# Stimuli and Business Policies as Modelling Constructs: Their Definition and Validation Through the Event Calculus

Oscar Díaz and Norman W. Paton†

Departamento de Lenguajes y Sistemas Informáticos,
University of the Basque Country - San Sebastián, Spain.
e-mail: jipdigao@si.ehu.es
Department of Computer Science†,
University of Manchester - Manchester, U.K.
e-mail: norm@cs.man.ac.uk

**Abstract**

Accurate gathering of requirements is a major concern during conceptual modelling. Such accurateness can only be achieved through major involvement of users, who should check whether the system's specification conforms with their expectations. This task can be facilitated both by intuitive conceptual constructs and by executable models that allow interaction with the user to explain the behaviour of the system in accordance with its specification. This work proposes the notions of stimuli and business policies as intuitive behavioural constructs, and the use of the event calculus as an appropriate formalism for building executable specifications for behavioural models. The approach is borne out by an early implementation that allows the user to question why and how a given state is reached, where the answer is given in terms of the specifications, i.e. stimuli and policies, being applied.

## 1 Introduction

A key step in information system development is the accurate gathering of requirements that define the purpose of the system. This purpose must be sought outside the system itself, in its ability to create, change and maintain relationships in its domain, and should lead to the description of the external behaviour of the system: its interface with the surrounding domain. In this context, the concept of stimulus[1] is central to information system conceptual modelling. Its

[1]Stimuli have also been referred to as transactions or external events. We prefer to use stimulus, as transaction has a database flavour, and the notion of event is currently overload with different meanings.

importance stems from the fact that it represents the interaction between the user and the system itself [10].

However, in most approaches to requirements capture, the notion of stimulus is mostly diluted in favour of the notions of data or of operation due to the influence exerted by database technology. Data can be seen as the memory retained from past stimuli, whereas operations are realizations of stimuli.

Our contention is that the notion of stimulus is better suited for requirements gathering than the notions of data or operation. First, it is nearer to how users describe their environments: the interaction rather than the data is what is first obtained from the users, i.e what they do rather than the data required to do it. Second, DBMS technology is evolving to manage not only the structural features of the domain but also the behavioural ones. Object-oriented and active database management systems (DBMS) are examples of this trend. New conceptual models should account not only for the domain structure but also the domain behaviour, and in this setting, the notion of stimulus should play a core role.

The notion of stimulus is roughly supported in current relational DBMS through transactions. The important difference is that relational systems provide commands for the user to define what a transaction is (e.g. *begin-transaction, end-transaction*), but transactions themselves are embedded in application programs. The DBMS knows about data updates (i.e. how to insert, delete or update data in a table), but ignores compound behavioural units. However, it is these more abstract units that correspond to the terms in which the user describes his/her requirements. However, the tendency is for DBMSs to be enhanced to support these features as well. Procedural attachments, stored procedures, packages and the like are being added to current commercial products.

As well as the notion of stimulus, business policies are increasingly being considered as a modelling notion [12, 7]. When describing the system not only interactions (i.e. stimuli), but also the policies that govern such interactions, describe important aspects of the domain. Like stimuli, the main rationale for incorporating business policies into our conceptual model are naturalness and feasibility – mechanisms such as triggers are currently available in most commercial DBMSs that reflect the need to capture policies within information systems. 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 [14].

Once the conceptual model has been described in terms of stimuli and policies, the model has to be validated, i.e. its accurateness has to be checked against the users' intended requirements. We follow the approach described in [13] where the model is validated by providing explanations about the results of model execution. Here we apply their ideas to our model of stimuli and policies, and show how the use of the *event calculus* [8, 5] greatly simplifies this task.

