On Expressing and Monitoring Behaviour in Contracts[1]

Z. Milosevic[1], R.G. Dromey[2],
[1]Distributed Systems Technology Centre, Brisbane, Qld 4072, Australia
zoran@dstc.edu.au
[2]Software Quality Institute, Griffith University, Nathan, Brisbane, Qld., 4111, Australia
rgd@cit.gu.edu.au

Abstract

This paper addresses the problem of transforming natural language descriptions of contracts into a form that is suitable for automating various contract management functions. We investigate two complementary methods that can be used to achieve this. One method is suitable for the contract specification phase – to specify expected behaviour of contracting parties so that they can satisfy policies stated in a contract. This method also allows for checking aspects of contract consistency as well as flexible integration of internal organisational policies with the contract policies. Another method targets the contact run-time phase – for monitoring behaviour of parties to the contract and other aspects of contract performance. When combined, these two methods provide a basis to support an increasing level of automation of many mundane contract activities, while allowing humans to be involved in ultimate decision making.

1. Introduction

A contract is an agreement governing part of the collective behaviour of a set of objects [1]. It specifies obligations, permissions and prohibitions of the objects involved, all of which are regarded as constraints on the objects' behaviour in relation to other objects involved in contracts. However, behaviour of parties involved in a contract is also influenced by other factors, such as constraints arising from their own objectives, policies of other domains (e.g. internal policies of enterprises, as depicted in Fig.1) and some other *force majeure* factors. These can lead to actual behaviour that is not always compliant with contract specifications. To determine this compliance one needs to put in place mechanisms for monitoring their behaviour. This can be used to influence parties to take appropriate actions or to impose other corrective measures so that their mutually accepted behaviour as agreed in the contract is ensured.

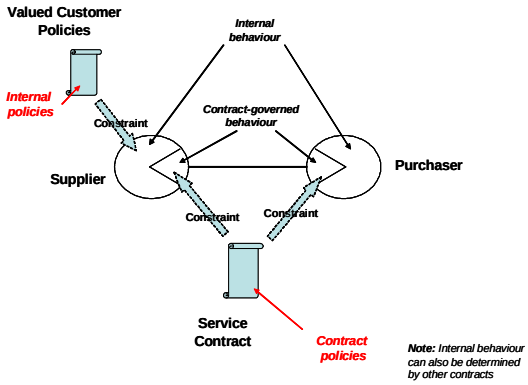

Figure 1 Internal Policies and Contract Policies: Service Contract between Purchaser and Supplier

There is currently a large gap between this system-theoretic model of contracts and the way contracts are traditionally described - using natural language. This paper investigates two solutions which can be used to facilitate mapping of natural language contract representation into models suitable for contract automation - as increasingly needed for cost efficient operations associated with electronic contracts.

One solution makes use of the recently proposed Genetic Software Engineering (GSE) method [2]. This method enables transformation of individual policies stated in natural language in contract clauses into a computer language representation. This representation is in a form of component-based behavioural trees – which if executed by parties to the contract would ensure contract-compliant behaviour. Further, GSE method allows

[1] 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).

composing behaviour trees corresponding to the individual clauses into an overall tree for the whole contract. Additional value of the method is to facilitate detections of inconsistencies in contract. Thus, GSE main value is at the *contract specification* phase – it can be used to derive guidelines to the parties for contract-compliant behaviour and as a valuable verification tool for ensuring consistent contract description.

The other solution addresses the *contract execution phase* and is motivated by the fact that enterprise behaviour execution can bring a certain degree of inconsistency in relation to its specification. In order to deal with such inconsistencies one can deploy various monitoring mechanisms. The solution presented in the paper makes use of a policy language that implements key ideas behind ODP policy specifications [1]. This language is an alternative way of describing behaviour. As opposed to the generic behaviour nature of the GSE concepts, the policy language gives prominence to obligations, permissions and prohibitions – to which many contract clauses can be reduced. In terms of a contract management architecture, this language is central to the interpreting of behaviour traces of parties to the contract and forms the basis for various monitoring aspects of contracts. The ultimate objective is to facilitate detection of possible non-compliance to the contract and take appropriate corrective actions.

Both of these approaches are illustrated with a simple example from B2B domain (given in Appendix). The example introduces a Service Contract between a Supplier and a Purchaser, specifying their mutual policies and it also shows Valued Customer policies of the Supplier specifying Supplier's internal policies, as shown in Fig. 1.

The rest of the paper is structured as follows. Section 2 describes key features of the GSE method and how it can be used to capture behaviour in contracts. Section 3 outlines the main differences between enterprise and computational specifications and presents our policy-based contract monitoring - as part of a Business Contracts Architecture, previously proposed [7]. Section 4 discusses relation between the behavioural expressions of GSE and of the policy language. Conclusions and future work directions are provided in section 5.

2. Genetic Software Engineering method

In order to provide general support for the implementation, and monitoring of contracts across enterprises it is necessary to have a way of capturing, first in a formal specification, and then in software, the policies, actions, events, decisions, obligations, behaviors and constraints expressed in original natural language representations of contracts.

The legal nature of contracts and an increasing need for automated management of contracting activities makes it a priority that there is a direct and clearly traceable relationship between what is expressed in the natural language representation and in the formal specification. Behavior trees [2] have been successfully used in software engineering to capture functional requirements behavior and are an attractive option for this purpose because they may be used to translate, on a sentence-by-sentence basis, the behavior expressed in a contract.

2.1 Requirements Translation

Translation to behaviour trees involves identifying the *components* (including actors and users), the *states* they realise (including attribute assignments), the *events* and *decisions/constraints* that they are associated with, the *data* components exchange, and the *causal*, *logical* and *temporal* dependencies associated with component interactions. The translation process can be quickly and easily mastered because of the use of a simple Component-State defining form and the small number of operations that can be associated with components, states and attributes. The tagged component-state notation captures the essential behaviour. For example, the third box in Fig.2 to clause 4.2 in the contract and it says "component 'SUPPLIER' has realised the state that 'the Goods are Available'. At the same time the component 'Goods' has realised the state 'Available'.

