavior is depicted in figure 7. This figure also depicts the relationship between BCL event patterns and Finesse event relationships which will be discussed further in sections 3.10 and 3.11. The basis of the mapping is that the event pattern associated with a policy can be transformed into an equivalent Finesse event relationship specification, with the policy condition transformed into a guard. The UML does not distinguish between role and interaction behaviour, with this distinction captured previously in figure 4.

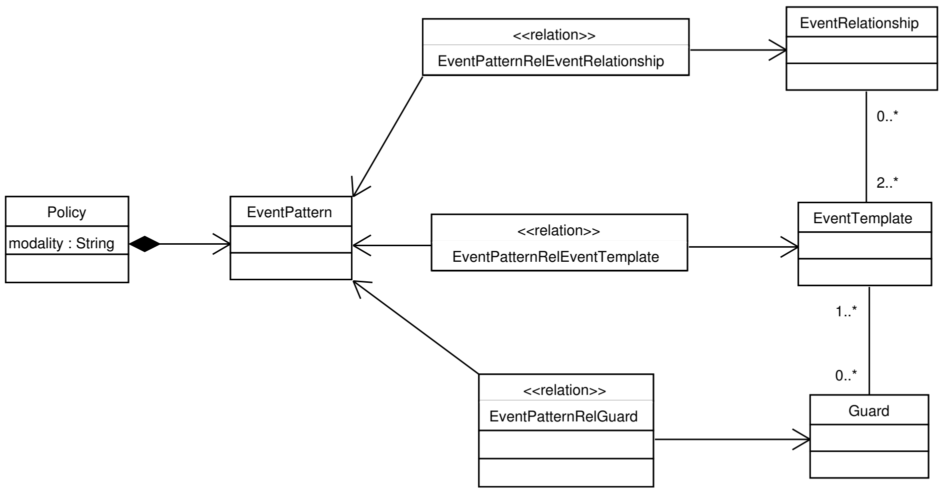

Fig. 7. Transforming policies

A key issue in the mapping of BCL policies to Finesse behavior is that the policy condition can reference attributes of an event being governed by the policy. This means that policy checking occurs *after* the event has occurred. Consider the credit limit constraint in the BCL purchasing contract of section 2.7: it is not possible to determine the purchase order value until after the purchase order has been placed, so the constraint can only be enforced after the event becomes visible. Thus in order to implement the required enforcement behavior, the guards reflecting policy conditions involving event attributes must be applied to all behavior that can follow an event being governed rather than the governed event itself.

There are three modes of policy expression: obligation, permission and prohibition. The mapping to constraints on the Finesse binding for each of these is discussed in the following subsections.

3.9.1. *Obligation*

A BCL `Policy` can have an `Obligation` modality, indicating that the governed behavior defined in the policy *must* occur. In order to conclusively identify a violation of the policy, there must be observable behavior indicating non-fulfillment. This can be achieved through one of the following:

(1) Time limits on the obligation, for example the obligation to perform *X* must be satisfied by time *T* as done in policy 5 of section 2.7.
(2) Attributes of events in the behavior that identify a violation, for example *X.value > $100*.
(3) The choice of an alternate and mutually exclusive path in the binding, for example, *Z* has occurred, indicating that *X* can no longer occur.

The approach to defining obligations in the Finesse binding depends on which approach is used to identify non-fulfillment. In case 1 above, the behavior *X* is guarded with an expression only allowing it to occur before time *T* with the alternative behavior signifying a violation, which can only occur at or after time *T*. This is illustrated in the code fragment below. In case 2 above, all events that can immediately follow *X* are guarded with an expression to check that the event properties satisfy the obligation, with the alternative behavior being termination. In case 3 above, the occurrence of *Z* can only be followed by termination.

We note that identifying a violation for case 3 above is particularly difficult in the general case because *Z* can be an arbitrarily complex piece of behavior and can also take an arbitrary period of time. We believe that this is not a useful way to determine fulfillment of obligations and suggest that any obligation without a deterministic time or event attribute constraint is potentially unenforcible.

There are several examples of obligations with time limits in the BCL purchasing contract, with the obligation of the supplier to have goods ready within five days illustrated in the following Finesse fragment:

```
Roles {
  ...
  Supplier {
    receiveOrder ->
        [timeless(prev, 5*3600*24)]
          orderFilled XOR
        [not timeless(prev, 5*3600*24)]
          Supplier.Termination.initiate }
    AND ...
  }
}
```

Note in the above example 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. Similar generation of Finesse constraints to reflect obligations can be defined for the other obligations of the contract. The complete, policy-augmented Finesse specification is defined in section 3.12.

In this mapping we have used the knowledge that both `receiveOrder` and `orderFilled` events are defined in the `Supplier` role. 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 policies is an instance of the general problem of making consistent observations in distributed systems[16,32]. This issue is further discussed in section 4.2.1. In our example we have taken the simplest approach which is to assume that the choreography engine at the source of the violating behavior can be trusted to report the violation.