Hence, the second contention of this work is that the event calculus is an adequate formalism for describing stimuli for reasons of uniformity and modifiability: uniformity because the notion of stimulus is at the core of this calculus

and is thus fully integrated with other concepts such as that of the extension of the database or the rules for deriving new data from that which is stored; and modifiability because the event calculus accounts for new stimuli or policy additions or modifications without major rewriting, as support for such concepts is handled in a localised and declarative manner. This is seen as a major advantage during requirements analysis where the conceptual model is likely to experience updates as a result of suggestions from the users.

The rest of the paper is structured as follows. Stimuli and policy definition are addressed in the first section on requirements analysis. Section 3 introduces the event calculus, and its extension to cope with complex stimuli is presented in section 4. Once stimuli and policies have been defined in the context of the event calculus, their accurateness is addressed through model execution as shown in section 5. Finally, conclusions are given.

# 2 Requirements Analysis

User-centered analysis is the process of capturing requirements from the user's point of view. *Use cases* are a popular approach to user-centered analysis where system behaviour is partitioned according to particular external goals. Each use case abstracts different ways of interacting with the system to achieve a similar aim. It is a user-level, black-box view of the system. This approach is popular among object-oriented methodologies.

Use cases for conceptual modelling could seem odd at first glance. However, they help to ground requirement gathering in concrete examples, and their intuitiveness facilitates user involvement. Besides, they can be used to ascertain the scope of the system and to provide valuable test cases.

A common approach is to describe use cases through interaction diagrams. Interactions can refer to stimulus notifications (i.e. inputs to make the system aware that some relevant stimulus occurred), reports from the system, or queries (i.e. requests for information from the system) The flow in the diagrams is top-down, where alternative paths are delimited through boxes [1]. Our intuition is that use cases are repeating patterns where after a stimulus is received from some actor, some exceptional alternatives are considered which end the use case (depicted as a box with an $X$ at the top left corner); if exceptional situations do not hold, the basic procedure to handle this stimulus follows, which usually implies the generation of some reports from the system or changes in the system state that record the happening of the stimulus; finally, some ending alternatives are considered (depicted as a box with a $?$ at the top left corner). During requirements analysis, exceptions describe abnormal endings of the use case. Interaction diagrams can then be seen as a structuring mechanism whereby the ordinary context is described first, and exceptional circumstances are postponed for later analysis.

An example is shown in figure 1 where the procedure followed to borrow a book is described. The first step is for the member to request the book from the library and then, for the library to notify acceptance of this request

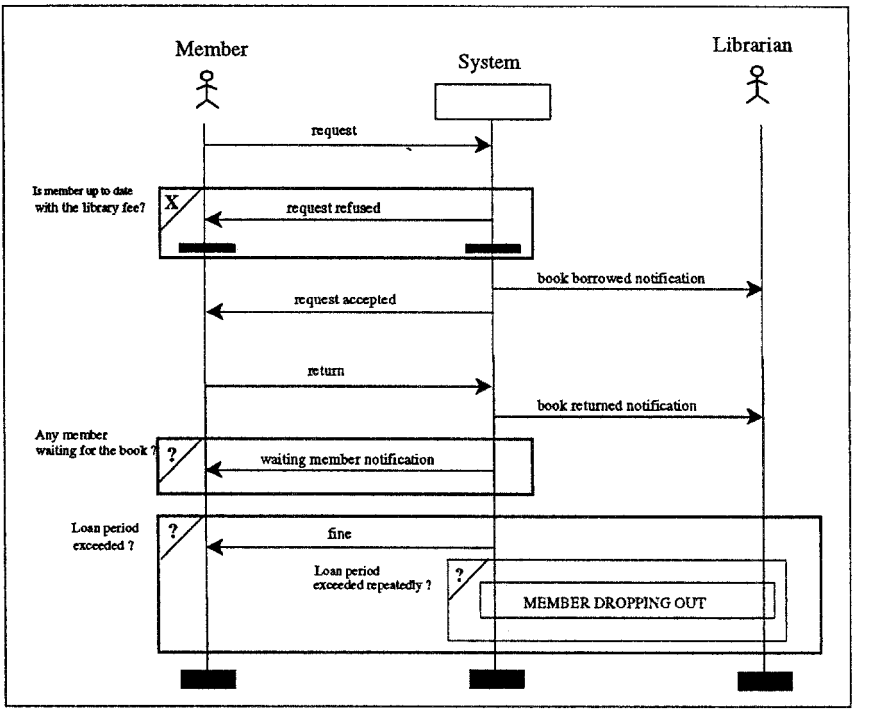


Figure 1: Interaction diagrams for *requesting* and *returning*.

