International Journal of Cooperative Information Systems

AUTOMATING SUPPORT FOR E-BUSINESS CONTRACTS

PETER F. LININGTON
Computing Laboratory, University of Kent, Canterbury
Kent, CT2 7NF, UK

Received (16th November 2004)
Revised(28th February 2005)

If e-business contracts are to be widely used, they need to be supported by the IT infrastructure of the organizations concerned. This implies that the interactions between systems in different organizations must be guided by the contract and there must be sufficiently strong checks and balances to ensure that the contract is in fact obeyed. This includes facilities for the unbiased monitoring of correct behaviour and the reporting of exceptions.

One of the ways to provide this support is to generate it directly from the agreed contract. This paper considers the steps necessary to provide sufficient automation in the support and checking of e-Business contracts for them to offer efficiency gains and so to become widely used. It focuses on the role of models, taking a model-driven approach to development and discussing both the source and target models and the transformational pathways needed to support the contract-based business processes.

Keywords: monitoring; contracts; model-driven development.

1. Introduction

In the real world, business activities during which organizations cooperate are regulated by contracts. These are agreements on the patterns of behaviour needed to achieve mutually agreed goals, and often include contingencies and sanctions to be applied if the expected behaviour is not performed correctly and on time. These contracts are governed by rules or laws established by the society in which they are agreed. It is highly desirable for the ICT infrastructure supporting business activities to be controlled directly by some expression of these contracts, so that correct operation is assured with a minimum of human intervention.

However, each participating organization has its own agenda and, although the contract represents a mutually acceptable outcome for all the parties involved, it is not generally the most advantageous outcome for any of the organizations if they are considered separately; each must therefore have some assurance that the others are keeping their side of the bargain, and not deviating from the agreement to maximize their own gains. Reflecting this division of interests and responsibilities, the infrastructure will itself also consist of parts serving each organization and

possibly of other parts operated by third parties; each party will need some way of checking that the others are indeed operating according to the contract.

Previous work has proposed an architecture for contract management within the ICT infrastructure[15], for expression of the contract as a set of policies[13] and for a monitoring component that can be used to check adherence to the contract[17]. Work on the Business Contract Language (BCL)[12] has proposed a language for expressing such contracts in a form suitable for a checking component to operate on. However, the proof-of-concept prototypes constructed in the course of that work were hand-built and hand-configured. A better solution is needed if electronic contracting is to be cost effective; one option is to use the approach of Model Driven Development, based on effective integrated tooling to support software construction, and this is the approach discussed in this paper.

The remainder of this paper is structured as follows. Section 2 gives some background to Model Driven Development and section 3 outlines the ways in which contracts are used. Section 4 analyzes the requirements for expressing and manipulating contract support. Section 5 outlines features of the Business Contract Language. The next two sections address the requirements for the main metamodels and their interpreters; the notification metamodel is described in section 6 and the monitoring metamodel in section 7. Section 8 provides an overview of the transformation process, section 9 discusses other previous work in this area and section 10 draws conclusions.

2. Model Driven Development

The key to the flexible evolution of ICT systems is automation, particularly to reduce the amount of human intervention needed, and this applies strongly to the automation of the production of implementation detail. What is needed is a way to establish an implementation style, expressing it as a transformation template. This template is then used in each specific case for elaboration of the high level design. Future modifications to the business design can then be carried through mechanically, with the minimum of human intervention and duplication of effort, into changes to the detailed implementation of the infrastructure by reusing the same transformation template. This is the concept behind the model driven development movement. Much of the interest in this style of development has arisen from the OMG's definition of its specific version, the Model Driven Architecture[19].

In the model driven approach (figure 1), the system designers have two kinds of task to perform. Firstly, they have to generate a design in terms of a model that abstracts away from the details of the supporting infrastructure; that is to say, a model in business terms. They will create this model using a suitable domain-specific language, or metamodel, using appropriate tools that are aware of the metamodel to manipulate their model correctly. Second, they will need to define a transformation from their model to a running solution that uses available infrastructure components. If the business metamodel is stable and reasonably well known, these two

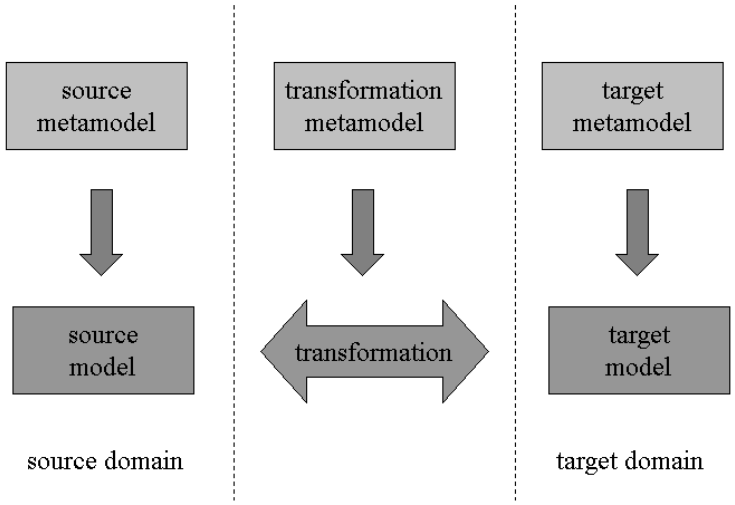

Fig. 1. A transformation between models.

design tasks can be performed by different specialists within the team; the source metamodel needs to be developed by business specialists, the target metamodel, which is also needed, will be supplied by infrastructure providers and the transformation between them will be defined by infrastructure specialists who are aware of business constraints and conventions.

In model driven development, it is assumed that the source and target metamodels are comparatively stable, so that transformations expressed between them become reusable, and the tools generated to perform these transformations are then themselves long-lived. The transformations are likely to be constructed from reusable component transformations which take the form of broadly accepted templates or patterns. Thus any contract expressed in the contract language can be mapped to the reusable checking components by using the transformation metamodel to construct a specific transformation, without the designer having to intervene in the transformation process on a case by case basis.

However, to apply this style to any particular domain, we need to have available suitable target metamodels, and encouragement of reuse dictates that these should have as broad a scope as possible. They are likely to be produced to reflect the properties of the available platform architectures or of general-purpose components.