The principal conventions of the notation for component states are the graphical forms for: [State], ??Event??, ?Decision?, [Component[State]], [Attribute := expression | State]. Exactly what can be an event, a decision, a state, etc are built on the formal foundations of quantifier-free formulae (qff). To assist with traceability to original requirements the following conventions are followed. Tags (e.g. 4.1 and 4.2, see below) are used to refer to the original clause in the document that is being translated. Record/data definitions and other constraints or comments are signaled by a "/" and are included in round-cornered rectangles (see example translation below). System states, which are used to model high-level behavior, preconditions/postconditions and possibly other behavior that is not associated with particular components, are represented by rectangles with a double line === border.

In practice when translating contract requirements into behavior trees we often find that there is a lot of behavior that is either *missing* or is only *implied* by the requirement

(or clause). We mark implied behavior with a (+) in the

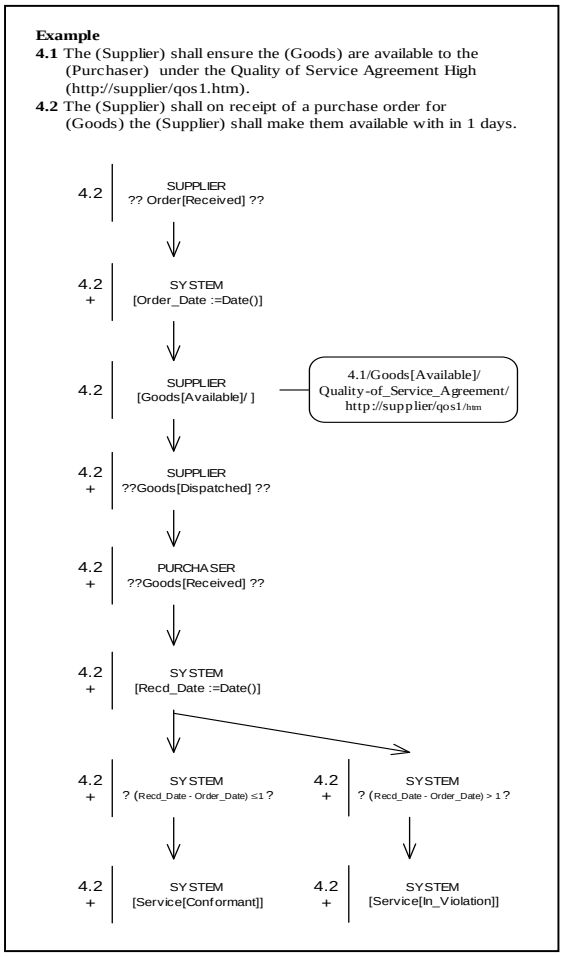

Figure 2 Behaviour Tree for Clause 4.2

tag (and/or the colour yellow if colour can be shown). Behavior that is missing is marked with a (-) in the tag (and/or the colour red). Explicit behavior in the original requirement that is translated and captured in the behavior tree has no marking (+/-) (and the colour green is used), Fig. 2. These conventions maximize traceability to original requirements. The Green-Yellow-Red traffic light metaphor is intended to indicate to readers of the specification the need for caution (yellow) and danger (red) and to draw attention, to deficiencies in the original requirements. This is particularly useful when discussing requirements and designs with users or clients.

Example Integration

An example of requirements translation into a behavior tree is shown in Fig. 2 for sections 4.1 and 4.2 of the Contract for Services (see Appendix)

In translating the clauses 4.1. and 4.2 the obvious components that exhibit behaviour are the SUPPLIER and the PURCHASER. Other components are the GOODS, the ORDER and the SERVICE. We have also added the SYSTEM as a component to absorb the behaviour needed to deal with late/on-time delivery of goods. In specifying the behaviour we have allowed for the possibility that the service will not be provided in the time specified. An example of an event is "order received", (an event is a behaviour that only *may* occur and if it does occur we cannot predict exactly when it will occur – it holds up the flow of control to its child nodes in the behaviour tree until it occurs), a decision is "are goods received within one day", and a state realisation is "Order[Received]". The latter is also an event because of its temporal dependence. Note also, "Received" is a state realised by the component "Order". Usually in doing requirements translations we do a literal translation first, followed by any necessary augmentations to make the behaviour usable in an automated system. Here, because of space limitations, we have not shown the literal translation step. For this behaviour tree the number of (+'s) indicate that there was a significant amount of implied behaviour in these two clauses. The behaviour in 4.1 and 4.2 could have been expressed more succinctly by avoiding specification of details about recording dates (which may be regarded as implementation level detail) and simply providing a choice of "?Late?" or "?NOT:Late?" decisions to determine whether or not there has been a violation of the contract's service provision. There is ambiguity in the original requirement about exactly what is meant by "make them (the Goods) available". We have interpreted this as "requiring that the goods are received by the purchaser".

The Behavior Tree Notation is a graphic notation for representing a wide range of behavior that is likely to be found in areas as diverse as advanced technological applications, legal documents, standards and procedures. Important advantages of behavior trees are their expressive power coupled with notational simplicity, their ability to accommodate complexity and detail, their ease of use, their composability, their ability to expose behavioral defects and their derivable properties [2]. They allow complex behavior to be expressed both in detail and at an abstract level within the one framework (see [2]). And, importantly, they allow behaviour of individual components to be easily partitioned and separated out. In genetic software engineering (GSE) individual functional requirements (or sentences) are first translated into composable *behavior trees*. Each requirements behavior tree (RBT) is then integrated one at a time to create a design behavior tree (DBT). This amounts to building a

system ***out of*** its requirements, rather than simply building a system that will satisfy its functional requirements. This maximizes traceability, which is vitally important when dealing with contracts. Because a design is composed out of its requirements, one at a time, this greatly simplifies the design process (compared with attempting to construct a system that merely *satisfies* a large number of requirements) and makes adding, modifying or deleting a requirement (that is, change or evolution) relatively straightforward.