to the member and to the librarian. The exceptional alternatives to this flow can be that the member is not up to date with library fees. This causes the request to be refused. Once the book is returned, a notification is sent to those members waiting for this book. In addition, if the legal borrowing period has been exceeded a fine is imposed on the borrower.

Stimuli and policies are obtained from interaction diagrams. Each repeating pattern is examined: each pattern roughly corresponds to a stimulus, whereas exceptional and ending alternatives could lead to business policies. In the previous use case two stimuli are identified, *requesting* and *returning*. As for policies, they are identified based upon the stability of the alternative i.e. its likelihood to be changed in the future: if the alternative is stable, then it is described as part of the stimulus definition; if the alternative is volatile, it is described separately as a business policy. In this way, changes in the policy can be accomplished without impacting upon the stimuli (see [4] for further details).

## 2.1 Describing Stimuli

A stimulus is described through five aspects, illustrated in figure 2 for the *requesting* stimulus:

```
stimulus definition [
  name: requesting([Person,Book]),
  preconditions: [
     property(Book,state,book_available)
     not property(Person,has_books,_) ],
  exceptional_alternatives: [ property(Person,up_to_date,no) ],
  initiates: [
     property(Person,has_books,Book) if true,
     property(Book,available,fail) if true ],
  terminates: [ property(Book,available,true) if true ],
  notification: [
   request_rejected([Person,Book]) if property(Person,up_to_date,no)),
   request_accepted([Person,Book]) if not property(Person,up_to_date,no),
   book_borrowed_notification([Person,Book]) if not property(Person,up_to_date,no)
] ].
```

Figure 2: Description of the *requesting* stimulus.

- name and parameters (e.g. *requesting([Person,Book])*),
- pre-conditions, which state realistic requirements if the stimulus is to happen [3] (e.g. the book is available and the member has no books, as a member can keep only one book),
- exceptional circumstances, that do not prevent the stimulus from happening but that avoid the standard processing of this stimulus. For instance, a member can apply for a book (i.e. the *requesting* stimulus occurs). However, if the member is not up to date with library fees, the requesting process cannot be completed. They can be seen as conditions that lead to no change in the system.
- post-conditions, which describe the resulting state in terms of what becomes valid and invalid through the *initiates* and *terminates* clauses, respectively. These clauses are lists of *property if condition* statements. As an example, the *requesting* stimulus initiates the fact that the person has the book and that the book is unavailable, whereas it ends the fact that the book is available.
- notifications, which describe notifications issued by the system in response to certain circumstances. In the example, the system notifies the member and the librarian of the request being satisfied, or of the rejection of the request if the member is not up to date with library fees.

## 2.2 Describing Policies

A policy reflects cause-and-effect dependencies. Its description includes the cause or situation where the policy applies, and the effect that such a policy conveys.

Being in a behavioural setting, such descriptions involve stimuli rather than states. As an example, consider the policy whereby if the loan period is exceeded, a fine is imposed on the borrower when the book is returned. We opt for a policy rather than a stimulus notification as this semantics can evolve in both its scope (e.g. the number of days a book can be on loan) and its response (e.g. a temporary removal of the membership rather than a fine could be the most appropriate corrective action). As this example shows, the occurrence of a stimulus in isolation is quite often not enough to describe the cause, which usually involves different stimuli. In the previous example, the cause refers to a *requesting*, a *returning* and a clock stimulus: the book not being returned, thirty days after it has been borrowed. Hence, composition operators have been introduced to reflect these situations [6] (e.g. *conjunction* – $S_1 \wedge S_2$ happens when both $S_1$ and $S_2$ have occurred in any order; *sequence* – $S_1; S_2$ occurs when $S_1$ occurs before $S_2$; *TIMES(n,S) in Int* is signaled when stimulus $S$ occurs $n$ times during the time interval *Int*).