One way of looking at the process is to see the target metamodel or metamodels as defining a virtual machine on which the source behaviour is to be executed. If some required behaviour is expressed in terms of rules for the interpretation of a series of tokens by progressively testing and updating a body of contextual state, we can generally think of the set of rules as defining a virtual machine; the rules and the set of input events are interpreted to check validity and to generate a set of

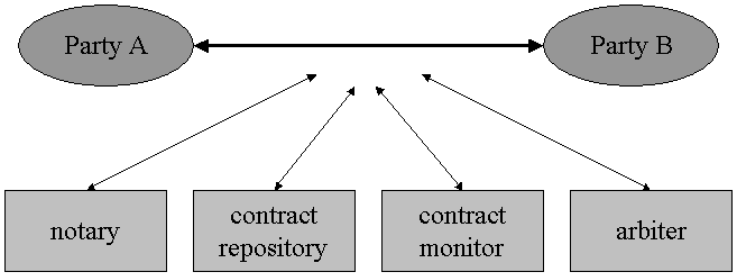

Fig. 2. Different uses of electronic contracts.

outputs, which, in this case, will be reports of successful completion of the contract or of violations of it. If the behaviour we are interested in can be considered at a number of different levels of abstraction, then we can think of a corresponding family of virtual machines, linked by refinement relations.

In the case of the monitoring of a contract, we can consider an abstract virtual machine that polices the general form and framework of the contract; this machine can be refined as necessary to capture the negotiation of specific options on different occasions, and a corresponding, more specific, virtual machine is defined and instantiated to monitor each variant as it is negotiated. These specific definitions and their instantiations can be short lived, lasting for just as long as the negotiated agreement holds.

Although this paper concentrates on the application of model driven development to contract support, it is being applied to a progressively wider range of problems including test generation[4], performance analysis[25] and security support[11].

3. Use of Electronic Contract Support

Before turning to the application of these ideas to electronic contracts, we must first identify requirements by looking at the ways in which electronic contracts can be used. It is possible to identify a number of different uses and corresponding different ways in which the representation of a contract will be applied (see figure 2). These contract representations can be used:

(i) to record the results of negotiation between the parties involved, or their agents, thereby creating a contract defining some activity that is to be undertaken either immediately or at a later time; this can be on a one-off basis, supporting just a single activity, or it can result in a contract that will be applied more than once. A common example is the call-off agreement, which is negotiated and then holds for a specified period, during which time the agreed terms may repeatedly be invoked for the supply of specific goods or services;
(ii) to steer the performance of activities while carrying out the contract; the con-

tract is used in identifying obligations and in scheduling resultant actions or identifying situations where a specific response is needed to violations;

(iii) as a basis for run-time monitoring activities, which may be carried out by components or organizations separate from the parties directly involved in the contract;

(iv) during subsequent arbitration of disputes arising from the contract; this is likely to be based on the audit trail established by the participants, together with non-repudiable statements lodged by them with a mutually agreed trustworthy repository.

A single model should be able to represent the contract in each of these cases, but, each time, a specific model and metamodel for contract processing will be needed; this is because the actions to be taken will be based on an interpretation of the contract that will differ in each case, and so specific corresponding actions will need to be defined for each field of application.

Consider, for example, the supply of telecommunication services, governed by a service level agreement. The agreement needs to be negotiated, probably by selection from a proforma and completion of specific detail, and the agreed choices need to be recorded. Some aspects, such as time-varying constraints on the load the customer can generate, need to be made available to the customer at run-time; the active parties and the monitor all need to be aware of any obligation that might be placed on the active parties to provide periodic performance information; some statistical constraints may be checked by the monitor but not supported at run-time by the customer systems; and, finally, longer term properties, such as availability, may need to be assessed from extended historical records in response to customer complaint to a regulator or other independent body. All these activities are still taking the same negotiated contract as their basis.

4. Requirements on the Metamodels

The description of contact monitoring can be divided into two parts, each with its own metamodel, corresponding respectively to the collection and interpretation of significant events. This subdivision helps to separate the different parts of the target domain, the resulting metamodels being:

(i) *the notification metamodel*, which encapsulates requirements that apply to the infrastructure for the contracting parties, including how they are to supply reports on key events, and the information content these reports must have. There will be two kinds of information, covering the generic information common to all reports, such as the event type, a timestamp and the source and destination of the report, and further details of the nature of the event reported, such as description of goods and their value, the definition of which will need to be imported from the contract description.

(ii) *the monitoring metamodel*, which describes requirements that influence the de-

cision process to be carried out by the monitor, in which information about the actions taken by the parties is used to determine whether the activity conforms to the agreed contract or not. It includes the interpretation of the contract-specific content of events, when parties are obliged to report them, and the acceptable behaviour of the parties in terms of these events.

The support for the contracting parties will consist of some message generation mechanisms, consistent with the event definitions in the monitoring metamodel, and a general message transport mechanism, consistent with the generic notification metamodel. The implementation of a monitoring component will interpret contract descriptions in a way that is consistent with the monitoring metamodel.

4.1. *More detailed requirements*

To clarify the roles played by different metamodels, let us consider the monitoring process, and the environment in which it is carried out, in rather more detail. Sufficient detail need to be considered to justify the metamodels proposed as being well fitted to their purpose, allowing accurate decision making and providing sufficient information to allow the efficient engineering of reusable monitoring components.

We assume for simplicity that a contract has been negotiated, and that it has been signed by a notary and lodged in a trusted contract repository. From there, it is accessed independently by the contracting parties to guide their activities and by the monitor to verify that the actions taken are consistent with the contract.

The contracting parties need to be able to determine at any point which actions are permitted, which of them are definitely required to fulfil some obligation, and, for these obligations, how soon action is required and how severe the penalties for failure to comply with the contract would be. They also need to identify which special actions participants are obliged to perform to demonstrate progress, and to whom the progress information is reported. The reporting actions are likely to be distinct from the significant business steps, and may exist solely to enable monitoring, although they may also be necessary to form part of some subsidiary customer service, such as order tracking. For example, the main business process may be in terms of a physical *dispatch-of-goods* event, but the contract may require the generation of a corresponding electronic counterpart in terms of an event notification or the posting of a state change to the customer progress system.