3.9.2. *Permission*

A BCL ***Policy*** can have a ***Permission*** modality, indicating that the behavior defined in the policy is allowed to occur. From a semantic perspective, this is equivalent to a logical `OR` of the permitted behavior with the existing Finesse binding behavior. If we assert that the binding defines all possible behavior then permissions in BCL do not add behavior in the general case and the mapping need only ensure compatibility between BCL and Finesse specifications, that is, check that the permitted behavior is reflected in the binding.

In our sample contract we have one permission: "the supplier is permitted to provide an invoice immediately after goods delivery". The binding defined in 2.5 implies this behavior, since it requires an invoice after goods delivery, thus the policy is implicitly satisfied.

3.9.3. *Prohibition*

A BCL `Policy` can have a `Prohibition` modality, indicating that the behavior defined in the policy must not occur. There are two cases that must be dealt with in mapping BCL prohibitions to Finesse behavioral constraints:

(1) The BCL prohibition condition can apply to attributes of the governed events, for example, a purchase order shall not be processed if the value of that order plus the current outstanding debt is greater than an agreed maximum.
(2) The BCL prohibition condition can apply to event-level behavior, for example, if the supplier receives a credit rating below level `A` for the purchaser, no further orders are possible.

In the first case above we need to guard all behavior that follows the event to ensure that the prohibition is respected. In the purchasing process, for example, we need to access the state variable capturing the outstanding debt before evaluating the condition on all events potentially following the purchase order. A sample Finesse fragment for the condition above relating to our sample binding is thus:

```
Interactions {
  ...
  Purchaser.purchaseOrder ->
  RPC(Supplier.balance, OutstandingDebt.balance) ->
    {[valueOf(Purchaser.purchaseOrder.o) + Supplier.balance.receive.x >
       Values.creditLimit.limit]
         Supplier.creditExceeded.terminate{reason="over credit limit"}
     XOR
     [valueOf(Purchaser.purchaseOrder.o) + Supplier.balance.receive.x <=
       Values.creditLimit.limit]
         Supplier.receiveOrder{o = Purchaser.purchaseOrder.o}}
  ...
```

In the second case identified above, we can ensure that the prohibition is respected either by ensuring that the prohibited behavior is not permitted by the binding, or by guarding the prohibited behavior to reflect the prohibition. For the example prohibition defined above, we would need to add the following to the Finesse binding:

```
Roles {
  ...
  Supplier {
    ...
    receiveRating(r:CreditRating) ->
      { -- allow order if credit rating >= A
        [receiveRating.r >= A] receiveOrder()
        XOR
        -- terminate if credit rating < A
        [receiverating.r < A] ratingTooLow{Termination.Initiator}
      }
    XOR
      -- allow order if credit rating has not been received
      [not occur(receiveRating)] receiveOrder()
  }
}
```

The logical AND of this behavior with the existing binding behavior effectively applies the necessary constraint, noting that in this code fragment we have introduced another possible termination.

3.10. *Event Patterns*

BCL event patterns are conceptually equivalent to Finesse event relationships, allowing the specification of required causal, parameter and timing relationships between events. Assuming BCL event abstractions are refined to a level compatible with Finesse event templates, the mapping between event patterns and event relationships becomes a relatively straightforward matter. The primary activity is in identifying and matching the syntactic elements defining such relationships in each language. We demonstrate the mapping for the constructs in our example binding through code fragments and the complete augmented binding definition of section 3.12.

There are some issues that introduce complexity and require further discussion:

- Finesse distinguishes between role and interaction behavior because locality is important in defining the execution of a binding. BCL does not make this distinction. To resolve this issue, any relationship between event templates from distinct roles is placed in the interactions section of a binding program, while relationships for event templates in the same role are placed in the roles section. The outcome is that the augmented Finesse binding can split an event pattern across several parts of the program, thus losing the structuring in the original BCL definition. This is unavoidable; and
- The semantics of BCL event patterns is such that these patterns match any occurrence of the behavior embodied in the pattern. In other words, iterative behavior can be implicit in the event pattern. In contrast, Finesse requires explicit iteration in roles with implied iteration in interactions. The consequence is that BCL event patterns must be applied to all instances of iterative behavior in a Finesse program. This is generally not a problem, but must be considered when performing the mapping;

In the general case, there are also some key differences in the abstraction of events used in BCL and Finesse. Events in BCL can have a duration and can also be abstractions over a set of events in a lower-level context (event roles), meaning that events are potentially non-atomic and can have more than one location. In contrast, Finesse events are instantaneous, atomic and have a single location. If event abstraction mechanisms are used in policy definitions, then the abstract BCL events must be refined to a level where they can be mapped to a set of Finesse event templates.

In our purchasing example, the BCL event patterns are defined in terms of the event templates in an existing Finesse binding so we do not have any abstraction in the BCL. If we assume in the general case that BCL contracts are specified after the binding definition has been developed and in terms of the existing Finesse events, then we do not need to refine any event abstractions. Note that some BCL constraints are specified separately from the binding definition, for example internal policies or statuatory obligations. Such policies might need to be refined before transformation.