Sometimes the policy itself is more subtly reflected. Consider that when a book is returned, a notification is sent to those borrowers whose request was refused as the book was on loan. The question is which member has to be notified: the first to have a request rejected, the most recent to make a request, or all members in this situation so that the book can be lent to the first arriving at the library. Such ambiguity should be resolved by deciding the *consumption policy* that indicates which of the different stimulus occurrences (of the same type) should be used in the policy. Four options are described in [2] in the context of active databases: **recent**, which considers the most recent set of stimuli that can be used to construct the composition (in the previous example, the last member that had a request refused is the one to be notified when the book is returned); **chronicle**, which consumes the stimuli in chronological order (in the previous example, the first member gets the notification; in turn, once this member returns the book, the second member is notified and so on); **continuous**, which defines a sliding window and starts a new composition with each arriving primitive stimulus (here all the member are notified simultaneously); and **cumulative**, which accumulates all the primitive stimulus until the composite stimulus is finally raised (this is not applicable in our example, as each notification is sent to a single member rather than to a set of members). A more comprehensive discussion of the rationale for each context can be found in [2].

The definition of a *notification* policy can be as follows:

```
policy definition [
   name: r2,
   event:
       chronicleSequence(request_rejected([Member,Book]),returning([_,Book])),
   condition: [( property(Book,kind,academic))],
   action: [( => book_arrival_notification([Book,Member]) )]].
```

with the following components:

- the event, in this case a sequence of the *request_rejected* and *returning* stimuli. The stimulus occurrences that can participate in the composition

are restricted based on equality between their parameters (e.g. the book just returned and the book whose request was rejected must be the same for the composition to be meaningful; this is expressed using unification, so that variables with the same name must be instantiated with the same value). These stimuli correspond to any of the interactions found in the use cases. As for the consumption policy, it is reflected as part of the composite operator name (e.g. *chronicleSequence*); thus, for each kind of composition four operators are available (e.g. *chronicleSequence, recentSequence, continuousSequence* and *cumulativeSequence*).

- the condition, which together with the event, reflects the context where the policy is applied by imposing a condition on both the system state and the event parameters. In this example, the policy only applies for academic books i.e., a notification is sent only if the book is academic; otherwise (e.g. for leisure books), the notification service is not available,
- the action, which states a set of stimuli (e.g. *book_arrival_notification*).

The following sections present how the stimulus and policy constructs are formalised using the event calculus.

# 3 An Introduction to the Event Calculus

The event calculus [9, 8] is a logic-based calculus of events. In essence, happenings in the domain of interest are recorded in the extensional database, and deductive rules are used to infer the state of the domain that results from the events that have taken place. The event calculus has been used in this work as it is both expressive and executable, and because it had previously been indicated how the event calculus could be used to model reactive behaviour [5].

The facts that are true as a result of the events that have taken place are deduced by `holdsAt`:

```
holdsAt(Atom,TimePoint2):-
   initiates(EventId,Atom), at(EventId,TimePoint1),
   TimePoint1 =< TimePoint2,
   not brokenPersistence(Atom,TimePoint1,TimePoint2).
```

This rule indicates that `Atom` is true at `TimePoint2` if an event `EventId` caused `Atom` to become true, `EventId` took place at a time `TimePoint1` before `TimePoint2`, and nothing has happened between between `TimePoint1` and `TimePoint2` to prevent `Atom` from continuing to be true. For example, if the event `EventId` was the borrowing of the book `Grapes of Wrath` by the borrower `Oscar`, then it could cause the atom `property('Oscar',has_books,'Grapes of Wrath')` to become true.