The reporting requirements are likely themselves to form part of the contract, and may imply a requirement for the implementation to provide reporting either to a specific entity or to a well-known channel, defined so as to make key events visible to some or all of the participants on an opt-in basis. Depending on the nature of the event being reported, this may require additional procedures by human operators to ensure that significant state changes are accessible to the IT infrastructure. This may need to include reports of disruption of the contractual processes, or of the infrastructure, or of instances where the terms of the contract are set aside by Force Majeure.

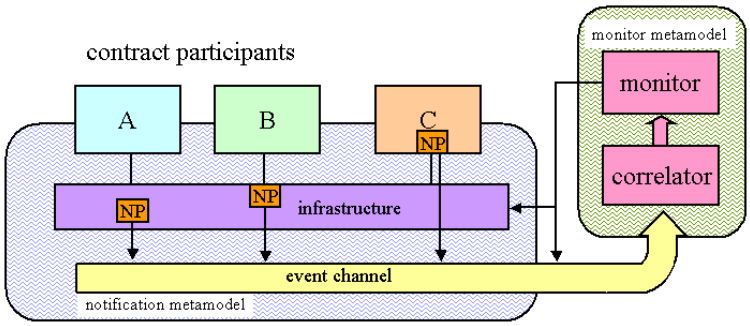

Fig. 3. Outline of the Monitoring Architecture.

An example of this can be found in many commercial ports, where port fees are payable by the minute from the time a ship is first moored to the time the last rope is released on its departure. These events are the result of an essentially manual process, and are not easily determined by the electronic parts of the system. Attempts to report them automatically in a robust way have had little success and the definition of the terms of the contract cannot easily be varied because of the importance these events have come to have in many interdependent regulations and agreements. Manual entry of reports is then the only viable alternative.

A typical monitoring architecture is illustrated in figure 3. This shows the systems of three contracting parties A, B and C, together with a separate monitoring component. The contracting parties are supported by a distributed infrastructure, which might be an object or service oriented middleware or a more loosely coupled message queuing and passing system. The parties can discharge their reporting obligations in a number of ways (so long as the necessary information is visible – before, for example, any message encryption needed to protect communications). C incorporates explicit reporting in the application implementation, B uses interception services associated with its access interface to the infrastructure, and A, which might be a less accessible legacy system, relies on interception rules in the core of the infrastructure. These pieces of function are shown in the figure by the small rectangles labeled NP (for Notification Point); they relay the events detected to the monitor using some form of event channel, which might, if appropriate, be integrated with the production infrastructure, but might be a separate, more easily auditable, overlay service.

The monitor, being a separate system, is based on independent design choices, and so makes a minimum of assumptions about the nature of the infrastructure and the event channel. The monitor receives a stream of low level events which are first pre-processed (see below) by an aggregation and correlation element, labeled *Correlator* in the figure, which recognizes events at the level of abstraction appropriate to the contract and passes these higher level events to the correct instance of the

monitor's main matching engine. Depending on the closeness of coupling between the monitor and the parties being monitored, the monitor may then generate interactions with the parties to report violations; it may also re-insert events into its own event stream to signal progress or violation to other contract instances.

The monitor needs to be able to determine whether observed actions are valid at the time and place where they occur, or not, and what effect they have on the state of, and progression of, the contract. The monitor needs to record enough information about the state of each activity to be able to perform this kind of validation.

So why is the distinction between the Monitor and the Correlator made? It exists because a contract will generally leave the participants free to determine the details of their supporting system designs independently. There may well then be a mismatch between the events observed and the actions declared in the contract. This can occur because the contract is expressed in more abstract terms than the actions reported, so that the monitor has to match patterns representing the more abstract events to the input event stream in order to recognize the abstract events. These patterns need not be fixed in the contract, since contracting parties will generally have autonomy in determining how they are to perform contractual actions and what infrastructure they are to use, but the way significant events are reported must be communicated by the parties to the monitor in some way. This can be done by declaration of the method to be used at negotiation time, or during run-time initialization; it may be by selection from a set of foreseen optional formats, or by specification in a generally accepted schema language, such as one of those associated with XML, or a service definition language (although the expressive power of many of the solutions currently on offer is rather limited).

Another reason for there being a non-trivial mapping between observation and contractual action is that there may be delegation, for example to a sub-contractor, so that there would again be flexibility as to how the contractual action is to be achieved in detail by the sub-contractor. In this case, the details of the required event representation must be provided by the sub-contractor.

In any of these cases, a mechanism is needed to support the dynamic binding of detailed behaviour to the contractual actions. What the binding actually is might be determined by pre-registration or by inspecting the parameterization of initial exchanges in the contract, where details are being negotiated.

The combination of the freedom to declare reporting formats and the ability to re-inject events to the generating monitor, or to one or more other monitors, leads naturally to the ability to federate monitors and so to handle dependencies between separate contracts. It would also be possible to link monitors to other consensus-building tools, such as reputation servers, and then use the information obtained to optimize the monitoring process.

4.2. *Correlation Requirements*

Another area where significant flexibility is needed is in identifying when new instances of the contractual behaviour begin, and which of the instances of the contract subsequent actions are to be associated with. This is particularly important where several instances may be being executed concurrently. The situation is quite similar to the problem of identifying correlation sets in a choreography language like BPEL[24], but with the additional problem that hierarchical interpretation may require several steps of binding from basic to more abstract actions to be interpreted before the key information for identifying the correlation set is available.

In simple cases, the correlation requirements might be met by ensuring that all the messages monitored included key items, such as the initiator's identity and order number. This information could then be used to dispatch events to the appropriate monitor instance. Although this is straightforward if the message formats were designed with monitoring requirements in mind, it is more complex if designers have chosen to exploit the creation of some form of session between parties to eliminate common information from messages, and instead using a local handle as a reference. If this is done in a way that is not completely visible to any observers, it may be difficult for them to track the session life-cycle.

Thus, in more complex cases, this can lead to a need for the monitor to carry forward a number of possible interpretations, and to prune incorrect guesses when further information becomes available. Consider, for example a contract that includes a sequence of actions involving some part of the infrastructure that, for legacy reasons, does not support the correlation identifier used in the initial activities of the contract instance (see figure 4).