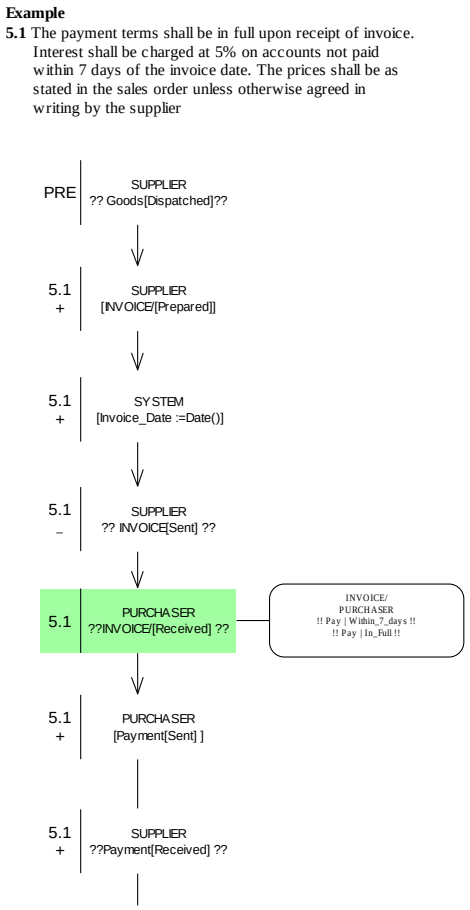

Figure 3: Behaviour Tree for part of Clause 5.1

The advantages and the process for using behavior trees that applies in genetic software engineering can be directly carried over to the contract specification and implementation problem domain because the behavior that needs to be captured for contract automation is essentially the same as the behavior expressed in functional requirements. Using the behavior-tree notation we can *translate* each individual contract requirement, use case, or constraint, expressed informally in natural language, into its corresponding formal graphic behavior-tree representation. Behavior trees capture/express behaviors in terms of state realizations, state transitions and component interactions. There are three important advantages that flow from such translations: (1) there tends to be little variability among literal translations made by different people of the same requirement (2) The component-state form of behavior-trees is compatible with object-oriented and component-based designs (3) The translation process is very effective for revealing defects and/or missing behavior in the original natural language requirements.

Due to the limited space available we do not provide full details of the GSE method – rather we concentrate on illustrating the application of the method to contracts. Elsewhere (see http://www.sqi.gu.edu.au/gse/papers in an SQI Technical Report titled *Genetic Software Engineering)* we have used Dijkstra's weakest precondition conventions to formally define the semantics of the textual equivalents of the core elements of the Behavior Tree Notation. The notation is particularly well suited to having its semantics formally defined using Modal Logic.

2.2 Requirements Integration

The design process proceeds as follows. First each individual requirement is translated to its corresponding requirements behavior tree (RBT). We can then systematically and incrementally construct a design behaviour tree (DBT) that will satisfy all its requirements *by integrating the requirements' behavior trees* (RBT) one at a time. Integrating two behavior trees turns out to be a relatively simple process. It most often involves locating where the component/state root node of one behavior tree occurs in the other tree and grafting the two trees together at that point. This process generalises when we need to integrate N behaviour trees. We only ever attempt to integrate two behaviour trees at one time.

Example Integration

To illustrate the process of requirements integration we will take another of the requirements in the Contract for Services (example in the Appendix) and integrate it with the translated requirement for section 4.1 and 4.2 given above. The requirement we choose to integrate is that for section 5.1. Fig. 3 depicts a partial requirements translation for section 5.1. The root of 5.1's behavior tree, after direct translation, is the component-state SUPPLIER[INVOICE[Prepared]]. To try to integrate it we can look for this node in the 4.1/4.2 Behavior Tree – it is not there. When this happens, as is often the case, it usually means there is a missing precondition for the

behavior tree. To remedy this problem and integrate this invoicing/payment requirement we need to ask the question, "what precondition is necessary in order to send the purchaser an invoice?" Domain knowledge and/or good business practice would suggest that you do not send the purchaser an invoice until the goods have been dispatched. We therefore add this precondition as the new root of the 5.1 behavior tree. (see Fig. 3). We now have the information needed to go ahead and integrate the two behavior trees using the *root node matching behavior tree integration strategy*.

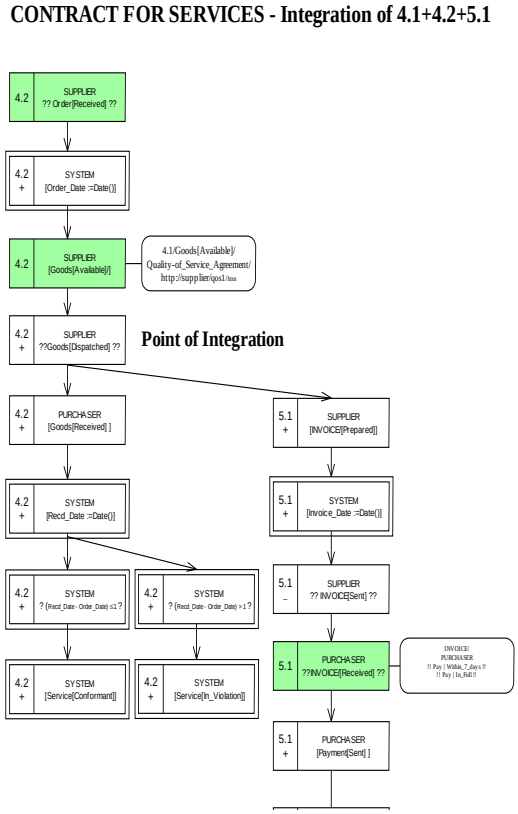

Figure 4: Integration of behaviour trees

Figure 4 shows the result of integrating the behavior trees for two requirements 4.1/4.2 and 5.1 by grafting 5.1 onto 4.1/4.2 at the common component node SUPPLIER??Goods[Dispatched]??. The resulting integrated behavior-tree (or DBT) *satisfies both* 4.1/4.2 and 5.1. The contributions of the individual requirements 4.1, 4.2 and 5.1 are traced in the integrated or design behavior-tree by the their respective numbers used to tag each of the component/state nodes in the tree. This level of direct traceability is important when we are concerned with embedding contract requirements in a design.

