ELSEVIER

The Journal of Systems and Software 41 (1998) 105-115

The Journal of Systems and Software

Promoting business policies in object-oriented methods [1]

Oscar Diaz [a,*], Jon Iturrioz [a], Mario G. Piattini [b,2]

[a] *Departamento de Lenguajes y Sistemas Informáticos, Universidad del País Vasco/Euskal Herriko Unibertsitatea, Apd. 649, 20080 San Sebastián, Spain*

[b] *CRONOS IBERICA, S.A. Clara del Rey, 8, 1-4128002 Madrid, Spain*

Received 25 July 1996; received in revised form 28 October 1996; accepted 17 March 1997

Abstract

Business policies have been proposed to bridge the gap between business and information system professionals, and at the same time, for easing system evolution. So far, however, most approaches to business policies have been biased towards providing a structural perspective. Here, we argue that there is much to be gained from moving the business-policy idea to a behavioral setting such as the one used in most object-oriented methods. This paper proposes a division of behavioral domain features into two orthogonal dimensions *depending on the stability* of these features: *the event dimension* which mainly corresponds to state-transition diagrams that are rarely changed, and *the policy dimension* which describes restrictions and dependencies among elements on the event dimension that routinely evolve with time. This explicit and separate description of business policies allows the changing of these policies without impacting unnecessarily on the underlying domain, thus easing requirement modifications and finally, enhancing software evolution. The paper addresses policy identification, description and implementation. © 1998 Elsevier Science Inc. All rights reserved.

Keywords: Object oriented analysis; Business-rules

1. Introduction

For some time now, it has been possible to identify two tendencies in system analysis: the business-centered and the context-promoting tendencies. The former attempts to overcome weaknesses in traditional analysis techniques whereby data processing formalisms are introduced early in the analysis process, thus deterring business analysts and managers from using these formalisms directly (Goti, 1994). Business-oriented models aim at providing a means understandable by customers to enhance their engagement and participation in order to facilitate assessment of the accurateness of the analysis, and in doing so, bridge the gap between business and information system professionals.

At the same time, the software development community has realized the need to broaden its view of analysis to include not only application requirements as such but also the context within which the application will function (Siddiqi and Shekaran, 1996). Such a perspective is shared by object-oriented analysis methods such as OMT (Rumbaugh et al., 1991), where the complete analysis model includes both the domain and the application model.

In the convergence of these two tendencies can be placed the business-policy paradigm (Moriarty, 1993; Henderson-Seller et al., 1995; Odell, 1995) which aims at capturing the *domain* in terms of declarative, atomic, *business-meaningful* policies. Policies are explicit statements of constraints placed on the business concerning both structural features (i.e. asserting the description of *an essential concept, relationship or policy for the business* (Ross, 1996)) and behavioral characteristics (i.e. describing the procedures which govern/regulate how the business operates).

So far, the business-policy approach is biased towards a structural perspective as these ideas have been fueled by research among data base practitioners. In Halle (1996), policies are classified in accordance with their intent as *definitions* if a business term is defined,

* Corresponding author. E-mail: jipdigao@si.ehu.es.

[1] This work has been carried out in the context of the ACTIVA project between the University of the Basque Country and CRONOS IBERICA S.A., with the support of the Dirección General de Tecnología y Seguridad Industrial del MINER (Ministerio de Industria y Energía), and the Fondo Nacional para el Desarrollo de la Investigación Científica y Técnica (CICYT), conducted by the CDTI (Centro para el Desarrollo Tecnológico e Industrial).

[2] E-mail: mpiattini@inf-cr.uclm.es.

0164-1212/98/$19.00 © 1998 Elsevier Science Inc. All rights reserved.
PII: S0164-1212(97)10011-5

facts if business-meaningful connections are described among business terms. *constraints* which restrict those connections. and *derivations* which enable new knowledge or action to be inferred. The pioneering work of Ronald G. Ross provides a fine-grained classification of policies as well as a notation for representing terms, facts and policies for constraining/deriving business information (Ross. 1994). Structural rather than behavioral features are the main concern. By contrast, this paper focuses on behavioral business policies, i.e. those procedures under which a system operates.

In today's competitive world, companies should adapt to changing conditions. and rapid evolution is a key success factor. From a business operational view point, this implies frequent update of business policies as they represent chunks of expertise likely to be changed to accomodate policy evolution. As an example, consider a company policy whereby an item is subject to a five percent cost reduction if no orders for this product are received in a five-month period. The policy indicates the context which allows for the price to be reduced. Such a policy can evolve in both its scope (e.g. instead of a number of months, the context for the price reduction is described as a minimum number of items to be sold within a period) or its response (e.g. a bonus for future purchases rather than a discount is obtained).

Despite their importance and evolutionary character, previous methods do not emphasize the explicit capture of these policies. and thus policies are scattered among processes or objects so that maintainability, traceability and easy evolution are jeopardized. However, one of the main aims of object-orientation is seamless system evolution – indeed. encapsulation and specialization can be seen as techniques to achieve this goal – but practice shows that this aim is still far from being achieved. Promoting explicit capture of business policies in object-oriented methods will certainly help in adapting systems to a changing world. Business policies are chunks of expertise, i.e. self-contained, isolated units that can be enlarged, removed or updated more easily. Therefore, policy evolution is easier to accomplish if policies are declaratively and separately supported rather than being embedded in conventional programs (Tsalgatidou and Loucopoulos. 1991).

This paper proposes a division of the behavioral features exhibited by the domain into two orthogonal dimensions *depending on the stability* of these characteristics: *the event dimension* which roughly corresponds to OMT's dynamic model, and *the policy dimension* which describes restrictions and dependencies among elements on the event dimension. This explicit and separate description of business policies, largely neglected in traditional methods, allows the changing of business policies without impacting unnecessarily on the underlying model, thus easing requirements modification, and finally, enhancing software evolution.