The truth of an atom can be changed by events, as captured by `brokenPersistence`:

```
brokenPersistence(Atom,TimePoint1,TimePoint3):-
```

```
terminates(EventId2,Atom), at(EventId2,TimePoint2),
TimePoint1 =< TimePoint2, TimePoint2 =< TimePoint3.
```

This rule indicates that `Atom` is not true at `TimePoint3` if at some point between `TimePoint1` and `TimePoint3` an event `EventId2` took place that is able to stop `Atom` from being true. For example, if the event `EventId2` was the *returning* of the book `Grapes of Wrath` by the borrower `Oscar`, then it could cause the atom `property('Oscar',has_books,'Grapes of Wrath')` to stop being true.

In the event calculus, the semantics of the domain are captured by the definitions of `initiates` and `terminates`, which describe the consequences of events (e.g. that the borrowing of a book causes the person who borrowed to book to possess the book until such time as the returning of the book means that the person no longer has the book). This, as detailed in [8], is a very general framework for modelling evolving situations, and allows questions to be asked not only about the final consequences of a series of events, but also about past states and the ways in which states changed. For more details and a more formal treatment of the event calculus, the reader should consult [9, 8, 5].

# 4 Describing Stimuli and Policies Using the Event Calculus

## 4.1 Describing Stimuli Using the Event Calculus

A stimulus, as described in section 2.1, is essentially a happening in the domain which, depending upon the circumstances, is associated with a number of consequences. The notion of stimulus therefore accords well with the notion of *event* in the event calculus. For example, in the library example, we can provide a definition of `initiates` to indicate that the atom `property(Person,has_books,Book)` is true if the event `requesting(Person,Book)` has taken place:

```
initiates(EventId,property(Person,has_books,Book)) :-
    happened(EventId,requesting(Person,Book)).
```

This provides a definition for a very straightforward stimulus, in which most of the features of stimuli from figure 2 are left undefined:

```
stimulus definition [
  name: requesting([Person,Book]),
  initiates: [
    property(Person,has_books,Book) if true
  ]].
```

However, although there is not room to describe the complete mapping here, it turns out that the other properties of stimuli from figure 2 also have natural mappings onto rules in the event calculus framework – the syntax from figure 2 simply provides a syntactic and organisational framework that eases the description of stimuli, and that can be used as a basis for further analyses, as outlined in section 5.

```
initiates(EId, recentAnd(DefId,EIdTime,LHS,RHS)) :-
    % Find candidate component events.
    recentAnd(DefId,_,LHS,RHS),
    at(EId,EIdTime),
    holdsAt(LHS,EIdTime), holdsAt(RHS,EIdTime),
    signalledAt(LHS,LHSTime), signalledAt(RHS,RHSTime),
    (LHSTime == EIdTime ; RHSTime == EIdTime),
    % Check that candidates are most recent.
    recentAnd(DefId,_,LHSTemplate,RHSTemplate),
    not ( LHSTemplate = LHS,
          holdsAt(RHSTemplate,EIdTime),
          signalledAt(RHSTemplate,RHSTemplateTime),
          RHSTemplateTime > RHSTime),
    not ( RHSTemplate = RHS,
          holdsAt(LHSTemplate,EIdTime),
          signalledAt(LHSTemplate,LHSTemplateTime),
          LHSTemplateTime > LHSTime).
```

Figure 3: Definition of `initiates` for recent and.