Using this behavior-tree grafting process, a complete design is constructed incrementally by integrating one requirement at a time into the evolving DBT. This is the ideal for design construction that is realizable when all requirements are consistent, and composable. When it is not possible to integrate an RBT into the DBT it points to an integration problem with the specified requirements that needs to be resolved. Being able to construct a design incrementally, significantly reduces the complexity of this critical phase of the design process. And importantly, in the contract-automation domain, it provides direct traceability to the original clauses in the contract.

Requirements translation and then *requirements integration* works as a design strategy because individual functional requirements represent "fragments of behavior" whereas a design represents "integrated behavior". What is more, these fragments are genetic in nature – they (as a complete set) have the interesting property that they contain enough information within the set to support their integration to create a DBT. In some respects the GSE process is like solving a jigsaw puzzle, where the solution is built out of the pieces. With both GSE and solving a jigsaw puzzle, the key thing is the *position* where each piece is placed. Behavior trees make it possible to identify that position.

To take the GSE process through to the implementation stage we must transform the design behavior tree into its corresponding software architecture (or component integration network, CIN – that shows all the dependencies among all the components needed to implement the behaviour in all the requirements of the system) and project from the DBT the component behavior trees (CBTs) for each of the components mentioned in the original functional requirements (in this case we would have CBTs for the SUPPLIER and the PURCHASER). Due to space limitation these steps are not described here but examples are given in [2].

2.3 Detection of Specification Defects

Translation of Contract requirements into behaviour trees and subsequent integration of those behaviour trees provides a powerful means for detecting defects in a contract. As an example, what we find in translating the payment clauses, 5.1 and 5.2 is that, while they set some constraints for the payment process they leave out important steps and they make essentially no provision for what should happen when the constraints and other important implementation steps are not met. The specification strategy for dealing with this particular contract style therefore needs to involves the following.

- Introduction of new behavior implied, but not explicitly required by the payment clauses, that is necessary and sufficient to implement the payment process both from the purchaser and supplier's perspectives.
- Integration of the constraints specified in 5.1 and 5.2 into the overall payment process.
- Augmentation of the payment processing process with additional behavior to properly accommodate what should happen when the constraints and possibly other implementation conditions are not satisfied.

What is important, and what the behavior tree notation allows, is the clear delineation of what was *explicitly expressed* in the contract. It is also possible to separately identify what is *implied* by the contract but which is not explicitly stated. Finally, we can identify behavioral *incompleteness* problems where an alternative situation is possible, but the contract has failed to specify what should happen. We do this most often by examining decision nodes or event nodes or leaf nodes in the integrated design behaviour tree for missing alternatives or additional behaviour. For example, clause 4.2 talks about goods being available within one day. Clearly, we need to accommodate the case where it is not available within one day. Other particular examples of defects found when clauses 5.1 and 5.2 are fully translated into behaviour trees are:

- No provision has been made for the supplier to send an invoice to the purchaser.
- No provision has been made for the purchaser to check that the invoice is correct.
- There are several other instances where it is not clear what should happen next when a certain situation arises. For example, if the invoice sent to the purchaser is incorrect, according to Purchaser, then it is unclear, from what has been provided in the contract, what should happen next.
- Another example is that it is unclear how the Supplier should proceed, when the payment is late or payment is not made in full.
- A mechanism is also needed to charge interest and receive the payment.

2.4 Incorporating Policy External to a Contract

It is important within the contract management framework to be able to properly accommodate company-wide policy that sits above or outside individual contracts. There are three strategies that can play a role in implementing organizational internal policy in contract execution: (1) when a policy related event arises control can be passed to a human operator who has the responsibility to recognize what action and input to the system is needed to implement company policy, or (2) we can formalize required policy as behavior using either the GSE behavior trees or the Policy Language method (see next section) and let the contract system take care of it automatically, or (3) we can use a mix of the first two strategies that will give us the best mix in terms of flexibility, productivity and outcome, and (4) additionally, we can choose to build up our store of formalized Policy behavior on a needs basis. This way, over time, our library of formalized and reusable Policy Behavior grows in increments without a large up-front investment. Total formalization of company policy is likely, in most cases, to be either impossible, or not worth the investment of effort needed.

When the behavior tree representation is used for incorporating external Policy we can proceed as follows. First we can build up a store of policy requirements that have been encoded as behavior trees. Second we can integrate and adapt the appropriate sub-set of policy behavior fragments into each contract as required. Our requirements for the application of Policy are that they need to be easily deployable to many different contracts. Furthermore only parts of the behavior in any given policy may be relevant to, and applicable for, a given contract. It is therefore important that we have a flexible and efficient way of adapting or tailoring policy behavior to any particular contract. One way to satisfy these requirements using the GSE method is to functionally integrate the behavior tree fragments that define that part of company policy that has been formalized. This gives us an integrated design behavior tree for available formalized policy. We then extract, by behavior tailoring, the Policy behavior that is needed for a given contract. Finally, we manufacture (using a largely automated process) a *contract-specific* policy component that we in turn integrate into the component-based system configuration needed to implement the particular contract. This may require application of a Behavior Alignment strategy where we work out correspondences between generic components and states and the particular components and states that need to be instantiated for the particular policy.