The paper addresses policy identification, policy description and policy implementation. The first section outlines the elements commonly found in state-transition diagrams which form the event dimension. and motivates and illustrates the appearance of an orthogonal plane to support the separate definition of policies. The identification of policies during analysis requirement is addressed in Section 3. Section 4 focuses on policy definition whereas policy implementation is addressed in Section 5. Related work is presented in Section 6. Finally, some conclusions are drawn and future directions are given.

2. The event plane vs. the policy plane

2.1. The event plane

State-transition diagrams (STD) are the final result of the behavioral analysis phase based on the careful examination of the requirements imposed on the system. Using a user-centered analysis process, the first step is to identify the different ways of using the system, which are known as *use-cases* (Jacobson. 1995). The informal, English description of use-cases can be formalized by using *interaction diagrams* from which the main objects. their states and events are identified. The concept of *event* is particularly meaningful in this context.

STDs represent the allowable sequence of events for each entity. We follow here the approach presented in Cook and Daniels (1994) whereby an event is an instantaneous, "meaningful and atomic system change which can affect multiple objects, each object describing the effect of the event on its state" (Cook and Daniels, 1994). As an example, consider a library system: the event *request* can lead to a change in the state of both the object *book* and the object *member*. Fig. 1 illustrates the different elements involved in a STD definition, namely: parameters, which convey information about the current event occurrence (e.g. the book and the member identifiers) [3]; pre-conditions, which state realistic settings for the event to happen (e.g. the book being in catalogue); guard conditions, which allow for the same event to have different transitions leaving the same state [4] (e.g. the number of books held by the member determines the final state to be either *without_books* or *with_books*); actions, which describe the consequence of the realization of the event (e.g. a state change or some other action).

[3] By default. the event parameters always include those objects in whose STDs this event is being defined.

[4] Whereas pre-conditions describe the context required for an event to occur, guards reflect internal details about the impact of the event.

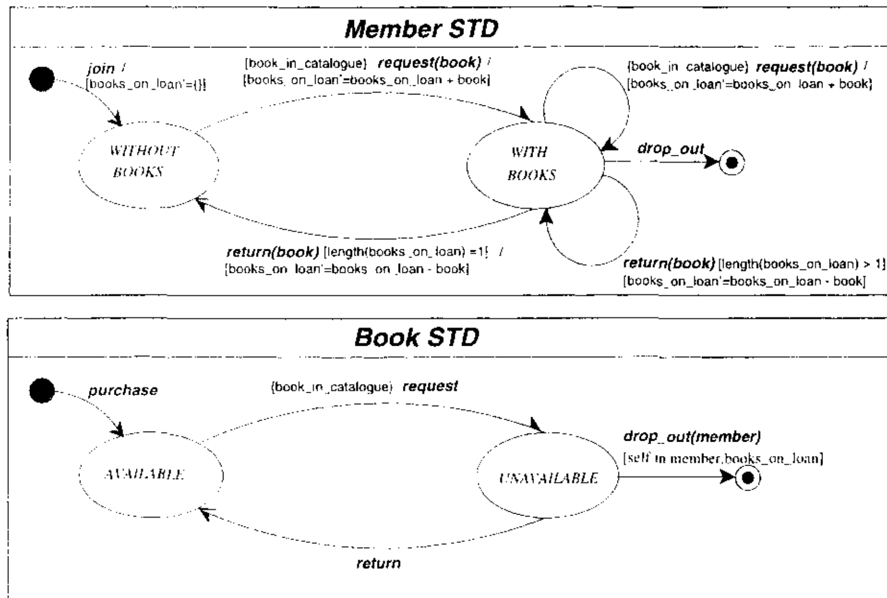

Fig. 1. State-transition diagrams for *book* and *member*.

2.2. The policy plane

Use-cases focus on the usability of the system, and, at most, they can help in obtaining generic application requirements in terms of the entities – state and behavior – involved. By contrast, policies do not describe applications but the context in which applications are run. That difference is not commonly stressed and as a result, policies end up being embedded and distributed among STDs through preconditions, guards and actions. Since STDs intertwine event description and policy enforcement, it is a cumbersome task to dig out STDs to update, validate, test or explain a policy. The policy is now no longer an isolated unit but it is now buried into STDs, and its updating requires unraveling such diagrams.

As shown in Fig. 2, the policy plane aims at extracting those policies from the event plane, and representing them explicitly. The policy plane takes the relevant elements from the event plane, and describes a relationship among them: the business policy. The example shows the STDs for objects *O* (the left one) and *O'* (the right one) which are described in the event plane using STD notation. On the other hand and using a notation described in Section 4, the policy plane gives an example of a cause-and-effect link where the event *t2* is induced when the successful ending of *t2'* is eventually followed by the successful ending of *t1*, provided that the occurrence of object *O* is in the state *S1*. Being separated in a different dimension, such a cause-and-effect relationship could suffer changes to accomodate evolution of business criteria without impacting unnecessarily on the underlying event plane.