## 4.2 Describing Policies Using the Event Calculus

A policy, as presented in section 2.2, is essentially a mechanism for describing the relationship between one or more stimuli and the actions that need to be taken in response to the stimuli. The principal mechanisms used to indicate when a particular policy is applicable are *composition operators* that apply to stimuli (e.g. conjunction, disjunction, ...) and *consumption modes* that indicate which stimuli of the same type are combined when preparing for a response. These two aspects of situation description are supported using the event calculus through general definitions of `initiates` and `terminates` that support each of the composition operator/consumption mode pairs. The mapping onto the event calculus described below considers the composition operator *and* and the consumption mode *recent*. The definition of `initiates` is in figure 3 and the definition of `terminates` is in figure 4.

*Inititates.* Composite events are modelled as atoms that are true at certain points in time. A `recentAnd` starts to become true for a pair of operands if its operand events have both occurred and they are the two most recent such events. The definition of `initiates` is in two principal parts. The first part identifies pairs of events that have taken place and that match the arguments of a particular definition of `recentAnd` from the application (the definitions of `recentAnd` are asserted into the clause base as facts). For any pair of events that are possible arguments to the `recentAnd` the second part of the definition checks that there are no more recent candidate component events.

*Terminates.* An occurrence of a `recentAnd` ceases to be valid if an event can be identified that is a potential component, and that has taken place more

```
terminates(EId,recentAnd(DefId,_,LHS,RHS)) :-
    at(EId,EIdTime),
    signalledAt(RHS,RHSTime),
    recentAnd(DefId,_,LHSTemplate,RHSTemplate),
    LHSTemplate = LHS,
    holdsAt(RHSTemplate,EIdTime),
    signalledAt(RHSTemplate,RHSTemplateTime),
    RHSTemplateTime > RHSTime.
```

Figure 4: Definition of `terminates` for recent and.

recently than the components of the `recentAnd`. This is described using two alternative definitions for `terminates` which, respectively, stop a `recentAnd` from holding whenever the right hand or the left hand components become out of date. To consider the first of these, the definition from figure 4 first finds an event that matches the right hand operand of the `recentAnd`, and then checks to see if it took place after the component event. The definition for the left hand operand is very similar.

To consider the components of a `policy` definition given in figure 2.2: the `event` description is captured in the event calculus using the approach described above, i.e. by defining appropriate `initiates` and `terminates` rules; the `condition` description is translated into a query against the facts deduced by `holdsAt` from the events that have taken place; and the `action` maps to a series of assertions that particular events have taken place, which may, in turn, change the facts that hold or signal new composite events.

# 5 Validation Through Model Execution

Once the conceptual model has been described in terms of stimuli and policies, the event calculus formalization of these concepts can be used to generate explanations about distinct executions of the specification. This approach has been extensively described in [13]. Olivé and Sancho propose model validation through providing explanations of the results of model execution in three different ways: why the current state holds, how some intended effects can be achieved, and what would have happened if some other stimulus were given. Here we apply their ideas to our model of stimuli and policies, and show how the use of the event-calculus greatly simplifies this task.

*Explaining why* a given state has been reached, is just a question of tracing the processes undertaken by the system to arrive at this state. Using an event calculus approach, the events rather than the state are recorded; the state is derived from the trace of events. Showing this derivation process is a form of explanation in itself. This is achieved by a revised version of the *holdsAt* predicate where some tracing has been added to make the user aware of the internal process.

```
stimulus definition [
       name: returning([Person,Book])
       preconditions: [property(Person,has_books,Book)],
       exceptional_alternatives: [],
       initiates: [ property(Book,available,true) if true ],
       terminates: [
                property(Book,available,fail) if true,
                property(Person,has_books,Book) if true
       ],
       notification: [ ]   ].

stimulus definition [
       name: buying([BookOid, Title])
       preconditions: [],
       exceptional_alternatives: [],
       initiates: [
                property(BookOid,iof,book) if true,
                property(BookOid,title,Title) if true,
                property(BookOid,available,true) if true
       ],
       terminates: [ ],
       notification: [ ]   ].

policy definition [
       name: r1,
       event:   times(3,request_rejected([_,Book])
       condition: [(true)],
       action: [(
          property(Book,title,Title), % recovers the Book's title
           => nextOid(NextOid),       % obtains next identifier available
           => buying([NextOid,Title])
       )]].
```

Figure 5: Description of the *returning* and *buying* stimuli and the *buying* policy.