Consider the following policy statements in English:

1. Internal Supplier Policy – Valued Customers
(a) Give 10% discount to valued customers (b) Valued customers are those with total purchases > $1,000,000 (c) Lose valued customer status after three consecutive late payments.

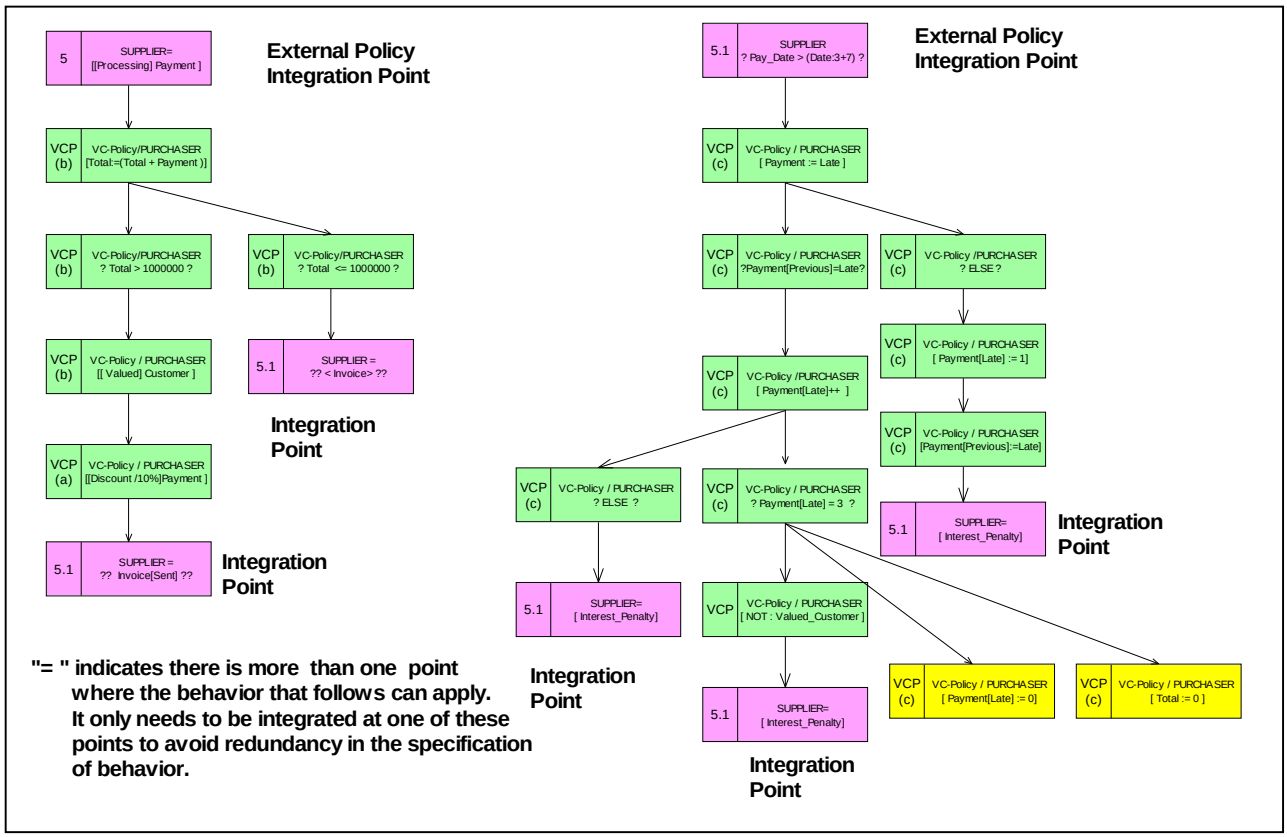

Figure 5: Integration with external policies (e.g. company's internal policies)

The behavior required to implement this external Valued-Customer Policy (VCP) needs to be implemented by two behavior fragments (Fig. 5), one that gives the discount and makes an update on any purchaser that meets the criteria for valued-customer status -VCP (a),(b) and a second that keeps track of consecutive late payments and removes valued-customer status from a purchaser when they make three late payments – VCP (c). These two reusable Policy Implementation Behaviors need to be integrated into the Contract behavior tree at two separate integration points as indicated in Fig. 5. A Behavior Tree maintenance tool can support this external policy application integration into a contract can either be automated or conducted under user control.

In expressing this required behavior we have chosen to express it close to the implementation level. We could equally well have chosen to express it at a much higher level, closer to the natural language Policy statements by incorporating essentially direct translations for states like "?[[Consecutive] Late]Payment = 3? and ?Total > 1000000?. This would have produced a much briefer but less informative specification. This sort of choice is always open to the specifier when using behavior trees. The Valued-Customer Policy behaviour tree introduces another small piece of notation associated with component-states. If we want to talk about the high-level behavior "supplier processing payment" we could use SUPPLIER[Payment[Processing]]. However to improve readability we can equivalently express this as SUPPLIER[[Processing] Payment]. A similar thing has been done with consecutive, late, payments, above.

3. Policy based contract monitoring

The GSE method enables expression of behaviour of parties to the contract in a way that would satisfy their obligations and other policies as stated in a contract. However, the nature of systems in which people and organisations are in decision loop, is such that some behaviour may not be executed as agreed by the contract. Thus, although the mapping of a contract into the corresponding integrated design tree provides a basis for prescribing parties' behaviour as far as this contract is

concerned, this may not be sufficient to guarantee that such behaviour will eventuate. To deal with such situations a monitoring mechanism should be put in place to compare desired and actual behaviour (Fig. 6).

This section begins with a brief description of differences between enterprise and computational specifications. It then introduces key monitoring aspects of our contract management architecture, in particular our policy language.

3.1 Enterprise vs computational specification

Enterprise specifications are needed for describing policies and behaviours of enterprises, both in terms of their internal policies and policies arising from contracts with other parties. Following the spirit of ODP Reference Model, an enterprise specification provides a way of describing business problem and a basis for subsequent specifications addressing information and computational aspects of the software, as well as key technology aspects.