However, not all cause-and-effect relationships can be considered a policy. Some of these relationships support event semantics by describing restrictions on the life cycle of the object. For instance, from the example shown in Fig. 1, it can be inferred that the events *return* and *request* are interleaved during the lifetime of the object, i.e. a book cannot be returned if it has not been previously borrowed, and vice versa. More sophisticated approaches such as the one described in (Jungclaus et al., 1996), allow the statement of completeness requirements for life cycles, i.e. requirements to be fulfilled before the object is allowed to die. As an example, consider that a *purchase_book* eventually implies a *borrow_book*; if such a requirement accurately reflects the library model, it implies that a book *must* compulsorily be borrowed at least once during its life time. Such restrictions are part of the definition of the object itself (i.e. its life cycle), and thus

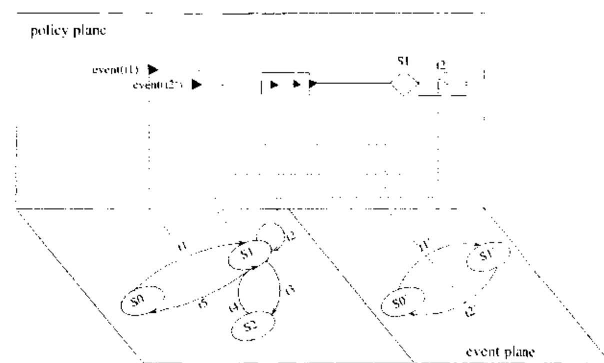

Fig. 2. Behavioral features are arranged according to their stability: the event plane (stable) and the policy plane (variable).

it is unlikely that they will be changed, i.e. they are part of the stable part of the domain.

The key notion is *stability*. The likelihood of change can be used to ascertain whether some procedure is part of the event's nature or an independent business policy. As an example, consider a newspaper company where journals are sent at the beginning of the month to customers. This effect should only occur for those customers whose subscription is on. This context can be modeled as the beginning-of-the-month instant occurring during the subscription (i.e. the interval limited by (*take_out_subscription, cancel_subscription*)). Such a situation is quite stable and is unlikely to be changed even if the company evolves. So, it should be supported by the event plane as a guard or an action of the *send-journal* event. This could require the recording of the occurrence of *take_out_subscription* and *cancel_subscription* by means of a Boolean attribute (e.g. *subscription_status*) which is the one checked by the guard.

Consider now that a promotion campaign is initiated during the tenth anniversary of the company whereby a free copy of the journal is also sent to each potential customer. Similar to the previous example, this situation can be modeled as the beginning-of-the-month instant occurring during the anniversary period, i.e. within the period limited by the events (*anniversary_started, anniversary_ended*). Unlike the previous case, the promotion policy is quite likely to change in the future, and thus, it should be moved out from the event plane and situated in the policy plane.

As this example points out, the best option is a matter of stability: if journals can be sent in a wide range of different conditions which can vary according to the different promotion policies of the company, putting this semantics in the policy plane is a better option, since its evolution can be better achieved without affecting the stable part of the domain.

3. Policy identification

The previous section has argued about the appropriateness of stability as a criteria for policy identification. Measurement of stability for software components has shown how elusive it is to assess this criteria (Castell and Slavkova, 1995). This task is even more challenging for user requirements due to the subjectivity and dependency of the domain at hand. Having said this, some guidelines can be offered to the policy designer based on four criteria, namely, the policy source, the policy arbitrariness, the policy impact and the policy complexity.

The policy's source indicates the rationales that justify the policy. Using the same classification found in integrity constraints (Wieringa et al., 1989), the source can be *analytical, deontic* or *empirical*. The former states physical or economical laws and as such, they are immutable. A policy is deontic if its rationale lays on some authorized agent (e.g., an institution, a government body and the like). Deontic policies can change as international regulations, trading agreements or markets evolve to face new situations. Stability for deontic policies can be assessed by the heuristic that the broader the scope of the authority is, the more stable are its policies. For instance, regulations from the central government tend to be more stable than those emanating from local authorities which in turn, are less likely to suffer changes than those coming from the business headquarters. The more people are affected, the more resistant is the policy to be updated. Finally, a policy is empirical if it expresses policies of thumb that nobody backs but experience demonstrates that the business behaves according to them (e.g., the purchase of those books that are on high demand).

The policy arbitrariness attempts to measure the strength of the rationales in which the policy is grounded by looking at the description of the situation where the policy is applied. The use of attribute values or constants to describe a policy situation could indicate some degree of arbitrariness which can lead to future changes. For instance, a situation where it is checked whether the member has some book seems less arbitrary than that where it is required that the member keeps three books (why not four or five?). A more stringent situation could be the requisite for the member to hold the book with title *el-Quixote*.

The policy impact criterion is based on the contention that the higher is the impact, the less likely is for the policy to suffer changes. That is, it is less dramatic to change a policy which regulates when a notification is sent than modify a policy which determines salary increase, and this, in turn has less impact than the policy regulating employee dismissal. Impact is measured by looking at the policy effect.

The fourth criterion looks at the complexity of the policy: the larger the number of properties referred within its situation is, the more likely is that a change on any of those properties will make the policy outdated.

These criteria are just heuristic guidelines for the designer to ascertain policy stability based on the domain semantics. The designer should also foresee future system requirements that can make current policies obsolete.

4. Policy description

Policies state cause-and-effect dependencies. However, the dependency's cause is not always so briefly described as an event but it can be a *composition* of different *events* whose relevance stem from occurring under *certain conditions*. These topics are addressed in the next subsections.

4.1. Events and their composition

Besides those events produced by user interactions, clock events describe the arrival of a scheduled point in time (e.g. 31 December 1995). The occurrence of one of these events in isolation is quite often not enough to describe the cause which makes an effect to follow, but usually involves different events. As an example, consider the policy whereby a book has to be returned thirty days after it has been borrowed, otherwise the keeper becomes a defaulter. Here, the cause includes the events book *request* and book *return* as well as a clock event.

Event composition has largely been studied for active databases (Widom and Ceri, 1996) where *event-condition-action rules* are supported. The event part can be a combination of other events using a range of operators. Among the most common operators are as follows.