3.11. *Event creation rule*

BCL event creation rules are a special case of event abstraction that can be dealt with directly. Such rules define a synthesized event in terms of an event pattern over lower level events. It is not necessary to directly represent the synthesized event in Finesse but in some cases the explicit inclusion of a distinct event template for the synthesized event can improve readability. Consider the BCL for a *PurchaseSupplies* event creation rule that matches the complete purchaser behavior in our purchasing binding:

```
ECR PurchaseSupplies
  GenerateOn
      PlaceOrder->GoodsDelivered->
        InvoiceRecived->Payment
```

This event can be modelled by adding a `purchaseSupplies` event after the purchasing behavior as illustrated in the following code fragment:

```
Roles {
  ...
  Purchaser {
    ... -> payment
    -> purchaseSupplies
  }
}
```

Subsequent behavior can then refer to this event template rather than the more complex behavior that preceded it. This is consistent with the current approach to event creation rules implemented in BCA, which creates a new event once the nominated event pattern is matched.

3.12. *Augmented Binding Definition*

The following code is the complete result of applying the BCL to Finesse mapping to our purchasing example, expressed as a single Finesse binding program. There are a few details to highlight in the constraints added to the binding definition:

- The termination behavior defined in section 3.3 has been added to each role to capture the points at which violations can be detected. As previously noted, termination behavior has both an initiator and a set of receivers;
- The responsibility for initiating termination behavior has been given to the role that would be most interested in detecting the violation. The issues associated with the monitoring location identified in section 3.9.1 are very relevant here, since notifications of the events constituting a violation must be transmitted to that location before detection is possible;
- Finesse naming is used to distinguish each instance of termination behavior associated with violations. The name indicates the type of violation, but could equally have been the name of the policy that gave rise to the violation;
- As discussed in 3.7 and 3.8, roles are used to capture contract state and start-up values. Notice that RPC is used for state whereas implicit messaging is used for values. In effect, the `Values` role publishes all values making them accessible to any subsequent events in the binding; and
- Keep in mind that the `Interactions` section identifies causal and parameter dependencies between events occuring at distinct roles. Messaging is thus declarative, not imperative.

Comments in the program describe the enforcement and monitoring behavior added to the binding. Comments are introduced by lines beginning with "`--`". The Finesse program augmented with BCL policy constraints is thus:

```
Binding PurchasingProcess {
 Import RPC;
 Import Termination;
 Roles {
   Purchaser {
     purchaseOrder!(o:EdifactPO) ->
       {-- termination can occur after placing the order if the credit
        -- limit is exceeded
        creditExceeded{Termination.Receivers}
        XOR
        {-- Purchaser can initiate termination if goods are not ready
         -- in time or goods are not delivered in time
          goodsNotReady{Termination.Initiator}
          XOR goodsNotDelivered{Termination.Initiator}
          XOR goodsDelivered?()
            -> invoiceReceived()
            -> {payment(amount:Real)
                -- Termination can occur if payment is too late
                XOR latePayment{Termination.Receivers}}
        }
   }
   Freighter {
     -- termination can occur before goods are ready, e.g. credit
     -- limit exceeded or supplier does not have goods ready in 5 days
```

```
  creditExceeded{Termination.Receivers}
  XOR goodsReady?() ->
    {-- Termination can occur if the freighter does not deliver the
     -- goods within 5 days of the goods being ready.
     goodsNotReady{Termination.Receivers}
     XOR goodsPickup!() ->
      {goodsNotDelivered{Termination.Receivers} XOR deliverGoods!()}
    }
}
Supplier {
  -- Supplier can terminate before order received if credit limit
  -- exceeded. Current credit balance must first be retrieved.
  balance(RPC.Client()) ->
    {creditExceeded{Termination.Initiator}
     XOR receiveOrder?(o:EdifactPO) ->
       -- Purchaser can terminate if order is not filled in time
       {goodsNotReady{Termination.Receivers}
         XOR
         {fillOrder!() ->
           orderFilled!(value:Real) ->
             -- Purchaser can terminate if order not delivered in time
             {goodsNotDelivered{Termination.Receivers}
               XOR deliveryConfirmed?() ->
                 -- Supplier can optionally terminate after invoicing
                 -- if payment is late.  See interactions for details.
                 {invoicePurchaser!(i:EdifactInvoice) ->
                   {paymentReceived!(i:EdifactInvoice)}
                   XOR
                   latePayment{Termination.Initiator}
                 }
             }
         }
       }
    }
}
OutstandingDebt {
  loop {
    debit{RPC.Server((x:Real)())} OR
    credit{RPC.Server((x:Real)())} OR
    balance{RPC.Server(()(y:Real))}
  }
  -- loop semantics allow us to fall through to termination
  -- after zero or more iterations.  Role does not need explicit
  -- termination behavior because it is entirely reactive.
}
Values {
  creditLimit!(limit:Real) AND
  startDate!(start:Date) AND
  endDate!(end:Date)
}
```

```
}

Interactions {
  -- Delivery of goods increases outstanding debt
  Freighter.goodsDelivered ->
    OutstandingDebt.debit.receive {x=valueOf(Purchaser.purchaseOrder.o)}

  -- Payment by purchaser decreases outstanding debt
  AND Purchaser.payment -> OutstandingDebt.credit.receive {x=prev.amount}}

  -- Sending an order that blows the credit limit results in termination
  -- initiated by the Supplier. We first need to make sure that the credit
  -- limit and the current balance are accessible for the guard.
  AND Values.creditLimit ->
    Purchaser.purchaseOrder -> RPC(Supplier.balance, OutstandingDebt.balance)
      {[valueOf(Purchaser.purchaseOrder.o + Supplier.balance.receive.x >
         Values.creditLimit.limit]
           Supplier.creditExceeded.terminate{reason="over credit limit"}
       XOR
       [valueOf(Purchaser.purchaseOrder.o) + Supplier.balance.receive.x <=
         Values.creditLimit.limit]
           Supplier.receiveOrder{o = Purchaser.purchaseOrder.o}}

  AND Supplier.orderFilled
    -> Freighter.goodsReady{}

  AND Freighter.deliverGoods -> Purchaser.goodsDelivered

  -- Permission to invoice the purchaser is implied by the following
  -- and statement corresponding dependency between deliveryConfirmed and
  -- invoicePurchaser in the Supplier role.
  AND Freighter.deliverGoods
    -> Supplier.deliveryConfirmed{}

  AND Supplier.invoicePurchaser -> Purchaser.invoiceReceived

  -- Order must be filled within 5 days or the Purchaser will initiate
  -- termination
  AND Supplier.receiveOrder ->
    {[timeless(prev,5*3600*24)] Supplier.orderFilled
      XOR [not timeless(prev,5*3600*24)]
       Purchaser.goodsNotReady.terminate{reason="goods not ready"}}

  -- Order must be delivered within 5 days of notifying Freighter that
  -- goods are ready or the Purchaser will initiate termination
  AND Freighter.goodsReady ->
    {[timeless(prev,5*3600*24)] Freighter.goodsDelivered
     XOR [not timeless(prev,5*3600*24)]
       Purchaser.goodsNotDelivered.terminate{reason="goods not delivered"}}
    -- Payment must be made within 30 days of the invoice or supplier will
    -- initiate terminate termination
    AND Supplier.invoicePurchaser ->
      {[timeless(prev,30*3600*24)] Purchaser.payment
        XOR [not timeless(prev,30*3600*24)]
         Supplier.latePayment.terminate{reason="late payment"}

    AND Purchaser.payment -> Supplier.paymentReceived

    -- We need to bind termination initiation behavior with the termination
    -- behavior of other parties to that termination
    AND Termination(Supplier.creditExceeded,
                    (Purchaser.creditExceeded,Freighter.creditExceeded))
    AND Termination(Purchaser.goodsNotReady,
                    (Supplier.goodsNotReady,Freighter.goodsNotReady))
    AND Termination(Purchaser.goodsNotDelivered,
                    (Supplier.goodsNotDelivered,Freighter.goodsNotDelivered))
    AND Termination(Supplier.latePayment,Purchaser.latePayment)
  }
}
```