This figure shows a situation where the initial exchanges use as the correlation identifier a value ID_A included in the initial message by party A. However, the legacy exchanges between party B and party C cannot convey this item, and so correlation is based on the value ID_X originated by party B; this is not a problem for party B, which maintains a local mapping between ID_A and ID_X, but this mapping cannot be inferred with any certainty by an external observer, particularly when concurrent instances of the contract are in progress. The observer can correlate the final exchanges between parties A and B with the initial ones, but can only correlate exchanges involving A-B and B-C if there is some other suitable data item, such as D, which can safely be used as a correlator between the subsystems. Thus the monitor needs to track multiple possibilities, and may never, in fact, be able to resolve the situation completely if there is no single, complete chain of correlations.

Other factors, such as event timing, can be used to determine the most likely interpretation. For example, the monitor might be able to be pro-active, consulting a known tracking service to resolve ambiguity; however, this kind of solution is likely to be application specific, and so difficult to generalize.

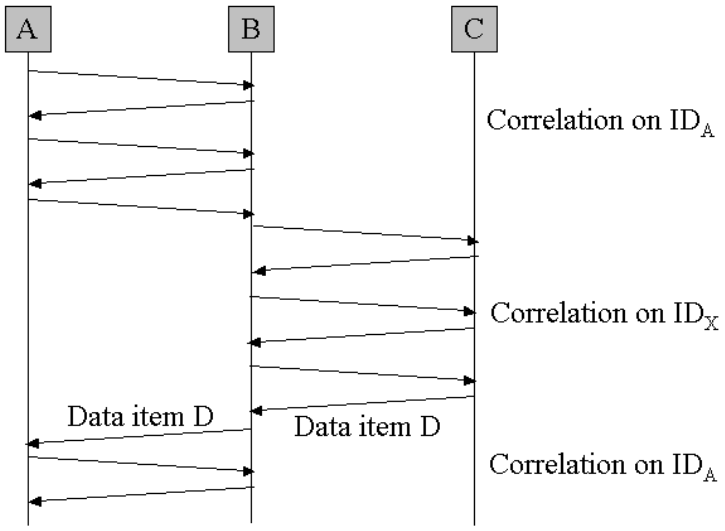

Fig. 4. Combining different correlation domains in a process.

4.3. *Continuous Quantitative Constraints*

Some contracts, such as Service Level Agreements, express a mixture of discrete behaviour, in terms of actions, and continuous quantitative measures and activities, and this illustrates another way in which processing decisions may need to be devolved to the monitor. Although such detail is largely concerned with the implementation of the monitor, it is important that the monitor metamodel includes the necessary control information to allow there to be an efficient implementation. Requirements for quantitative constraints of this kind might be found in the energy supply industry or in telecommunications, but it may be easier to think in terms of a more physical case. Consider, for example, a supplier of a raw material, such as orange juice. This is shipped as concentrate by road tankers to a packaging plant, and the telemetry on the receiving dock reports the flow of juice into holding tanks within the plant. The supply contract requires a lower bound to be placed on the rate of supply of juice, averaged over a three-day period. It does not commit to any particular size or number of tankers.

The average could be calculated whenever a telemetry message was received from the dock, but this could place a significant burden on the monitor and the notification infrastructure. Considerable savings could be made if the monitor were to operate in a pull-mode, in which the flow was integrated locally and the monitor then queried the packing plant system about the result of deliveries over a suitably chosen recent period. The problem is then how to choose this period.

If, at some point in time, the monitor updates its historical records of juice flow, it can calculate whether there has been a contract violation in the last three days,

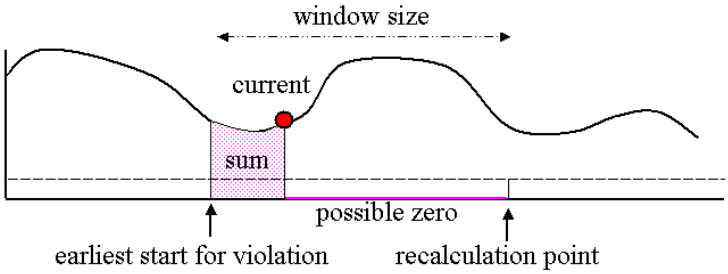

Fig. 5. Selecting the next review point.

or not. It can also make a worst case assumption (see figure 5) that the juice flow might have stopped just after this report and then remained at zero. The monitor is then in a position to calculate the earliest time at which the contract could, in these circumstances, have been violated. What it does is to calculate the total volume needed to achieve the required average rate throughout the assessment time window, and then to integrate the observed deliveries backward from the current time to determine the time in the past since which that amount has been delivered. By shifting forward from that point by the time-window size, it can determine the earliest possible time at which a violation might occur. The earliest possible violation time can then be used as a deadline for the next reassessment of the situation. If further deliveries have, in fact, been made, the process can be repeated and a further re-evaluation scheduled in the same way. If the supply has indeed dried up, a violation can be signaled.

Note that this technique can only be applied to a required maximum usage rate if there is a known upper bound on the flow rate, for obvious reasons. However, in telecommunications, such a bound will often exist (on bandwidth use, for example).

Although the precise details of this example are not likely to be found in many different contracts, the need to assess rolling averages subject to domain specific constraints, and to apply constraints on rolling averages, is likely to be found quite often in a variety of supply contracts and service level agreements. A monitor meta-model should therefore be able to support declarations of evaluation functions and provide a framework for this kind of efficient assessment of continuous conditions.

The requirements can be quite complex. One example found in the SLA for the UK academic network[20] is for assessment of network availability. This network supports some hundreds of institutions, and availability is a major concern. A metric is needed that can be met with current technology. This means that the targets need to be averaged over a long enough period and number of sites to prevent many spurious violations, but with stronger guarantees than a simple average over all institutions could give. The solution is to place a series of constraints on the distribution of percentage monthly availability over all sites – for example that more

than 96.5% of sites should see at least 99% availability. The target for acceptable performance is, in this case, defined by four such reference constraints at different points along the availability distribution. Calculating conformance to this kind of SLA requires flexible constraint expressions for rolling averages and rankings in the contract language.