- *Disjunction*: $E_1 \vee E_2$ occurs when either E_1 or E_2 occurs. For instance, the salary increase policy can be applied when either there has been an *employee_promoted* event or a *business_productivity_increased* event;
- *Conjunction*: $E_1 \wedge E_2$ happens when both E_1 and E_2 have occurred in any order;
- *Negation*: *not* E_1 *in Int* happens where no occurrence of E_1 has happened during the interval *Int*; the interval is itself described as a pair of events. As an example, consider a car insurance company. The allowance policy can be applied to a customer when no occurrence of the *customer_had_an_accident* event has occurred in the interval described as the pair of clock events (*beginning_of_year, end_of_year*);
- *Sequence*: E_1; E_2 occurs when E_1 occurs before E_2. Here, the order is important. When the event *customer_had_an_accident* occurs, the insurance compensation policy comes into effect only if previously, the event *customer_insurance_policy_paid* has happened;
- *Closure*: **E in Int* is raised only once the first time *E* is signaled, regardless of later occurrences of this event in the time interval *Int*. For instance, the membership fee has to be paid as soon as a member borrows a book during a year, and regardless of the number of books borrowed;
- *History*: TIMES(*n*,*E*) *in Int* is signaled when event *E* occurs *n* times during the time interval *Int*. The policy to remove careless drivers in an insurance company can be described as an excess in the number of accidents, e.g. the *customer_had_an_accident* event happens ten times in a month.

Fig. 3 shows the notation used to denote each of these situations in the policy plane. As for clock events, they are represented through a clock.

A final aspect to be considered is *the consumption policy* that indicates which of the different event occurrences (of the same event type) should be used in the composite event occurrence. As an example, consider a *book-purchasing policy* whereby additional book copies are ordered when a public subsidy is obtained. These copies are only for those books which were on loan when required. Here, the policy cause can be described as a sequence event of *book-on-loan* followed by the event *library-funding-transferred*. This sequence describes the cause that leads to the purchase of books as soon as funds are transferred. Let *bol-b1* and *bol-b2* be two occurrences of the *book-on-loan* event raised for books *b1* and *b2*, respectively. If eventually funds are obtained (i.e. an occurrence *lft-5000$* of the event *library_funding_transferred* happens), how are these occurrences paired to form the sequence occurrence? Three possible alternatives are: (1) *sequence*(*bol-b1*,*lft-5000$*) or (2) *sequence*(*bol-b2*,*lft-5000$*) or (3) *sequence*({*bol-b1*,*bol-b2*},*lft-5000$*). It is worth noticing that the policy semantics is different in each case: (1) imposes that only the very first book which happens to be on loan, is bought, (2) reflects that only the last book

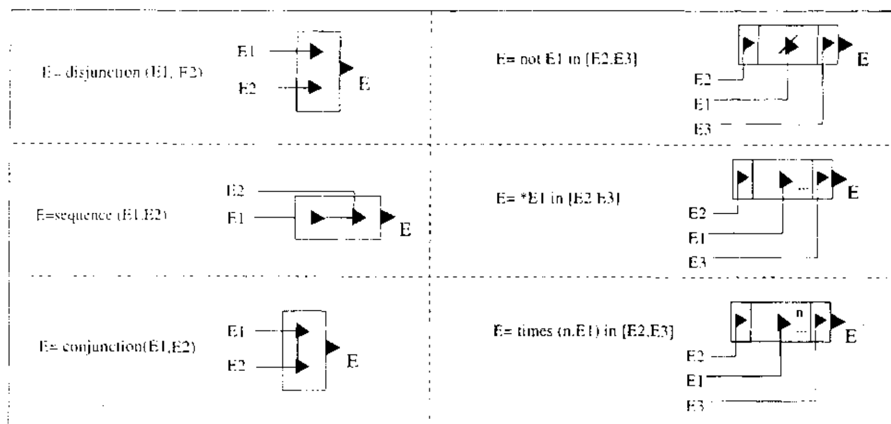

Fig. 3. Operator's notation for composition of different events.

which was on loan when requested, is bought, and (3) states that all books that were on loan, are bought. So, the consumption policy reflects part of the business policy and hence, should be represented as part of the model.

Policy consumption has been studied in Chakravarthy et al. (1994) where four options are introduced: *recent* (denoted here as $\prec$), which considers the most recent set of events that can be used to construct the composition (in the previous example, *sequence*(*bol-b2*, *lft-5000$*) is detected when *lft-5000$* arises, after which *bol-b2* and *lft-5000$* are no longer considered for the detection of *CE*); *chronicle* (denoted here as $\prec$), which consumes the events in chronological order (*sequence*(*bol-b1*, *lft-5000$*) is signaled when *lft-5000$* arises, after which *bol-b1* and *lft-5000$* are no longer considered for the detection of *CE*); *continuous* (denoted here as $\forall$), which defines a sliding window and starts a new composition with each arriving primitive event (two sequence events would begin to be constructed when *bol-b1* and *bol-b2* arise, and both sequence events are signaled as *lft-5000$* is detected); and *cumulative* (denoted here as $\uplus$), which accumulates all the primitive events until the composite event is finally raised (a sequence event is signaled only once when *lft-5000$* arises, where the first parameter of the sequence includes the parameters of all the occurrences of *bol-b1*, i.e. *bol-b1* and *bol-b2*). Unlike the continuous context, in the cumulative context an event occurrence does not participate in the construction of more than one composite event. A more comprehensive discussion of the rationale for each context can be found in Chakravarthy et al. (1994).