4. Discussion

This section discusses a number of technical and business issues related the transformation of business contracts to choreography expressions.

4.1. *Business issues in monitoring and enforcement*

In section 3 we have discussed transformation from the BCL into Finesse choreography for one specific approach to monitoring and enforcement, namely the prevention of non-compliant behaviour. This could be applied, for example, to mission critical systems where the risk of contract violations and associated costs are high. A higher level of risk requires more complete and perhaps more timely monitoring which can be achieved through in-band approaches as discussed in section 3.2. This approach to deployment of monitoring and enforcement, however, increases the cost.

In general, the transformation of contract terms to process constraints must consider business-level issues in choosing the monitoring and enforcement approach. The choice will be dictated by various economic, business and technology factors that affect risks associated with contract violations.

The first factor is the influence of *trust* in others, that is, the level of confidence that another party will perform as expected. This level of confidence will be based on on the experience of a party, typically arising from previous business interactions with the party or based on reputation. Note that in all cases, one needs also to consider the influence of uncertainty associated with the environment of the parties. The level of trust in other parties and related uncertainty directly affects the level of risk in a contract.

Issues of trust also arise in the choice of infrastructure technologies to support interactions between parties and the monitoring and control of those interactions. Monitoring in particular must rely on correct reporting of behaviour and requires trust that the infrastructure can resist tampering. For traditional middleware protocols, a protocol-level interceptor might be used to report behaviour to a third-party monitor, as is done in BCA. It is clear that a trusted third-party monitor can be chosen, but there must also be trust in the interceptor, which is typically located at the premises of a party and hence must be quite resistant to tampering.

In addition, reporting behaviour to the third-party monitor requires a reliable messaging infrastructure to ensure that behaviour is reported to the third-party monitor in a reliably and timely fashion. An infrastructure based on the Finesse engine is capable of monitoring behaviour and enforcing contract constraints without a third-party monitor, thus reducing the number of trusted components. This monitoring and enforcement behaviour occurs at the location of participants so we must again place trust in the ability of the Finesse engine to resist tampering.

A further business issue is the *complexity* of each business transaction: a more complex transaction is more likely to have violations and hence increased risk. This is most probably the result of bounded rationality problems, that is, that parties do not fully understand the transaction and make mistakes, rather than intentional failures caused by (mis)performance of the others.