Unlike the approach in [13], the explanation process does not stop when the stimuli responsible for the current state are obtained. The system has to check whether these stimuli were caused by the user or were the result of a policy application, in which case the explanation should continue describing why this policy was applied. This forces the system to keep track of the policies being applied, and the recursive search resembles that used in forward chaining expert systems.

*Explaining how* a property $P$ can be satisfied involves providing an ordered sequence of stimuli that leads to the satisfaction of this property from the current state. Again, the use of the event calculus approach greatly simplifies this task: the explanation system identifies which stimuli initiate $P$; each stimulus in this set represents a potential path to make the property true. For each stimulus the system checks whether its preconditions are verified by the current state. If not, a recursive call obtains those stimuli that initiate this precondition without terminating the initial property $P$. This approach is quite similar to the one used

```
******** First explanation
Level: 1  Stimuli: returning([m1, b11]) with preconditions:
   [property(m1, has_books, b11)]
Level: 1  Stimuli: returning([m2, b12])  with preconditions:
   [property(m2, has_books, b12)]
Level: 0  Stimuli: requesting([m2, b11])  with preconditions:
   [property(b11, state, book_available), not property(m2, has_books, b12)]

******** Second explanation
Level: 1  Stimuli: buying([NextBookOid,"el-Quixote"])  with preconditions:
   []
Level: 1  Stimuli: returning([m2, b12]) with preconditions:
   [property(m2, has_books, b12)]
Level: 0  Stimuli: requesting([m2, b11])  with preconditions:
   [property(b11, state, book_available), not property(m2, has_books, b12)]

******** Third explanation
Level: 4  Derived stimulus: request_rejected([_,b11])
   from stimulus: requesting([m1, b11])
Level: 3  Derived stimulus: request_rejected([_,b11])
   from stimulus: requesting([m1, b11])
Level: 2  Derived stimulus: request_rejected([_,b11])
   from stimulus: requesting([m1, b11])
Level: 1  Derived stimulus: buying([NextBookOid,"el-Quixote"])  from rule:  r1
   with condition: [true]  and event: times(3,request_rejected([_,b11]))
Level: 1  Stimuli: returning([m2, b12])  with preconditions:
   [property(m2, has_books, b12)]
Level: 0  Stimuli: requesting([m2, b11]) with preconditions:
   [property(b11, state, book_available), not property(m2, has_books, b12)]
```

Figure 6: Possible outputs to the command *how(property(m2, has_books, b11))*.

in [13] where they follow the trace of the SLDNF proof tree for explaining how a derived fact is true in the current state.

Instead of requiring from the user the explicit interaction to cause the set of stimuli leading to $P$, another alternative is to ascertain which are the policies that cause any of these stimuli to be automatically applicable by bringing about the context where these policies are triggered.

As an example, consider that as well as the *requesting* stimulus from figure 2, two further stimuli *returning* and *buying* a book are of interest. Figure 5 shows their definition: *requesting* has as a precondition that the book be available and that the member has no books (as a member can keep only one book), *returning* requires that the member has the book, whereas *buying* does not have any preconditions. A policy is also given which states that a book is bought if three requests for this book have been rejected due to this book being on loan. The policy's event is defined as *times(3,request_rejected([_,Book]))* meaning that the three rejected requests should refer to the same book. The action recovers this book's title, a new book identifier and generates the *buying* stimulus.

Now suppose that the state of the system is that members *m1* and *m2* have

books *b11* and *b12*, respectively. The system is asked how member *m2* can get book *b11* through the command *how(property(m2, has_books, b11))*. Each answer corresponds to a path of the search tree. Figure 6 shows the output where level 0 stands for the root of the search and subsequent level numbers state the depth of the search. Where stimuli are found at the same level this means that their order is not significant.