As stated in [3], one way in which an enterprise specification differs significantly from a computational specification is in the probable degree of inconsistency that must be expected to be present. While a computational design that is inconsistent can be rejected by a design tool until corrected, an enterprise specification may often place conflicting constraints or requirements which are defeated by changes in real world. Take for example the real life situation regarding driving rules. A rule stating that one is prohibited to drive through a red light or drive over the speed limit can be defeated in the case where there is a seriously ill person. Thus, when considering enterprise specification, the set of constraints needs to be seen more as a set of objectives to be managed than a rigid structure that deadlocks if any element is violated [3]. A more detailed analysis of the differences between enterprise and computational specification can be found in [4]. We note that every enterprise specification can be reduced to some computational specification – but using computational specification for describing enterprise systems can be often impractical. This is because it can lead to unmanageable number of possible options that reflect decisions of actors in the enterprise setting – and often even not being able to guarantee that all possible cases have been taken into account.

A key area of enterprise specification of relevance for contracts is specification of policies, in particular those policies that represent constraints on behaviour. These are policies that express obligations, permissions and prohibitions of parties, as specified by contract. We note that some other policy specifications, such as configuration information (e.g. structure of contracts and dependencies between their clauses) are relatively straightforward to specify and are not discussed here.

3.2 Language for describing enterprise policies

The policy language we use as part of contract monitoring is based on the ODP standards, our policy framework developed in [4] and on the work of Steen and Derrick, [5]. This policy language is developed to closely follow English language representation for typical behavioural policy statements that are also used in contracts. Most of these policy statements express constraints in terms of obligations, permissions or prohibitions. We refer to such statements as *deontic* statements as inspired by a special branch of modal logic, viz deontic logic, that is concerned with the problem of reasoning about the notions of obligations, permissions, prohibitions, authority and so on. Accordingly, in this paper we will refer to our policy language as *deontic policy language* (DPL).

In addition to its style being influenced by natural language used in contracts, the DPL needs to be suitable for interpreting behaviour execution so that it is possible to perform run-time evaluation of policy compliance. Essentially, the DPL is a specialised behavioural language which gives prominence to the concepts of Role, Modality (obligations, permissions or prohibitions), Action, Temporal Conditions and other Conditions that need to be fulfilled for a behaviour to satisfy policy. In its simple form the DPL grammar is as follows:

```
Policy <PolicyIdentifier>
A <RoleIdentifier> is
  ( permitted|obliged|forbidden )
    to(do<Action>|satisfy<Condition> )
      [,temporal <TemporalCondition> ]
      [, if <Condition> ]
      [, where <Condition> ]
```

The *if* clause above is used to express conditional policy statement. The *where* clause is used to specify parameters of Action and *temporal* clause expresses various temporal constraints (duration, relative or absolute time).

Thus, DPL is intended to describe constraints on behaviour of contracting parties and its emphasis is on describing what the parties are permitted, prohibited or obligated to do, under various temporal and other conditions.

We observe that typical policy statements expressed in natural language can be often impersonal. i.e. the responsible actor is not mentioned explicitly. This is because the actor may be implied by the outer context –

being it the clause where this policy is defined or as stated in some other clause of the contract. For example all policy statements from section 2.4 (given in the Text Box) are of impersonal nature. We note that this common style of expressing policies in contracts, along with the usage of implied and context-dependent expressions can often be a source of ambiguity in contracts.

In addition, natural language contracts often contain expressions of certain conditions relating to a state of some entity. Since the DPL is essentially a behavioural language where the actions of actors are of primary interest, the state information is dealt with through checking other conditions that can be included as part of *if*, *where*, *satisfy* and *temporal* conditions in the DPL. For example, policy *b)* from section 2.4 is a definitional policy related to state about items purchased. This state management is dealt with in other parts of our contract management system (referred to as internal structures in Fug. 6)..

The process of run-time policy interpretation rests with the interpretation of policy statements in the above form but the interpretation is also dependent on other mechanisms. In addition to the mechanisms used to maintain and interpret state information as mentioned above, this may include mechanisms to process events of various kinds and mechanisms for accessing data from the local or remote enterprise systems - as part of an overall contract management system.

We note that there may be other policies stated in contracts which are not directly of deontic nature, e.g. various configuration constraints and these policies need to be also checked. These are beyond the scope of DPL.

We illustrate the use of DPL with the examples of Valued Customer internal policy statements and Service Contract policy statements (as included in policy fragment in the the TextBox in section 2.4).

Consider first *Policy (a)*. Note that in its English description it is implied that a Supplier has an obligation to give 10% discount to a valued customer. It is also implied that this should be the case whenever she purchases an item. It is also assumed that a valued customer is defined elsewhere (indeed this is what *Policy (b)* states). Evidently, Policy (a) can be regarded as a shorthand expression of a more complete statement. It is only that such complete statements can be tested in run-rime and these complete statements are supported by DPL. Thus, according to the DPL:

Policy (a): A ***Supplier*** is ***obliged*** to ***give_discount*** **where** (customer = Valued Customer)

When considering *Policy (b)* one concludes that this policy does not *directly* specify a constraint on behaviour