4.2. Events and their context

Frequently, the description of a cause includes not only the events but also the context in which these events occurred. Such contexts can be described as a combination of (1) a set of conditions on the parameters of the event, (2) a set of conditions on the current state, and (3) a tracing interval. Fig. 4 shows some examples that illustrate this situation.

(1) The event parameters include those explicitly given in the event definition as well as the occurrence time (i.e. the moment where the event was produced). For instance, *withdraw_money* could be a relevant cause if the quantity withdrawn – a parameter of the event – is above 10 000$; otherwise no effect will follow. Notice that the condition is not on the state but on the event parameters. Another example describes the insurance compensation policy where the event *customer_injured* is followed by a compensation, provided the *same customer* has previously paid the insurance policy (i.e. the event *insurance_policy_paid*). As this example points out, the composition among the occurrences to form a composite event can be restricted to certain conditions being satisfied.

(2) The cause can also include a condition on the current state such as whether the member who requests a book has already five books on loan.

(3) Finally, the tracing interval is a pair of events that delimit the period of time during which the occurrence of another event is relevant. An example is the promotion policy initiated during the anniversary of a newspaper company. This promotion consists of sending for free journals at the beginning of the month to each potential customer. This effect should only occur if the *beginning_of_the_month* clock event occurs during the promotion period, i.e. within the period limited by the events (*anniversary_started*, *anniversary_ended*).

As an example, consider the information system required for managing a library. Besides common domain objects such as books, members and the like, a set of policies can be obtained from the user. For instance, the *book-purchasing-policy* whereby additional book copies are ordered when a public subsidy is obtained. Orders are put for those books either (1) requested by a member of the academic staff, or (2) being on high demand. A book is *on high demand* if five times a year someone wanted to borrow it but it was on loan. Notice that no further copies are bought if none of the books

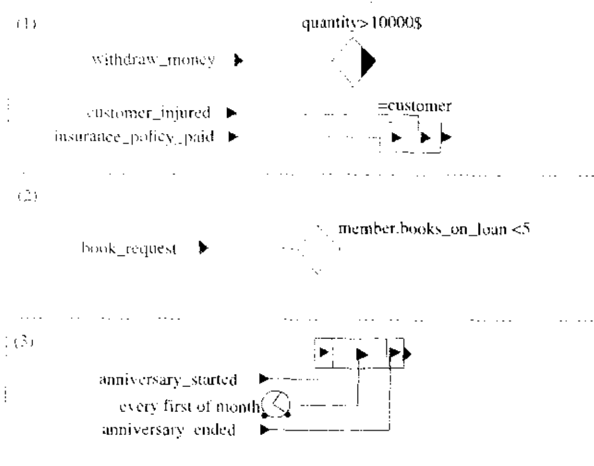

Fig. 4. Different context notation.

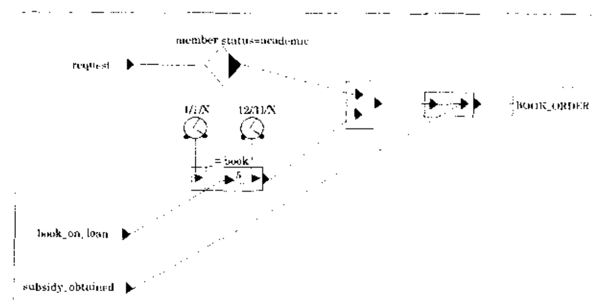

Fig. 5. Graphical description of the *book-purchasing* policy.

verified any of these two conditions. Fig. 5 shows its representation where events are denoted by the symbol ~ filled in, and causes are linked to their effect by an arrow →. The drawing should be read from left to right. As for the policy situation, the first condition (i.e. requested by a member of the academic staff) can be described as the occurrence of the event *request* provided that its parameter *member* is an *academic*. The second condition (i.e. being on high demand) can be captured as the *book_on_loan* notification event being produced five times a year. If any of these two events happen, and it is eventually followed by the *subsidy_obtained* event, the event *book_order* will be raised.

This section has proposed a notation to capture causality dependencies which support policy description. Different aspects of the cause description have been addressed whereas the effect itself is defined as a set of actions, i.e. events found in the ground plane. Again, this distinction aims at differentiating clearly at analysis time, the stable features from the most transient ones which are likely to be changed to adapt the model to the evolution of the domain.

5. Policy implementation

Section 4 has addressed policy description based on the elements found in the event plane. As we proceed from analysis to design, events are mapped into messages, and finally operations become apparent. The policy plane should be accordingly refined. Due to space limitations, we do not describe this process here but focus on how policies can be finally supported.

Two mechanisms can be available to implement business policies: methods and triggers. Methods are *call-driven*, i.e. they are explicitly invoked by other methods through their selectors, and hence, they need to know about each other. By contrast, triggers follow an event-driven pattern: they are not called but fired as a result of their event being detected, and thus, no selector is required as no explicit call exists. Triggers do not know about each other. In what follows, four approaches are considered to support business policies using these mechanisms.

5.1. First approach

The most straightforward option is to support business policies through the method that supports the associated operations. As an example, consider a *tutor-approval* policy whereby the *request* of a book by a student must be notified to his/her tutor for preventing the hoarding of books during some peak periods (e.g. during exams). Such a policy can be enforced by embedding the notification within the method which supports the *request* operation.