Violation *cost* is a significant factor in the choice of monitoring and enforcement approaches. A transaction might have a high cost associated with violations, thus increasing the need for monitoring and enforcement, even where there is a high level of trust and low complexity. For example, in the finance, defence and health domains, there is a stronger case to implement strict enforcement mechanisms like the preventative enforcement approach presented in this paper. In other situations where violation costs are lower, it might be sufficient to rely on monitoring either carried out by parties themselves or by a trusted third party, and use policies or existing legal remedies to deal with violations. This “soft” enforcement approach can leave the decision of whether to enforce violation policies to the parties themselves. Finally, in cases where there are minimal costs associated with violations, it might be sufficient to rely on the monitoring of some critical variables only, rather than on a full set of monitoring conditions.

These factors of risk, cost, trust and complexity that need to be considered when selecting deployment options for monitoring and enforcement.

4.2. *Technical issues*

The mapping presented in the section 3 provides an indication of the work required to compile business contract specifications into 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 poli-

cies and monitoring requirements.

(2) The choreography environment must support the generation of events to signal potential contract violations if external enforcement or monitoring is required.
(3) The multi-party nature of event patterns makes it necessary to collect notifications of events from many sources to effectively monitor and enforce policies. This often implies delivery of event notifications to more than one destination, or in other words, multicast. The Finesse model handles this elegantly, but the need for multicast could add significant complexity for choreography models based on explicit point-to-point messaging.
(4) 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.
(5) The ability to add constraints to an existing process definition using logical operators is significant to the transformation. Declarative approaches are typically more amenable to this flexible conjunction of process and monitoring behaviors.

Alternate contract and choreography languages need to address these issues to be useful in mapping contract terms to process constraints. The following subsections discuss a number of these semantic issues in more detail.

4.2.1. *Time and locality*

There are many possible ambiguities in a contract resulting from policies that do not identify time and location reference points. Consider the the statement "the supplier is obliged to have goods 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 the period begins when the supplier receives the order. Without this explicit specification, there is considerable potential for ambiguity and dispute;
- If the policy is monitored by the purchaser or a third-party monitor, then the monitor must have confirmation that 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? How is a trusted timestamp generated?
- If the policy is monitored by the supplier, then the supplier has a conflict of interest because reporting a violation potentially leads to a penalty. Can we trust the choreography engine to correctly report the violation from the supplier location?

Notice that in all three items above, the location of actions is critical to the contract. If the contract specifies that the time period is measured from the time the order is sent by the purchaser, the supplier carries the risk that the order might be lost in communication. If monitoring were to be carried out at the supplier, then we remove the problem of lost orders but add the need for increased trust in the supplier to admit violations (which we have assumed in our mapping). If we use transactional messaging, the problem of lost orders 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 behavior at multiple locations so using a generic rule to detect violations at the source cannot be universally applied.

We highlight these issues by noting that there is, in fact, a problem in the augmented specification of section 3.12. Consider the enforcement behavior associated with late payments. The condition requires that either the purchaser pays within 5 days of the invoice or the supplier will initiate the late payment enforcement action. Because the purchaser and supplier are not co-located, it is possible the the purchaser will pay before the deadline is reached but notification of that payment will not reach the supplier before the deadline and the supplier initiates the enforcement action. Due to the use of true concurrency and explicit locality in the operational semantics of Finesse, it is possible to detect this problem through automated analysis of the program.

We can resolve many 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.

4.2.2. *Implementing global state*

BCL includes the concept of state shared by the participants of a community. While such global state is relatively easy to maintain consistently in an environment with centralized control, maintaining this state in a distributed context like that of our purchasing example is considerably more difficult. Global state requires synchronization semantics and thus can introduce the possibility of unbounded blocking due to deadlocks and network failures. Synchronization becomes even more problematic in a system involving participants connected by the Internet because protocols like two-phase commit do not scale to widely distributed networks and suffer from starvation in unreliable networks, that is, updates can often be unsuccessful because of failures or conflicts. Consider the choices first identified in section 3.7:

(1) Maintain state in an explicit role.

This option is reliable but it is not always desirable, for example, if a remote location wants to evaluate a policy referencing the state value it must always access the state across the network. Network failure means the state is unavailable.

(2) Evaluate functional expressions over events in a parameter relationship.

For example, if role A is responsible for the state and has seen two update events X and Y, then the current state is defined by f((X,Y)) at role A. This is similar to option 1 except that we rely on the non-persistent storage of event history in the Finesse engine. This suffers from so-called *dirty reads* because the local state might not include a recent update made by another participant at the time of evaluation.

(3) Through explicitly defined state synchronization behavior.

This option suffers from the possibility of poor performance due to synchronization overheads or unbounded blocking in the event of network failures.

While one of the above solutions might be ideal for some circumstances, all generic solutions are susceptible to dirty reads, failures or starvation when consistent state is required in more than one location. Given this discussion, any `Community` definition that requires consistent state shared across participants must be handled with some care. A scalable and robust solution is to explicitly permit the use of local and potentially inconsistent state (option 2 above), but this cannot always be accomodated by the parties to a contract.