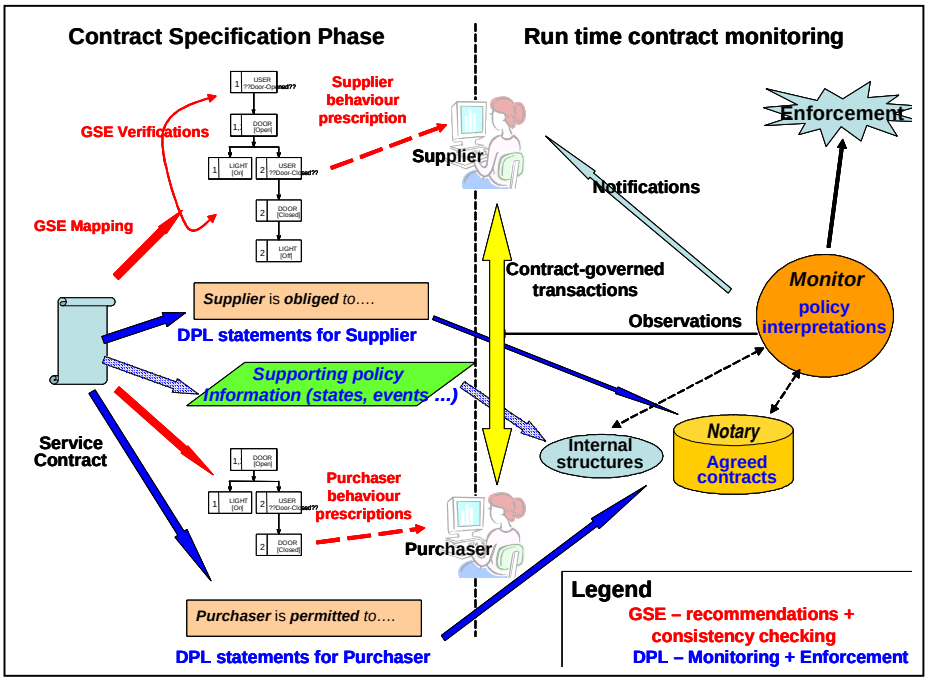

Figure 6: Positioning of GSE and DPL methods

in its deontic sense (for which DPL is designed). In fact, this policy is of a definitional nature using information about state (> $1, 000, 000) which is managed elsewhere in the contract management system.

In relation to *Policy (c)* it is obvious that someone needs to change the status of a valued customer if the new condition is met (i.e. NumberOfLatePayments > 3). It is natural to assume that this is done by the Supplier role (in case a Supplier is an organisation, this can be a role which is authorised to act on behalf of the Supplier). In this case, the policy can be interpreted as an obligation that the Supplier should do this in any case. However, considering that there may be other factors that can influence the decision of the Supplier (e.g. the Valued Customer Purchaser changed their IT payment infrastructure, which caused the delay in the third payment) the ultimate decision is under the discretion of the Supplier. Therefore, the policy can be stated in its weaker form, in terms of a permission. Thus, the corresponding DPL expression is:

Policy c: A ***Supplier*** is ***permitted*** to do ***remove_customer*** (Customer) ***where*** (Customer.NumberLatePayments > 3)

Similarly, the policies in two Payment clauses of Clause group 5 from contract example are stated in DPL as:

Policy 5.1a A ***Purchaser*** is ***obliged*** to do ***Purchase_Payment temporal*** CurrentDate ***after*** Recipt_of_Invoice

Policy 5.1.b A ***Supplier*** is ***permitted*** to do ***Charge_interest*** *where* (Date > Invoice_Date + 7)

Observe that the clause 5.1 includes several policies, and we show two of them, i.e. 5.1a and policy 5.1b. *Policy 5.2* can be expressed as:

Policy 5.2 A ***Purchaser*** is ***obliged*** to do ***Send_Payment_electronically satisfy*** (PayPal rules)

These examples show how English language format of the Service Contract payment policies can be expressed in the DPL form.

Our experience with examining many forms of behavioural-oriented policies so far (i.e. in the deontic sense) suggests that most of such policy statements in natural language can be reduced to the form above as supported by the DPL grammar. The grammar is being extended to better support temporal constraint expressions (making use of some of the results from [6]) and also relationships between different policies. The latter will also enable expressing dependences between policy constraints of contracts and other related fragments of behavior

3.3 Key roles supporting monitoring

The DPL statements for specific policies are interpreted by a Contract Monitor (CM) component. The role of CM is to observe actual behaviour and compare it to the agreed behaviour in contract. The signed contract instances are stored in a repository referred to as Notary (see Fig. 6). There are several other components in the BCA as introduced in [8], augmented with necessary infrastructure components that support description of other policy-related information such as state and event related contract data.

4. Discussion

This section provides several discussion points assessing the use of GSE method in the contract domain and how the two methodologies presented in this paper can be used synergistically.

The behavior tree notation and supporting GSE Method appear to have a number of characteristics and capabilities that make them suitable for application in contracting applications. There are several areas where GSE can make its most useful contributions:

- Behavior trees appear to have enough expressive power to accommodate the wide range of behaviors and constraints that are needed to represent contract semantics for automated management applications.
- GSE is an effective tool for finding incompleteness and inconsistencies in a contract. It is essential that these problems are uncovered and resolved before the contract is deployed. The payment component, and the rest of the Contract for Services example, considered above, provide evidence for this capability.
- Behavior tree descriptions of contracts employ a simple intuitive notation and have direct traceability to the original contract. This makes them easy to read and verify by end-users, and legal people, without a large investment of time to master the notation.
- The behavior tree notation also has the advantage that it may be easily simulated and/or mapped to object-oriented or component based implementations or to use in distributed component frameworks.
- Similar to large software systems, some enterprise contracts can have a significant amount of complexity and detail. GSE is easily able to accommodate the needs in this area.
- Perhaps more than any other notation for expressing behavior the GSE method is much more easily able to accommodate and trace changes. In some contract

application areas it is extremely important to have this capability and flexibility.

It is essential to employ powerful tool support when using behavior trees and GSE in order to maximize the benefits from using the method. Currently there is some tool support for software engineering applications. We are presently assessing how this technology can be leveraged, adapted and integrated into the existing contract management environment that supports the implementation of the Policy language and overall contract monitoring.