Regardless of whether such extra code is automatically generated by some CASE tool or manually inserted by the user, enforcing business policies through methods jeopardizes proper policy maintenance. This stems mainly from embedding functionality with different aims (e.g. operation implementation, policy enforcement) within the method body. As a result, several problems arise:

(a) *Policies are no longer explicit*. Since method code intertwines operation implementation and policy enforcement, it is a cumbersome task to dig out method code to update, validate, test or explain a policy. The policy is no longer an isolated unit but it is buried into the method code, and its update requires unraveling such code. For instance, in the *tutor-approval* example, the policy is enforced only during peak periods: otherwise, no approval is required. Enabling and disabling such policy will require changing code and recompiling the affected method. Moreover, if methods support not only operation implementation but also policy enforcement, the specialization of any of these two concepts is now tight to the same implementation construct. Careless operation overriding can lead to policy disregarding, and policy specialization forces artificial method overriding (e.g. the *tutor-approval* policy can be refined for *computing books*, a subclass of *book*, by artificially specializing the *request-book* method into the *computing book* subclass). Methods realize operations which constitute the interface or type to which an object belongs. Method specialization must then conform to subtyping norms (e.g. the contravariance rule). Being encoded within methods, policies are also affected by these rules.

(b) *Conflict resolution strategies are embedded*. Even if methods are carefully specialized, *order* of method execution may not coincide with the order in which policy has to be enforced. In general, the order of policy application proceeds from the top of the class hierarchy to the bottom. This follows from the notion that policy has to be checked in an appropriate context (i.e. more specific policy wait until more general ones have been checked, in order to endure a correct environment). However, when policies are implemented using methods, the order of evaluation depends upon where the more specialized method invokes the method that is being refined. Procedural representation of business policies requires the execution model for the policies (e.g. the order) to be coded, whereas if policies have been represented declaratively then, it is the environment that chooses the policy to be applied whenever the situation arises (Poo and Layzell, 1992). This level of indirection allows the policy execution logic to be removed from the process code, and placed independently.

(c) *Policy enforcement dispersal*. A single policy can be scattered around distinct methods, if the policy's event refers to different objects. Besides maintainability, this jeopardizes traceability, i.e. tracking how primitives

of the design level are mapped into constructs of the implementation level. It hinders the reconstruction of the policy from the code as this code is now split in different methods.

5.2. Second approach

Some of the problems can be alleviated by supporting policies as separated methods. The policy is coded into an *approval-policy* method which is invoked from within the triggering method (e.g. the request method attached to the book class). Operations and policies are supported by separate methods, easing policy maintenance (i.e. facilitating modification, deletion or addition of policies) and traceability. However, there still exists an explicit link between the operation's method and the policy's method: the operation is still aware of the policy. The problem arises from confusing the *when* and the *how*. Although business policies have to be enforced *when* operations are invoked, this does not mean that they share the same control functionality (i.e. specialization strategies, execution order, etc) as operations.

5.3. Third approach

Unlike methods, triggers are fired implicitly by the occurrence of events, such as method calls. As the binding between methods and triggers is now implicit and dynamic, this mechanism provides a suitable approach for supporting business rules. Indeed, Maring (1996) mentions that: "Business rules should be bound dynamically … ideally, you should be able to dynamically bind business-rule operators and parameters to behaviors implemented as class behaviors."

Triggers can be supported as another object feature. Besides attributes and methods, object definition encompasses the description of triggers. This approach is illustrated by the SOMA methodology (Graham, 1994) which enhances object models by adding a set of *rule sets* (i.e. an unordered set of assertions and rules of either *if/then* or *when/then* form) to each object. SOMA rules are used to make an object's semantics explicit and visible instead of being coded within methods. Poo (1993) describes a system that supports this approach.

Such perspective is akin with the object-oriented philosophy which promotes the grouping of information around the notion of object, and partially favors maintainability as it brings policy definition closer to where they are used. Moreover, reuse of triggers is easily achieved via inheritance. The bad news are policy dispersal for those policies where their events refer to several objects. In this case, one may either create a special purpose 'policy' object to which all other objects refer (known as the *enforcer*, Ross, 1996), or distribute the policy among the pertinent objects. As this is quite common, this approach has a serious drawback as far as maintainability and traceability of policies are concerned.

5.4. Fourth approach

Triggers have also been supported as first-class object. Such option is more in tune with our contention that policies should be preserved as independent concepts not polluted with the aspects that they regulate or govern. Policy isolation was promoted during analysis and design, and implementation should not be an exception.

This option resembles the architecture exhibited by expert systems where the domain knowledge is described as an unordered set of condition-action rules. This architecture was criticized due to the lack of structuring mechanisms available to cope with large rule sets: "The main problem of rule-based programming that contributes to its untestability and unmaintainability is that simplification is obtained at the cost of sacrificing some essential requirements for quality software, such as modularity (or encapsulation) and abstraction (hierarchical representation)" (Li, 1991). From this point of view, attaching rules to objects, and the derived benefit of rule inheritance, provides a first grouping criterion based on the associated object. The size of each rule set is smaller than if all the rules are taken together, and potential conflicts can be better focused.

However, some proponents claim that the *what* rather than the *who* is a more appropriate grouping criteria (Diaz and Jaime, 1997; Baralis et al., 1996). That is, rule changes, rule addition or rule conflicts are better coped with by looking at those rules with a share functionality (i.e. the purpose of the rule) rather than focusing on those rules attached to the same object. Rather than facing a large set of rules, the designer should find criteria which serve to partition this rule set into disjoint subsets of independent (criterion-wise) rules. Correctness is then checked at this higher level, and then for each rule subset. The idea is that rule grouping criteria do not necessarily coincide with the associated classes.

Trigger definition overcomes some of the pitfalls of the method-based approach, namely:

1. *Policies are explicit.* A declarative definition of policies eases both user understanding and inconsistency checking, since policies are no longer polluted by how they are enforced. It also facilitates querying, updating or relating their functionality. For instance, it will be possible to retrieve all policies that are temporally deactivated because this information can be stored explicitly as an attribute of the trigger. Moreover, methods can be overridden without jeopardizing policy enforcement.
2. *Conflict resolution strategies are kept separated.* The inheritance of constraints is no longer linked to the

inheritance of methods. It is the trigger manager that determines the order in which triggers are fired, a strategy which can be adapted to cope with different requirements.