4.3. *Complexity and usability*

The augmented binding definition is considerably more complex and verbose than the original binding definition. Given that this is a simplistic process and contract definition, the complexity is cause for concern if the augmented binding definition must be human readable. In part the extra complexity occurs because considerable additional behavior has been added to the definition. The fact that the Finesse behavioral model is capable of accurately capturing the contract constraints is significant because it validates the belief that contract constraints can be translated into process constraints. The basic process steps are unchanged, however, and it would be worthwhile to consider syntactic changes in the Finesse language allowing exception behavior to be separated and abstracted from the basic binding definition.

5. Related Work

The following subsections discuss alternate research, standards and commercial technologies for the definition and implementation of contracts and business processes. The goal is to define the relationship of BCL and Finesse with other technologies and indicate how the technique described in this paper might be relevant to these technologies.

5.1. *Alternate Contract Definition Technologies*

As far as the authors are aware, BCL is currently the most comprehensive contract language developed to express contract semantics for cross-organizational enterprise models. It has a number of distinguishing features, including:

- It supports the expression of deontic policy constraints in contracts using powerful and versatile event-based behavior specification, while at the same time positions policies as part of its context, namely the community;
- It has the ability to express relationships between communities and delegation of policies within a community allowing hierarchical and peer-to-peer relationships between policies;
- It was designed with extensibility in mind and can allow expression of many types of monitoring condition, whether they come from an internal business process or as part of cross-organizational collaborative interactions;
- It uses model-based paradigms and is suitable for use as part of a corresponding tool chain for the full contract management life-cycle[13]; and
- It is compliant with ODP standards for enterprise language[37].

In certain aspects the Contract Expression Language (CEL)[7,42], recently proposed and currently under development within the Content Reference Forum[9], is similar to BCL. Like BCL, CEL is inspired by deontic logic style of policy expressions in contracts but is motivated by the need to represents contracts used in content distribution. In fact it is based on the ISO/IEC MPEG standard Rights Expression Language[31,9]. It is an XML-based language designed to capture and communicate contractual information, and facilitate contract execution and enforcement by machines with respect to granted permissions, mandated obligations and stipulated prohibitions. Unlike BCL, the current version of CEL does not define an organizational and business process framework so that contract conditions can be represented as governance of cross-organizational interactions. It also does not have a clearly defined behavioral model like BCL event patterns for the expression of deontic constraints.

Farrell et al[15] are working on a contract language that also has its basis in deontic formalism. It makes use of an event calculus[20] to track normative state of contract in response to contract events. The normative state of the contract at a specific time is the aggregation of instances of normative relations (e.g. obligations, permissions and power) plus the current values of contract variables. This language is similar to BCL in its use of event relationships to describe behavioral constraints associated with policies. Unlike BCL, the focus of the language is on tracking and visualizing contract state: it does not consider the overall enterprise model in which contracts typically serve as a governance mechanism for cross-organizational transactions.

There are some similarities between BCL and the Ponder language[10] with respect to the specification of policies. Ponder provides a common means of specifying security policies that map onto various access control implementation mechanisms for firewalls, operating systems, databases and Java applications. It includes authorization, filter, refrain and delegation policies for specifying access control, as well as obligation policies to identify and control management actions. In comparison to BCL, Ponder provides a subset of the BCL compositional operators for specifying behavior, reflecting the system management needs for which it was primarily developed.

In terms of relevant open standards, the results of BCL are feeding into the current OASIS e-contracts TC standardization[22]. In addition, some aspects of BCL are similar in style to current business process definitions like BPEL[45], WSLA[44], WS-CDL[18] and WS-Policy[46].

The distinguishing feature of BCL is that it was primarily developed with the aim of specifying abstractions of the business contract domain and it allows a business-oriented specification of policies or contract terms to be applied to existing process definitions. In fact, one of the key targets for BCL is the specification of policies for business-to-business interactions[38]. Early process technologies for workflow and most existing business process management products are restricted to definitions of a local process with centralized control. There is no ability to express constraints on interactions between autonomous business processes. The emerging choreography technologies and standards[18] address this deficiency. As with local process management technologies, BCL can be seen as complementary to these technologies.

In terms of commercial offerings, there are a number of dedicated contract management vendors that have emerged in recent years. Examples are DiCarta[11], UpsideContracts[40] and iMany[17]. A feature common to all these products is the intention to support full contract lifecycle management. This ranges from collaborative contract drafting and negotiation (mainly exchanging electronic documents), through storage of contracts and milestone-driven notifications, to control and analysis features. These enterprise contract management systems generally follow the database approach typical of most ERP systems and the contract semantics is implicitly encoded as part of various data and processes. This is perhaps because there is no overall model that expresses the semantics of contracts as a governance mechanism for cross-organizational collaboration: this is the problem that is being addressed by BCL, CEL and the like. It is also important to note that the commercial offerings are also inward-facing, focusing on the definition and management of contracts from the perspective of a single organization. In contrast, BCL provides a cross-organizational model.