5. The Business Contract Language Framework

Previous work by the author in collaboration with Milosevic's team at DSTC has proposed the main features of a contract monitoring language[12]. This language supports the structuring of contract definitions by using the Open Distributed Processing (ODP) Enterprise Language concept of communities[8]. A community in ODP is a configuration of collaborating objects, representing entities, that is formed to meet some goal, and so the parallel between relations of entities in a community and the structure of participants in a contract is quite clear. In the ODP work the modeling is generally assumed to be object based, and so the contracts are expressed as collaborations of objects. Some commentators have questioned the applicability of object-based methods when modeling more abstract, socially based situations, but this is not a serious limitation when considering business contracts, because the parties must be reified at least to the degree necessary to assign obligations and responsibilities to them, so the same style is practically always appropriate. Indeed, stating that something is a party can be taken to imply that it can be modeled as an object.

The idea, then, is to identify nested or overlapping communities as corresponding to contracts, subcontracts or more broadly applicable bodies of regulations. In future, for example, a business contract might be defined in the context provided by a formal re-statement of the local tax laws. Community definitions are expressed by declaring a collection of roles and stating the behaviour that these roles are involved in. The roles are the formal parameters of the community, and we can think of the community type as a template that is instantiated by filling the roles with suitable objects. There is then a correspondence between these objects in the representative model and the parties to the contract.

The behaviour of the contract, seen as a community, will generally consist of some straightforward basic behaviour, representing the expected course of normal execution of the contract, and a set of supporting clauses detailing responses to various exceptions and violations The general shape of the behaviour description is similar to many existing process algebra-based notations, with the ability to express sequence, concurrency as interleaving and guarded choice, with the outcome when more than one action is possible being determined either by the object or by its environment, as appropriate.

This basic behaviour is then qualified by a number of policies, generally expressing violation conditions, which either enable some required corrective behaviour or report violations to some higher level at which failure of the contract can be handled,

either manually or by some over-arching procedure.

The BCL language[12] also supports a flexible sliding window construct to express rolling or periodic constraints. From the point of view of the behavioural specification, this is essentially a special kind of iterator with support that allows the iteration process to be driven by temporal constraints and it supports quite general guards, which can be a mix of temporal guards and conditions over historical behaviour within the window defined. It is, therefore, a generalization of existing flow-based control structures and so integrates quite smoothly with the style of the rest of the behaviour specification.

6. The Notification Metamodel

The aim of the notification metamodel is to provide a target for the generation of calls to the infrastructure linking the contract participants to the monitor. It is concerned with the transport of reports and with those pieces of information common to all reports, such as source, destination or timing information. The content associated with specific report types is more specialized, and is derived from the monitoring metamodel (see below).

The notification metamodel is quite straightforward, and similar in style to any of the commonly used publish and subscribe messaging services (the JMS model[9] might be taken as typical). The main additional requirement is for a more detailed timing and quality of service model than would perhaps be the norm.

Detailed timing information is needed so that there is enough information for the monitor to reconstruct the sequence of events from different sources. To do this, it needs to be able to correlate source time stamps in the presence of variable transmission delays and lack of synchronization of the various local clocks involved (note, for example, that the JMS model does not name the source clock domain, making clock synchronization and compensation by the receiver impossible). This attention to the correctness of clocks is a particular requirement for contract monitoring because manipulation of clocks or introduction of artificial transmission delays can form part of fraud by, or malicious attack on, the parties involved. Considerations of this kind of threat have in the past, for example, led to the banks agreeing to use an independent time signal from a trusted third party, such as their network provider, to mark the end of the day for clearing purposes.

Even with detailed information about timing, there will still potentially be ambiguity about the actual sequence, and the monitor will need to take this into account, allowing for some margin of measurement error before flagging any violation, and considering the possibility of local reorderings. In order for it to do this, there is also a need to know the quality of service properties of the notification mechanism, so that allowance can be made for its properties in interpreting the data received.

Finally, the model needs to create the framework for classifying events and naming recipients. A contract could express reporting requirements in terms of reporting to a single named monitoring entity, but there may well, in practice, often

be multiple such entities covering different requirements. If, for example, the parties are mutually suspicious and do not accept any single monitor as being sufficiently trustworthy, there may be multiple monitors, each protecting the interests of one of the contractual parties. If there is subcontracting, a single event may form part of the checking of both the main contract and the subcontract, so the report may need to be delivered to the monitors for each of them, so that they can perform different interpretations. To meet these requirements, it may be appropriate to use a publish and subscribe style with named channels used to classify the report types.

The notification metamodel therefore consists of:

(i) a message addressing and routing part, describing source and destination identity, message categorization and associated metadata; there will generally be a need to link this with a broader-based security model;
(ii) a message specific model dealing with the identity and description of the event being reported and with the timing and quality of service considerations mentioned above. It is important here for the identity and type information to cover both the identity of the contract applied and identity of the event within the contract, since there will, in general, be a need to track a number of nested or overlapping contracts at any particular time. The model should also describe instance identification data that can be used for message correlation, although not all of the message transport mechanisms will provide this information, leading to the need for recourse to the kind of content-based correlation discussed above.

7. Monitoring Metamodel

The aim of the monitoring metamodel is to provide a target for the transformation of the contract specification into a form that is suitable for a reusable component to interpret while matching the stream of action reports generated by the parties to the contract. Thus the monitor takes a statement of the contractual constraints which is an instance of this metamodel, and uses it as steering information, interpreting it as instructions for matching the observed event stream.

Since, in general, the events received may be a local representation of the contractually meaningful events, there will commonly be more than one recognition step, even if the same technology can be applied to each of them. The first is a transformation that sifts through the raw event stream (the *Correlator* in figure 3) to recognize the contractual events, and this is then followed by the contract recognizer proper, which matches the abstract contractual events to the acceptable patterns of contractual behaviour.

The target patterns for these different steps will typically originate from different sources. The main pattern comes from the contract, but the pre-filter typically comes from subsidiary interworking agreements between the pairs of parties that need to interact directly. In simple cases this might reduce to the identification of a transport mechanism and a concrete schema for the information conveyed.