3. *Policy enforcement gathering*. Since each trigger represents the enforcement of a single policy, removal or temporal deactivation of individual policies can easily be achieved. Explicability is also enhanced.

Despite the appeal of this approach, few systems offer full support for triggers. However, the gains to be obtained for separated trigger definition can even justify an ad hoc implementation. In Taylor (1996), where the author is confronted with a similar issue, the advice is that: "although you may have to code an inelegant work-around to the limitations of your language, you only have to do it once. From that point on, you have a business engine that can manage rules dynamically and maintain them in their natural business form, rather than burying them in method code".

Previous discussion is summed up in Table 1 where the distinct approaches are compared along five dimensions namely, availability (i.e. how common is the implementation mechanism available in commercial systems), maintainability (i.e. how easy is to update the policy base), explicability (i.e. how ease is to make the user aware of the policies being applied), traceability (i.e. how straight is the mapping from primitives of the design level into implementation constructs) and testability (i.e. how easy is to follow the control and data flow). The latter is a drawback of the dynamic binding between triggers and methods, and among triggers themselves (i.e. a trigger firing other triggers). The insidious ways in which triggers can interact make triggers difficult to test and debug (Diaz et al., 1994).

6. Related work

In the development of business information systems, has long been recognized the existence of three elements: data, process and policies. Whereas the first two have been integrated using the object-oriented paradigm, policies are commonly neglected and left implicit in the program code. The drawbacks of this approach has been identified in Appleton (1984) and Tsalgatidou and Loucopoulos (1991). In Poo and Lee (1996) and Poo (1993), an object model is proposed where policies are integrated and explicitly represented in terms of rules. This work classifies rules as constraints (e.g. preconditions, static and dynamic constraints), control rules (i.e. temporal activations of events) and derivative rules which derive conditions or new facts from some given facts. Our aim is similar to the Poo's one but here we stress the evolutionary nature of policies. In so doing, we force the analyst to consider stability as a main design criteria which we think is paramount for smooth system evolution. Besides, Poo's work is biased towards how rules can be supported. This makes the rules being object-centered where the situation mainly implies a single object. By contrast, our work attempts to face the multi-object description of policies through the use of composite events.

In the database field, the $(ER)^2$ model proposes to extend the ER model with policies (Tanaka et al., 1991). This model especially suits the visualization of the event-condition-action policies found in active databases. As the ER model encompasses the structural features of the domain, this extension mainly focuses on the relationship between the ER structural components and the policies. Hence, these relationships state the data used or the data modified by the policies, rather than the links among events (i.e. operations) which is more akin with the object-oriented paradigm and it is this paper's claim.

In the context of the Chimera active DBMSs, Baralis et al. presents a support environment for trigger generation, analysis, debugging and browsing (Baralis et al., 1996). They point out the importance of triggers for enforcing "data management policies, as they can provide a large amount of the semantics that normally needs to be coded by means of application programs; this … brings the nice consequence that data management policies can effectively evolve just by modifying rules instead of application programs". We share the same rationales but their work is focused on DBMSs rather than object-oriented methodologies.

Policy's modularity makes them specially attractive for describing processes where process' steps are represented as policies (Martin and Odell, 1995; Herbst, 1996). Martin and Odell promote the use of policy mechanism to allow for operations being isolated from cause-and-effect considerations, i.e. any operation must be completely unaware of both (1) the events which might have triggered it, and (2) the operations which might be triggered by the event raised once that operation ends.

Table 1
Comparison of distinct approaches for policy support

	First option	Second option	Third option	Fourth option
Availability	++	++	−−	−−
Maintainability	−−	−	+	++
Explicability	−−	−	+	++
Traceability	−−	+	+	++
Testability	++	++	−	−

Such isolation improves operation re-usability and eases system evolution. We agree with the postulates of Martin and Odell, and we propose an extension to a more general context where the cause is described not only by a single event but by a more complex situation.

In Herbst (1996) several case studies are presented which stress the importance of supporting policies explicitly, and presents a meta model and a repository for policy management. It is stated that a main use of policies is for integrity constraint maintenance. However, from a design view point, such integrity constraints should not be described in terms of policies but as declarative statements in the structural part of the domain. Triggers as an implementation mechanism for business policies can be quite cumbersome to define due to their flexibility and the difficulty of predicting how they will interact with one another (Diaz and Embury, 1992).

Conclusion

There is a growing interest in methods which help to develop software that is capable of evolving and adapting to changing conditions. These conditions frequently refer to the policies that govern the business. The early identification of those policies during analysis, avoiding mixing them with STDs during design, and supporting them as independently as possible during implementation, will end up building systems which can be smoothly adapted to changing conditions.

This paper has proposed *stability* as a main design criteria whereby domain features are arranged into the event plane and the policy plane according to their likelihood of being changed. Being self-contained units, policies can be enlarged, removed or updated without significant involvement of other elements. The paper presents a notation, some heuristics for assessing policy stability, and a comparison of distinct approaches for policy implementation. Currently, these ideas are being formalized through an event calculus mechanism which is being used to animate and validate the correctness of the policies.

Acknowledgements

The authors would like to thank Norman Paton and Antoni Olivé for useful comments in an early draft of this paper. Jon Iturrioz enjoys a grant from the Basque Government.

References

Appleton, D., 1984. Business rules: the missing link. Datamation 30 (16), 145–150.

Baralis, E., Ceri, S., Paraboschi, S., 1996. Modularization techniques for active rules design. ACM Trans. Inform. Systems 21 (1), 1–29.