5.2. *Alternate Choreography 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 behaviors 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[26] and ebXML[14]. In particular, the explicit inclusion of locality and causality, and the use of a truly concurrent operational semantics are key distinguishing features that make it more amenable to describing processes that span distributed, autonomous participants. With this operational semantics, the Finesse engine can be thought of as a distributed, asynchronous and programmable process engine.

In the context of business contracts and choreography, we expect that the Finesse engine or an equivalent choreography engine would be embedded in a the enterprise-level interface 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 organization. The current state of these alternate technologies is not sufficient, however, to address many of the issues raised in this transformation.

Web services choreography[18] perhaps comes closest to providing the expressiveness and capabilities required for the implementation of contract constraints. It has explicit roles, a level of abstraction over messaging, guarded behavior, the ability to define local variables, and state alignment protocols for managing contract state variables. That said, the current draft language has some design features that make it difficult to directly apply contract constraints, specifically:

- Interaction is performed over two-party channels and is imperative rather than implied, making passive monitoring (interception), multicast and roles with non-unary cardinality very difficult to model;
- It uses an imperative notion of *ordering structure* including sequence, parallelism and choice to capture dependencies between activities. While the expressiveness of this approach is similar to the declarative causal dependency approach of Finesse, it can be structurally difficult to add new behavior to an existing process definition. In other words, it is difficult to specify a logical AND of two overlapping behaviors. This is ameliorated by the inclusion of a separate structure for exceptions, since most behavior added in the transformation is to deal with exceptions (i.e. contract violations); and
- Causality is not an explicit concept in the language and must be inferred from ordering structures. The draft specification does not include a formal operational semantics, but if a model based on interleaved concurrency is chosen, causality can be inferred incorrectly in certain situations.

Nonetheless, the draft standard is very promising. It is hoped that subsequent revisions of the standard can take the issues mentioned above into account and make it a much better platform for the implementation of contract constraints.

Process modelling notations like BPEL[45], ebXML[12,14] and BPMN[6] are primarily oriented towards the definition of centrally controlled processes and are not currently capable of implementing many of the constraints expressed in BCL. ebXML and BPMN have notations to model distinct localities and messaging between those localities, but fall short when attempting to model and identify the complex interactions captured by BCL event patterns. The BPMN specification explicitly admits its deficiencies in describing interactions between autonomous parties and identifies this as an area for future work. That said, any of these notations could be used to model and implement the local component of behavior associated with each party to a contract. The technologies for implementation and management of local processes based on these notations are also quite mature so it is likely that implementations of monitoring systems based on choreography would interact directly with these process management technologies. It will be important to accurately describe the necessary interactions with these systems.

6. Conclusions and Future Work

In this paper we have used BCL and Finesse as vehicles to demonstrate the translation 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 and BCA[23,29,29,32], but we envisage that such contract specifications will be used in environments based on standard process execution technologies like BPEL[45] and WS-CDL[18] 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 standardization efforts like OASIS eContracts[22] and WS-CDL[18] respectively.

While there were many issues raised in developing a mapping from contract to process, the exercise has indicated that a translation procedure is feasible and potentially useful. There were no particular technology limitations in the transformation for this example and it is expected that the mapping presented will work in a relatively generic fashion for the features used.

There is considerable scope for future work arising from the findings of this paper, specifically:

- The usefulness of the translation needs to be explored further through more complex examples to demonstrate its feasibility in a realistic business environment. We envisage that future development in model-driven transformation[33] will also be considered;
- The addition of contract constraints to the example binding has shown a significant increase in the complexity of the binding definition. The development of improved syntactic representations of the binding to manage this complexity would be valuable, for example, the addition of exception handling constructs in the language and the ability to abstract over exception semantics;
- While many of the issues identified are quite general, the mapping presented is specific to the BCL and Finesse languages and their respective semantic models. The development of a mapping for alternate technologies, particularly those embodied in standards-based work, would be very useful and relevant;

- A key business issue in defining contracts and processes is the need for dynamic change, as discussed in section 3.1. While we have suggested ways to ensure the Finesse implementation of the BCL contract constraints is flexible, there is considerable scope for future work in the management and implementation of change in a process;
- There are features of BCL that have not yet been explored, for example, sliding window temporal constraints[23,29]. Further work is required to develop a complete and generic BCL to Finesse mapping; and
- The domain of contracts and policy specification is under active research. BCL is evolving as the understanding of the domain improves, for example, in the area of policy refinement and the description of priorities associated with policy. These changes in the domain and BCL will need to be reflected in any mapping to choreography expressions.

The example has presented a solution to the problem of implementing monitoring and enforcement of contract constraints. It is important to remember that the business-level aspects of the problem will often have a significant influence on the implementation, for example, the issues discussed in section 4.1. While it is possible to translate contract constraints into expressions in a choreography language, this might not always be a viable solution because the enterprise might require the use of external tools or third parties that cannot be included in the choreography for business or technological reasons. As noted in 4.1, for example, the need for discretionary enforcement typically requires an external solution.

We conclude by emphasising the key contribution of the paper: that cross-organizational business processes can be monitored and enforced according to business contract specifications through the transformation of a precise contract definition to direct constraints on process behavior.