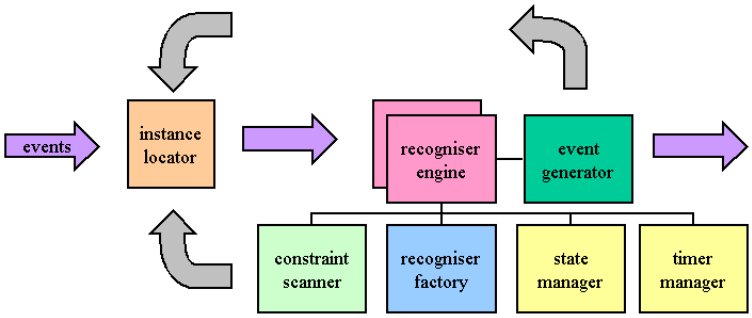

Fig. 6. Elements of the monitoring virtual machine implementation.

After the contractual events have been identified, the main items in the metamodel are structured to follow the main components in the monitor implementation, an overview of which is shown in figure 6.

The core of the monitoring virtual machine will be a set of event pattern recognizers, similar to the structure described by Neal[1618], and so the core of the metamodel will be the grammar for the event pattern language it recognizes. There will be multiple instances of this recognizer, linked to reflect both the structuring into subcontracts and the tracking of concurrent instances of contract execution.

These recognizers will be fed from an event-distributing component, the instance locator, which takes the incoming contractual events and finds the corresponding recognizer on the basis of the available correlation information associated with the events, and then passes each contractual event on to the correct recognizer.

If an event results in the contract reaching a significant state, such as the recognition of a goal, for example completion of a transaction, or an intermediate sub-goal, for example completion of a delivery, this will need to be signaled to the outside world. This may be as a report or event notification, by either an infrastructure signal or as a human-oriented message via, for example, e-mail or text messaging, or even the generation of solicitor's letters. Such outgoing event production is delegated to a separate event generator. Another option is to re-inject the events into the input stream of either this monitor or of others, so as to couple the interpretation of different contract instances.

Finally, figure 6 includes a number of supporting components that provide services to the recognizers, and the metamodel will contain corresponding structures to invoke them. Those illustrated are:

(i) *a recognizer factory*: if the instance locator discovers a new instance, or if behaviour within an existing contract requires creation of a new contract or subcontract, this factory produces a new recognizer instance to track it. The instance locator must be updated as a result so that it can direct relevant events to the new instance in future;

(ii) *a state manager*: many contractual steps need to change the context in which an instance operates, and the state manager centralizes these changes, preserving consistency between instances where necessary; it can create new instances of some or all of the state, enabling recursion to be used where it is necessary;
(iii) *a timer manager*: it is very common for the receiver of an event to be obliged to respond within a specific time. The timer manager maintains the necessary set of time-outs to police this behaviour, re-injecting exceptional events into the recognizer if a timer is not canceled before it expires;
(iv) *a continuous constraint scanner*: where continuous constraints on timeliness, or other service level properties, are required, the recognizer instantiates separate activities to maintain an ongoing test of the event stream for these properties; this can be optimized by using the techniques outlined in 4.3.

7.1. *Event Patterns*

The event pattern part of the specification will concentrate on the construction of patterns by the composition of atomic events or smaller pieces of behaviour by using behaviour composition operators. A formalization of the BPMN specification[27] would be a good starting point for this (the block structure in BPEL[24] enforces a unique initiating action and strict nesting of fork-join pairs and so is too restrictive to meet the requirements, since there may well be a requirement to describe rendezvous or for staggered fork-join structures).

The matching engine intended here would be a recognizer of behaviour expressions in a process-algebra style, similar to CSP[5] or LOTOS[6], in which a behavior is defined as a recursive composition of behaviour fragments, ending with individual actions as primitive pieces of behaviour. The set of operators would include, as a minimum:

(i) sequential composition, in which completion of one piece of behaviour is followed by the start of another;
(ii) concurrency by interleaving, in which two pieces of behaviour can be in progress, but without constraint on which should be the next to perform an action;
(iii) guarded deterministic choice, in which the various branches of the choice are determined by the state of the contract, as established by previous actions and their parameterization;
(iv) guarded non-deterministic choice, determined eventually by the environment;
(v) asynchronous exceptions that override some default behaviour, providing, for example, for asynchronous cancellation. Exceptions of this kind represent a particular problem for monitoring because they are inherently unsafe and subject to race conditions, making timing variations in reporting problematic.

Other variants seen in languages like BPMN, such as compensating actions, need not be distinct in the primitive behaviour of the recognizer, but can be constructed.

The internal structure of the monitor is essentially a recognizer for the grammar

of a set of token sequences derived from the behaviour specification, and most practical cases can be handled by transforming the specification into a state machine. This machine signals recognition of correct behaviour on reaching its final state and it may also be useful for it to recognize key intermediate stages or progress points within the defined behaviour. However, the main function of the recognizer is to signals violations on detecting any event pattern that is inconsistent with the given behaviour. These violations may be omissions or detection of events in the wrong context. Rather than just signaling the fact that there is an exception, the behaviour definition will include clauses associating exception events with particular predictable departures from the defined pattern – behaviours to catch the exception. Examples are the penalty clauses associated with failure to meet deadlines, where behaviour continues, but costs are modified.

The basic recognizer will report omissions either by detecting a subsequent event or by time-out. It may seem inconsistent to divide the handling of time into two areas, covering simple timeouts and the more complex service target monitoring described in 4.3 respectively, but in fact they require quite distinct detection strategies and different scopes of observation, and making the distinction allows them to be handled by different implementation mechanisms.

7.2. *Event flow and Configuration*

The distinction between instance location and recognition has been made so far in terms of the linkage of just two components. However, we may need to specify more complex ways in which the recognizer inputs can be bound to message categories, possibly providing for the specification of name translations to reduce the dependence on application specific details. The actions taken on pattern matches may also generate inputs to the lower level recognizer.

In cases where the application reports events with finer granularity than the contract, the hierarchy or recognizers can be extended downwards so as to construct the abstract events referenced in the contract. Since this may need to be done dynamically based on the observation of negotiation, the monitoring virtual machine must support dynamic binding of recognizers. A dynamic approach also allows the tracking of contracts that depend on short-term sub-contracts or the use of delegation. Similar hierarchical organization can be used to position contracts in appropriate local legal of regulatory frameworks, and this style, in particular, emphasizes the need to load contract information from multiple sources and interpret the structure to achieve late binding of names and inheritance of behaviour from separately defined contracts defining local context.