The system ascertains from stimulus definitions that *requesting([m2, b11])* initialises *property(m2, has_books, b11))*. Now the process continues in proving how preconditions of this stimulus (i.e. {*property(b11, state, book_available), not property(m2, has_books, b12)*}) can be achieved since none of them are satisfied in the current state. The system then searches for stimuli initialising *property(b11, state, book_available)* (e.g *returning([m2, b12])* ) and terminating *property(m2, has_books, b12)* (e.g. *returning([m1, b11])*). For these level-1 stimuli, the system has in turn to check whether their preconditions are satisfied. As they are, the first explanation ends.

The second sequence achieves the satisfaction of the *requesting* precondition that the book be available through the stimulus *buying* rather than by making *b1* return the book where the title of the book borrowed by *b1* is *el-Quixote*.

The third sequence applies the policy so that instead of having the stimulus *buying* directly issued by the user, the system forces *buying* as a result of three requests for this book being rejected (i.e. the *request_reject* stimulus). In this way, the designer can be aware of this short cut, and realizes how the policy is wrongly defined since the rejection should come from different users.

# 6 Conclusion

We have presented the notions of stimulus and business policy as intuitive concepts along with their formalisation, in which aspects such as composition and consumption policies have been expressed in terms of the event calculus. Such formalization allows validation of the model through explanation i.e., the results of model execution can be questioned in terms of why a given state has been reached or how a given state can be obtained. In this way, users can check whether the system behaves according to their expectations. This process has been facilitated by the suitability of the event calculus.

Future work includes extending the explanation facilities, and proving the completeness and correctness of the specifications. Moreover, structural features (e.g. hierarchies, structural restrictions, derived data) have not been addressed and are left for future extensions along the lines presented in [13].

**Acknowledgements:** The authors wish to thank A. Olivé for enlightening discussions on conceptual modelling. We are in debt to A. Fernandes on whose event calculus implementation this work was built. We would like also to thank J. Iturrioz and M. Piattini for fruitful conversations on use cases.

# References


[1] M. Andersson and J. Bergstrand. *Formalizing Use Cases with Message Sequence Charts.* PhD thesis, Lund Institute of Technology, 1995.

[2] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 606–617. Morgan-Kaufmann, 1994.

[3] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, 1994.

[4] O. Diaz, J. Iturrioz, and M.G. Piattini. Stability: a criterion for software evolution enhancement. *Submitted for publication*, 1996.

[5] A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A logic-based integration of active and deductive databases. *New Generation Computing*, 1996.

[6] S. Gatziu and K.R. Dittrich. Events in an active object-oriented database system. In M. Williams N. Paton, editor, *Proc. of the 1st Intl. Workshop On Rules In Database Systems*, pages 127–142. Springer-Verlag, Workshops in Computing series, 1993.

[7] B. Henderson-Seller, M. Fung, and Lin Mei Yap. The role of business rules and quality in methodologies. *ROAD*, Nov-Dec:10–12, 1995.

[8] R. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 12:121–146, 1992.

[9] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[10] P. Loucopoulos. Conceptual modeling. *in [11]*, pages 1–26, 1992.

[11] P. Loucopoulos and R. Zicari (Editors). *Conceptual modeling, databases and CASE.* Wiley, 1992.

[12] T. Moriarty. Bussiness rule analysis. *Database Programming and Design*, 6(4):66–69, 1993.

[13] A. Olivé and M.R. Sancho. Validating conceptual specifications through model execution. *Journal of Information Systems*, 21(2):167–186, 1996.

[14] A. Tsalgatidou and P. Loucopoulos. An object-oriented rule-based approach to the dynamic modeling of information systems. In H.G. Sol and K.M. vanHee, editors, *Dynamic Modeling of Information Systems.* Elsevier Science Publishers (North Holland), 1991.