7. 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).

The authors would also like to thank James Cole, Andy Bond, Michael Lawley and Guido Governatori for their assistance in preparing this paper.

References

1. D. H. Akehurst. Transformations based on relations, 2004. http://heim.ifi.uio.no/~janoa/wmdd2004/papers/akehurst.pdf.
2. Autonomic computing. http://www.research.ibm.com/autonomic/.
3. A. Berry. *Describing and Supporting Complex Interactions in Distributed Systems.* PhD thesis, University of Queensland, 2002.

4. A. Berry and S. Kaplan. Open, distributed coordination with finesse. In *ACM Symposium on Applied Computing*, Feb. 1998.
5. A. Berry and K. Raymond. The A1$\surd$ architecture model. In *Open Distributed Processing: Experiences with distributed environments*. IFIP, Chapman and Hall, Feb. 1995.
6. Business process modelling notation, 2004. http://www.bpmn.org/.
7. CEL: Contract expression language, 2003. http://www.crforum.org/articles/about/candidate.html.
8. P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models*. Springer, 1996.
9. Content Reference Forum. http://www.crforum.org/.
10. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder specification language. Workshop on Policies for Distributeed Systems and networks (Policy2001), jan 2001. HP Labs Bristol.
11. Dicarta. http://www.dicarta.com.
12. J. J. Dubray. Standards for a service oriented architecture, 2003. http://www.ebxmlforum.org/articles/ebFor_20031109.html.
13. K. Duddy, M. Lawley, and Z. Milosevic. Elemental and Pegamento: the final cut: Applying the MDA pattern. In *Proceedings of Enterprise Distributed Object Computing (EDOC2004)*, Monterey, California, USA, 2004. IEEE.
14. The OASIS ebXML standards. http://www.ebxml.org.
15. 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.
16. C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, Aug. 1991.
17. iMany. http://www.imany.com.
18. 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/.
19. 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.
20. R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.
21. R. Lee. A logic model for electronic contracting. *Decision Support Systems*, 4(1):27–44, 1988.
22. OASIS LegalXML eContracts. http://www.oasis-open.org/committees/legalxml-econtracts/charter.php.
23. P. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal. A unified behavioural model and a contract for extended enterprise. *Data & Knowledge Engineering*, 51:5–29, 2004.
24. Loosely coupled glossary: Choreography. http://looselycoupled.com/glossary/choreography.
25. D. Luckham. *The Power of Events*. Addison-Wesley, 2002.
26. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes: Part I and II. *Information and Computation*, 100(1), 1992.
27. Z. Milosevic. *Enterprise Aspects of Open Distributed Systems*. PhD thesis, University of Queensland, 1995.
28. 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.
29. 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.
30. Z. Milosevic, A. Josang, T. Dimitrakios, and M. A. Patton. Enforcement of electronic contracts. In *Proceedings of Enterprise Distributed Object Computing (EDOC2002)*, Lausanne, Switzerland, 2002. IEEE.
31. ISO/IEC 21000-5 Information Technology Multimedia Framework Part 5: Rights Expression Language, 2003. http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=36095.
32. S. Neal. *A Language for the Dynamic Verification of Design Patterns in Distributed Computing*. PhD thesis, University of Kent, 2001.
33. Request for proposal: MOF 2.0 Query / Views / Transformations RFP.
34. Oracle Contracts. http://oracle.com/appsnet/products/contracts/.
35. O. Perrin and C. Godart. An approach to implement contracts as trusted intermediaries. In *First IEEE International Workshop on Electronic Contracting*, July 2004.
36. ISO/IEC 10746-1 10756-2 10746-3 10746-4 Basic Reference Model for Open Distributed Processing.
37. ISO/IEC IS-15415 Open Distributed Processing-Enterprise Language, 2002.
38. K. Schulz and Z. Milosevic. Architecting cross-organisational B2B interactions. In *Proceedings of Enterprise Distributed Object Computing (EDOC2000)*, Makuhari, Japan, 2000. IEEE.
39. R. Tag, Z. Milosevic, S. Gibson, and S. Kulkarni. Supporting contract execution through recommended workflows. In *Proceedings of the 15th International Conference on Database and Expert Systems Application DEXA04*, Zaragoza, Spain, Sept. 2004.
40. UpsideContracts. http://www.upsidecontracts.com.
41. G. H. von Wright. Deontic logic. *Mind LX*, 237:1–15, 1951.
42. X. Wang, E. Chen, D. Radbel, H. Tsutomu, J. Clark, and G. Wiley. The contract expression language - CEL. IEEE Contract Languages and Architectures (CoALa) Workshop, Sept. 2004.
43. J. Widom and S. Ceri, editors. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufman, 1995. ISBN 1-55860304-2.
44. Web service level agreements (WSLA) project. http://www.research.ibm.com/wsla/.
45. Business process execution language for web services, May 2003. http://www.ibm.com/developerworks/library/ws-bpel/.
46. Web services policy framework (WS-Policy), 2004. http://www.ibm.com/developerworks/library/specification/ws-polfram/.