The event flow structure of the recognizer is thus specified in terms of the static and dynamic wiring of a number of primitive pattern recognizers. It expresses the basic structure of the contract into phases and sub-cases, but it also connects exception events to penalty or compensating structures.

7.3. *Continuous conditions*

Finally, the condition checking part can be expressed by defining constraints on the acceptability of the results of given assessment functions. This may include the results of applying assessment functions across defined intervals within the historical record; if such assessment functions can reference the time or age of the element, they can apply any required weightings internally, so a wide range of conditions could be applied by defining a map from the trace items to a result that is accumulated by one of a small set of built-in accumulators such as minimum, maximum or average values. The tactics for steering the polling of continuous values discussed in section 4.3 are subtle and best encapsulated within the virtual machine.

The sliding window mechanism discussed earlier is needed to control the basic timing mechanisms, defining which parts of the contract's history are within the scope of particular constraints, but apart form this the constraints required are expressed in a declarative constraint language relating terms in the contract that can be estimated from the observation of discrete events and continuous quantitative properties of the services being delivered.

7.4. *Managing Ambiguity*

The virtual machine implementation of contract state should also manage the tracking and pruning of alternative interpretations arising from ambiguous event sequences. Ambiguity often arises because the patterns representing different contract events share a common prefix. The implementation described by Neal[16] showed that this can be done efficiently without excessive space costs if alternatives are represented in terms of differences from the state at the point of divergence of interpretation. The implementation maintained a concise single representation of system state for periods sufficiently far in the past for all ambiguities to have been resolved, but generated a set of records for those parts of the system state affected by divergence whenever potential ambiguity was identified by the event pattern recognizer. Thereafter, the different branches were analyzed in parallel by the recognizer, with separate state records associated with each branch wherever state differences resulted.

Whenever a branch proved inconsistent with further observations, it was pruned, and the intermediate records merged and discarded if no ambiguity remained. The alternatives were also merged if they subsequently converged so that they represented a single state of the system reached via different routes. Duplicating only those parts of the state description where there was divergence has proved to be acceptably efficient in usage of both space and processing resources, and reconciliation could be carried out incrementally without sacrificing the real-time responsiveness of the implementation.

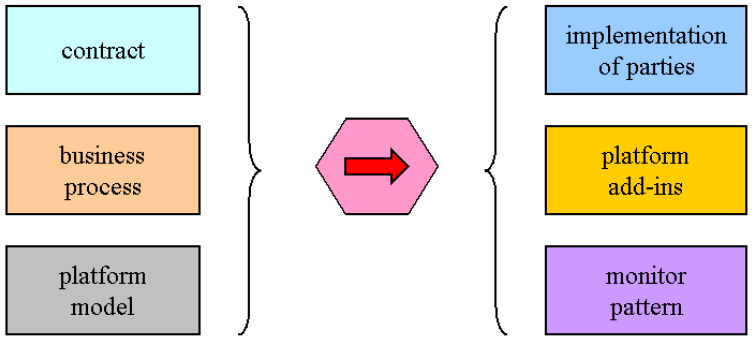

Fig. 7. The multi-way transformation applied to the support of the monitoring proccess.

7.5. *Combining the pieces*

The separation of the contract description into basic behaviour, with a monitor component that matches event patterns and re-injects abstract progress or exception events, configuration of a sequence of such matching recognizers, and constraint monitoring engines operating over a progressive moving windows on the contract's history leads in turn to a modular monitoring implementation.

Thus some quite complex matching mechanisms can be driven from straightforward contract descriptions, with the bulk of the complexity encapsulated within the reusable monitoring components, steered by descriptions produced by transformations of the contract originally negotiated between the parties, which was expressed in business terms.

However, the real test of the effectiveness of this approach is to apply it to a larger number of more complex contract examples, and increasing the level of integration will speed the process of investigating different contracts and contract styles. It is to be hoped that wider experience with a range of contractual styles will aid the selection of the basic set of common monitoring features that need to be included within the target metamodel, and will allow features of limited applicability to be discarded, leading to a tight and efficient reusable core.

8. An overview of the contract processing

This is an interesting application area for model-driven development because a more diverse set of sources and targets is involved than in traditional code generation. This is indicated in figure 7. On the input side there is a need first to ensure that the business processes of the individual parties do indeed refine and extend the original contract. These statements of behaviour then need to be combined with the platform models (which may be subsumed into the transformations to be applied) to generate partial implementations of the application support for the parties and infrastructure add-ins to provide local coupling to the notification mechanisms. This

process delivers the components necessary to convey reporting events in a way that matches the available platforms but is consistent with the notification metamodel.

At the same time, a distinct transformation applied to the same inputs is targeted at the monitoring metamodel, and results in the steering process that describes the particular contract monitoring task in terms of the re-usable matching component described above.

In summary, it is useful to review the processing flow from the negotiated contract through to the run-time monitoring. The steps are:

(i) the contract is parsed and converted to a model consistent with the input language metamodel;
(ii) the basic behaviour and policy or exception clauses are unified with this to form a complete graph of the behaviour to monitor;
(iii) a filter is applied to remove detail of only local significance, leaving just the behaviour that the monitor is expected to track; much of the structure aimed in the original contract at simplifying negotiation can be removed at this stage, although the hooks that provide tracability on violation need to be retained;
(iv) this filtered graph is marked with actions needed to create or cancel timers, to update local stored state, and to launch additional recognizer or constraint tracker instances where necessary;
(v) constraints are compiled and tracker tactics applied to give the necessary tracker behaviour;
(vi) the graph is transformed into a structure which is optimized for traversal on the basis of events received, so that actions can be derived directly from the matching of the event from the set of events allowed in the current state, and detection of violations is straightforward;
(vii) storage resources for dynamic state are optimized and allocations made, so that the monitor actions are in terms of simple accesses;
(viii) the resulting monitoring task specification is output to the repository, in form suitable for input to the reusable monitoring component.