Fig. 6 depicts the application of the GSE and DPL approaches at various stages in contract management. The left part of the diagram shows the use of GSE method to specify expected behaviour of parties to the contract. This behaviour representation can be used to generate various kind of notifications to the parties to execute their respective actions that are pending, e.g. a reminder to a role in the Supplier organization to send an Invoice to the Purchaser or a notification that a contract is approaching an end-date and needs renewal. In other words, the behaviour trees produced by the GSE method can be regarded as guidelines (or prescriptions) for the parties to execute actions as specified by the contract. The diagram also shows our current approach to support contract monitoring based on the use of DPL. Similarly to the GSE method, we derive policy statements for the monitoring based on the natural language description of contact. Currently, this is done manually, although we are planning to investigate various options for an automated tool to support this editing. Policy descriptions are stored in the Notary component. In addition to the policy descriptions, there are many other configuration-like information which are needed to describe particular contractual situations in terms of contract-significant events, states and notifications. These are then stored in various internal structures within BCA. During service execution, the BCA Monitor is observing behaviour of parties to the contract and other contract-related information from the enterprise systems environment and compare these actual behaviours with those that are specified in the contract and stored in the Notary. If a non-compliance is detected, the Monitor will signal this information either to the parties to the contract or to an enforcement system for some corrective actions. This can be a complex system that supports multiple escalation levels, ultimately involving a human decision maker, as discussed in [9].

5. Conclusions and Future Work

This paper has presented two methods that can be used for specifying behaviour of contracting parties and supporting monitoring of run-time aspects of contracts, including obligation monitoring of parties to the contract. The power of Genetic Software Engineering methodology - initially developed for the software specification domain - can be exploited during the contract specification stage. The key value of Deontic Policy Language approach is in supporting the run-time phase of contract execution, in particular contract monitoring. When used in combination, these methodologies provide the basis for automating key contract management activities.

Our next step will be to further integrate these two separately developed methods. The architectural underpinning for experimenting with the methods is mostly likely to be a generic framework of Business Contracts Architecture initially presented in [7], [8].

References

[1] Open Distributed Processing: Reference Model - Part2: Foundations, Int. Standard 10746-2, ITU-T, Rec. X.902

[2] R.G.Dromey, Genetic Software Engineering - simplifying design using Requirements Integration, IEEE Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Dec 2001.

[3] P.F. Linington, Options for Expressing ODP Enterprise Comminities and Their policies by Using UML, Proc. of the 3rd International Conference on Enterprise Distributed Object Computing, Sept. 1999, Manheim, Germany.

[4] P.F. Linington, Z. Milosevic, K. Raymond, Policies in Communities: Extending the ODP Enterprise Viewpoint, Proc. of the 2rd International Conference on Enterprise Distributed Object Computing, Nov. 1998, La Jolla, California.

[5] M.W.A Steen, J. Derrick, Formalising Enterprise Policies, Proc. of the 3rd International Conference on Enterprise Distributed Object Computing, Sept. 1999, Manheim, Germany.

[6] O. Marjanovic, Z. Milosevic, Towards Formal Modelling of e-Contracts, Proc. Of EDOC2001 conference, Seattle, USA, Sept.01.

[7] Z. Milosevic, A. Bond, Electronic Commerce on the Internet: What is Still Missing?, the 5th Annual Conference of the Internet Society, INET'95, Hawaii, USA, June '95.

[8] Z. Milosevic, Enterprise Aspects of Open Distributed Systems, PhD Thesis, Computer Science Dept., The University of Queensland, Oct.1995.

[9] Z. Milosevic, A. Josang, T. Dimitrakos, M-A. Patton, Discretionary Enforcement of Electronic Contracts, Proc. Of EDOC2002 conference, Lausanne, Switzerland, Sept.02.

Appendix: Example Contract

CONTRACT FOR SERVICES

This Deed of Agreement is entered into as of the Effective Date identified below.

BETWEEN ABC Company
Suite 100, Tall Towers, Surfers Paradise, Queensland, Australia
owner@abc.com

(To be known as the **Purchaser**)

AND: ISP Plus
1 Ocean Street, Mermaid Beach, Queensland, Australia
herring@dstc.edu.au

(To be known as the Supplier)

WHEREAS (Purchaser) desires to enter into an agreement to purchase from (Supplier) Application Server (To be known as (Goods) in this Agreement).

NOW IT IS HEREBY AGREED that (Supplier) and (Purchaser) shall enter into an agreement subject to the following terms and conditions:

1. Definitions and Interpretations

1.1 Price is a reference to the currency of the Australia unless otherwise stated.

1.2 This agreement is governed by Australia law and the parties hereby agree to submit to the jurisdiction of the Courts of the Queensland with respect to this agreement.

2. Commencement and Completion

2.1 The commencement date is scheduled as January 30, 2002.

2.2 The completion date is scheduled as January 30, 2003.

3. Purchase Orders

3.1 The (Purchaser) shall follow the (Supplier) price lists at http://supplier.com/catalog1.html.

3.2 The (Purchaser) shall present (Supplier) with a purchase order for the provision of (Goods) within 7 days of the commencement date.

4. Service Delivery

4.1 The (Supplier) shall ensure the (Goods) are available to the (Purchaser) under Quality of Service Agreement High (http://supplier/qos1.htm).

4.2 The (Supplier) shall on receipt of a purchase order for (Goods) make them available within 1 days.

4.3 If for any reason the conditions stated in 4.1 (a) or 4.1 (b) are not met, the (Purchaser) is entitled to charge the (Supplier) the rate of $100 for each hour the (Goods) are not delivered.

5. Payment

5.1 The payment terms shall be in full upon receipt of invoice. Interest shall be charged at 5 percent on accounts not paid within 7 days of the invoice date. The prices shall be as stated in the sales order unless otherwise agreed in writing by the (Supplier).

5.2 Payments are to be sent electronically, and are to be performed under standards and guidelines outlined in PayPal.

6. Termination: NOT SHOWN TO SAVE SPACE.

7. Disputes: NOT SHOWN TO SAVE SPACE.

SIGNATURES

[Signature] [Signature]