Castell, N., Slavkova, O., 1995. Metrics for quality factors in the lesd project. In: Schafer, W., Botella, P. (Eds.), Software Engineering – ESEC'95. Springer, Berlin, pp. 423–437.

Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.-K., 1994. Composite events for active databases: semantics, contexts and detection. In: Bocca, J., Jarke, M., Zaniolo, C. (Eds.), Proceedings of the Twentieth International Conference on Very Large Data Bases. Morgan Kaufmann, Los Altos, CA, pp. 606–617.

Cook, S., Daniels, J., 1994. Designing Object Systems: Object-Oriented Modelling with Syntropy. Prentice-Hall, Englewood Cliffs, NJ.

Diaz, O., Embury, S., 1992. Generating active rules from high-level specifications. In: P.M.D. Gray, R. L. (Ed.), Advanced Database Systems – Proceedings of the Tenth British National Conference on Databases. LNCS Series. Springer, Berlin, pp. 227–243.

Diaz, O., Jaime, A., 1997. EXACT: an EXtensible approach to ACTive object-oriented databases. VLDB J. 6 (4) pp. 282–295.

Diaz, O., Jaime, A., Paton, N., 1994. DEAR: A DEbugger for Active Rules in An Object-Oriented Context. In: Paton, N., Williams, M. (Eds.), Proceedings of the First International Workshop on Rules In Database Systems. Springer, Berlin, pp. 180–193.

Goti, C., 1994. Business rules: their role in systems requirements. Technical Report TR03.587. Santa Teresa Laboratory, San Jose, IBM.

Graham, I., 1994. Object Oriented Methods. Addison-Wesley, Reading, MA.

Halle, B.V., 1996. The never-ending searching of knowledge. Database Programming Design 9 (4), 15–18.

Henderson-Seller, B., Fung, M., Yap, L.M., 1995. The role of business rules and quality in methodologies. Rev. Object Anal. Design 2 (4), 10–12.

Herbst, H., 1996. Business rules in systems analysis: a meta-model and repository system. J. Inform. Systems 21 (2), 147–166.

Jacobson, I., 1995. Formalizing use-case modeling. J. Object Oriented Programming 3 (3), 10–14.

Jungclaus, R., Saake, G., Hartmann, T., Sernadas, C., 1996. TROLL: a language for object-oriented specification of information systems. ACM Trans. Inform. Systems 14 (2), 175–211.

Li, X., 1991. Whats so bad about rule-based programming? IEEE Software 8, 103–105.

Maring, B., 1996. Object oriented development of large application. IEEE Software 13 (3), 33–40.

Martin, J., Odell, J., 1995. Object Oriented Methods: A Foundation. Prentice-Hall, Englewood Cliffs, NJ.

Moriarty, T., 1993. Business rule analysis. Database Programming Design 6 (4), 66–69.

Odell, J., 1995. Business rules. Object Magazine 5 (1), 53–56.

Poo, D., 1993. Implementing an evolutionary structural software model. J. Systems Software 22 (2), 22–81.

Poo, D., Layzell, P., 1992. An evolutionary structural model for software maintenance. J. Systems Software 18 (2), 113–123.

Poo, D., Lee, S., 1996. TarTan: interweaving objects with rules in information systems development. J. Systems Software 33 (1), 3–33.

Ross, R., 1994. The Business Rule Book. Data Base Newsletter.

Ross, R., 1996. Business rules: a better foundation for information systems. In: Freeman, M. (Ed.), The Business Rules Summit.

Rumbaugh, J., Blaha, M., Premerlani, W., 1991. Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs, NJ.

Siddiqi, J., Shekaran, C., 1996. Requirements engineering: The emerging wisdom. IEEE Software 13 (2), 15–19.

Tanaka, A., Navathe, S., Chakravarthy, S., Karlapalem, K., 1991. ER-R: an enhanced ER model with situation-action rules to capture application semantics. In: Teorey, T. (Ed.), Proceedings of the Tenth International Conference on the Entity Relationship Approach, pp. 59–75.

Taylor, D., 1996. Can intelligence be inherited? Object Magazine 6 (7), 22–24.

Tsalgatidou, A., Loucopoulos, P., 1991. An object-oriented rule-based approach to the dynamic modeling of information systems. In: Sol, H., van Hee, K. (Eds.), Dynamic Modeling of Information Systems. North-Holland, Amsterdam.

Widom, J., Ceri, S., 1996. (Eds.). Active Database Systems. Morgan Kaufmann, Los Altos, CA.

Wieringa, R., Meyer, J., Weigand, H., 1989. Specifying dynamic and deontic integrity constraints. Data Knowledge Eng. 4 (2), 157–189.

Oscar Díaz. He received a B.Sc. in Computing Science from the Basque Country University in 1985, and the Ph.D. degree in 1992 from the University of Aberdeen for a thesis on the enhancement of an object-oriented database with active capabilities. He is an Assistant Professor at the Computing Science Department of the University of the Basque Country at San Sebastian. His current research interests include object-oriented database, conceptual modeling and active database systems.

Jon Iturrioz. He received a B.Sc. in Computing Science from the Basque Country University in 1993 and he currently enjoys a pre-doctoral grant. His current research interests include object-oriented conceptual modeling and business rules.

Mario Piattini. He is the Research and Development Director of Cronos Iberica, S.A. and also an Assistant Professor in the Software Engineering Department of the Universidad Carlos III of Madrid, Spain. He received the Ph.D. degree in Computing Science from the Polytechnical University of Madrid in 1994. He is also CISA (ISACA). His main interests include databases, object-orientation, development methodologies, and auditing.