9. Other related work

The idea of using a well-connected tool-chain to automate aspects of software development has been an objective for system development for a long time. It was one of the motivators for the viewpoint and transparency concepts in the ODP Reference Model[7], and can be traced back for many years before that. However, the concept was given additional impetus with the OMG's proposal that models should take centre-stage[23], followed by their promotion, via the MDA guide[19], of model driven architecture as their preferred solution to automation. Since then, the technology has been maturing, and Bézivin[1] gives a good example of the richness of transformations currently being considered.

Model Driven techniques have been applied to two contracts in two senses. First, contracts are often used as a metaphor, as in Eiffel, for example, and development

books, such as Frankel[3] now discuss this style. Weis discusses associated metamodeling issues[26].

Secondly, there is application to contracts in the legal sense, and a growing amount of work has been reported in this area. It dates back to Lee[10], and the model driven approach is now becoming of growing importance[2]. There has been particular interest in the application to Service Level Agreements[21 22 14], where generation of checkers for quantitative constraints is particularly attractive.

10. Conclusions

The main thrust of this paper is that before a model driven approach to the support of contracts can be successful, we need off the shelf components capable of supporting the monitoring of a large range of contracts, and that the key to reuse of such components is to define a family of metamodels for the event distribution and monitoring functions. If such models exist, they can provide the targets for transformations from the models representing the contracts to the models representing steering information guiding the monitors.

Based on the previous hand-built systems, we know what functions are required, and analysis of them to give the structure of a suitable target metamodel has been described here.

This is a general requirement, in that application of a model driven approach in other areas will also depend for their success on the creation of a supporting commodity market in components and in the corresponding target metamodels.

These automated transformations of control structures, together with the kinds of transformation from business logic to executable processes already given more prominence in model-driven code generation, should make the support of a wide range of different specific contracts tractable at reasonable total cost.

References

1. J. Bézivin, V.Devedzic, D. Djuric, J.M. Favreau, D. Gasevic, F. Jouault, An M3-Neutral Infrastructure for Bridging Model Engineering and Ontology Engineering, in Proc. Interop-ESA, Geneva, February 2005.
2. Keith Duddy, Michael Lawley and Zoran Milosevic, Elemental and Pegamento: The Final Cut - Applying the MDA Pattern, in Proc. 8th International Conference on Enterprise Distributed Object Computing (EDOC'04), Monterey, California, USA, September, 2004.
3. D. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, John Wiley, November 2002
4. Alan Hartman, Kenneth Nagin and Sergey Olvovsky, Model Driven Testing and MDA, Proceedings of Workshop on Model Driven Development (WMDD2004) in ECOOP 2004, Oslo, Norway, June 2004.
5. C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.
6. ISO/IEC IS 8807, Information processing systems – Open Systems Interconnection – LOTOS: A formal description technique based on the temporal ordering of observational behaviour, 1989.

7. ISO/IEC IS 10746, Information Technology – Open Distributed Processing - Reference Model – Parts 1-4, 1996.
8. ISO/IEC IS 15414, Open Distributed Processing – Enterprise Language, 2002.
9. Java Message Service 1.1, Sun Microsystems, April 2002.
10. R. Lee, A Logic Model for Electronic Contracting, Decision Support Systems, volume 4, 1988
11. P. F. Linington, A policy-based model-driven security framework, Middleware2003 Companion, Workshop Proceedings, 1st International Workshop on Model-Driven Approaches to Middleware Applications Development, Rio de Janerio, June 2003.
12. P. F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni and S. Neal, A unified behavioural model and a contract for extended enterprise, Data Knowledge and Engineering Journal, 51, 5–29, October 2004.
13. P. F. Linington and S. Neal, Using Policies in the Checking of Business-to-Business Contracts, Policy 2003 Workshop.
14. Molina-Jimenez, C., Shrivastava, S., Solaiman, E. and Warne, J. Run-time monitoring and enforcement of electronic contracts, Electronic Commerce Research and Applications, Volume 3, Issue 2, pp 108-125 Elsevier B.V., 2004
15. Z. Milosevic. Enterprise Aspects of Open Distributed Systems. PhD thesis, Computer Science Dept. The University of Queensland, October 1995.
16. S. Neal, A Language for the Dynamic Verification of Design Patterns in Distributed Computing, PhD Thesis, University of Kent, 2001.
17. S. Neal, J. Cole, P. F. Linington, Z. Milosevic, S. Gibson and S. Kulkarni, Identifying requirements for Business Contract Language: a Monitoring Perspective, in Proc. 7th International Enterprise Distributed Object Computing Conference, Brisbane, Australia, September 2003.
18. S. Neal and P. F. Linington., Tool Support for Development using Patterns, in Proc. 5th International Enterprise Distributed Object Computing Conference, Seattle, USA, September 2001.
19. OMG MDA Guide Version 1.0.1, omg/2003-06-01 ed., The Object Management Group (OMG), June 2003.
20. Service Level Agreement for the Operational Production Services Provided by UKERNA, http://www.mu.jisc.ac.uk/slas/ukerna/2004-05/ukernasla2004-05.pdf, July 2004
21. J. Skene and W. Emmerich, Generating a Contract Checker for an SLA Language, in Proc. of the EDOC 2004 Workshop on Contract Architectures and Languages, Monterey, California. IEEE Computer Society Press, September 2004.
22. J. Skene, D. Lamanna and W. Emmerich, Precise Service Level Agreements, in Proc. of the 26th Int. Conference on Software Engineering, Edinburgh, UK. pp. 179-188. IEEE Computer Society Press, 2004.
23. R. Soley et al., MDA, Model Driven Architecture, November 2000.
24. S. Tatte et al., Business Process Execution Language for Web Service Version 1.1, BEA Systems, IBM and Microsoft, May 2003.
25. Tom Verdickt, Bart Dhoedt, Frank Gielen and Piet Demeester, Incorporating SPE into MDA: Including Middleware Performance Details into System Models, Fourth International Workshop on Software and Performance (WOSP 2004), Redwood City, California, USA, January 2004
26. T. Weis et al., A UML Meta-model for Contract Aware Components, in Proc. 4th International Conference on Unified Modeling Language (UML 2000), Lecture Notes in Computer Science volume 2185, Springer-Verlag, 2001.
27. S. A. White et al., Business Process Modeling Notation, BPMI.org, August 2003.