

# Reconstructing a Semantics for Action Rules by a Transformation to Almost Plain Prolog

*Bart Demoen, Phuong-Lan Nguyen*

*Report CW456, August 2006*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Reconstructing a Semantics for Action Rules by a Transformation to Almost Plain Prolog

*Bart Demoen, Phuong-Lan Nguyen*

*Report CW456, August 2006*

Department of Computer Science, K.U.Leuven

## **Abstract**

Action Rules are currently only implemented by B-Prolog. One reason is that it is difficult to extract their exact semantics from the manual or the implementation. This issue is attacked by describing a transformation from Action Rules to almost plain Prolog. The transformation captures most of the B-Prolog semantics, and shows clearly how it could be extended. It also allows for an easy to understand source level description of many optimizations.

# Reconstructing a Semantics for Action Rules by a Transformation to Almost Plain Prolog

Bart Demoen\* and Phuong-Lan Nguyen†

\* Department of Computer Science, K.U.Leuven, Belgium  
bmd@cs.kuleuven.be

† Institut de Mathématiques Appliquées, UCO, Angers, France  
nguyen@uco.fr

**Abstract.** Action Rules are currently only implemented by B-Prolog. One reason is that it is difficult to extract their exact semantics from the manual or the implementation. This issue is attacked by describing a transformation from Action Rules to almost plain Prolog. The transformation captures most of the B-Prolog semantics, and shows clearly how it could be extended. It also allows for an easy to understand source level description of many optimizations.

## 1 Introduction

Action Rules are described in [9] - see also [8] for their predecessor Delay Clauses. They offer a surface syntax for specifying conditional actions (the bodies of the rules) as a result of posted events.

As far as we know, B-Prolog is the only system that currently implements Action Rules. This is a pity, because Action Rules are very expressive and apparently, they form a superior specification tool for (finite domain) constraint solvers: see [9]. We see two reasons why other systems have not embraced Action Rules: (1) it is difficult to extract their semantics from published papers, the B-Prolog manual and the B-Prolog implementation. (2) The implementation of Action Rules seems to be tied to the TOAM. This paper attacks the first issue by describing a transformation from Action Rules to almost plain Prolog. The transformation captures the B-Prolog semantics as far as we understand it. The transformation also shows how some of the choices made by B-Prolog, or restrictions that it imposes, can be relaxed<sup>1</sup>, i.e. how the design space could be further explored. Finally, our transformation allows for an easy to understand source level description of many optimizations. The second issue - the apparent necessary link between Action Rules and the TOAM - is only partly resolved by our transformation: there is still some ground to cover from our transformation to an efficient non-TOAM implementation. [3] make a first step in the direction of an efficient Action Rule implementation in hProlog. A forthcoming report will deal fully with the efficiency issue in the WAM context.

---

<sup>1</sup> Whether this is desirable is a different matter

The implementation and the semantics of Action Rules keep evolving: the initial Delay Clauses of [8] have become Action Rules [9] and new application domains for Action Rules (like [6]) lead the implementation to initially unintended features. From the beginning, we have been struggling with understanding the intended semantics of Action Rules, and in particular of Action Rules with more than one rule with event patterns. We have spend many hours typing in far-fetched combinations of Action Rules which are not described/explained by the manual, just to find out how the implementation deals with them. At some point, we thought that we understood Action Rules good enough to start their implementation in hProlog [1]. We had a good idea about the low-level support we needed, but we realised soon that our understanding of Action Rules was still incomplete. We then decided to move our implementation plans one level up: to almost plain Prolog. There was a good chance that if we could express the meaning of Action Rules in Prolog, we would understand Action Rules and other people would as well. At first, we wanted to write an Action Rules interpreter in Prolog, but that turned out a bad idea: in order to be able to run some example programs, the interpreter would have to deal with most of Prolog, and that would put the focus on Prolog stuff instead of on the Action Rules. So we shied away from the interpreter very quickly. We settled for a transformation schema that transforms Action Rules to almost plain Prolog.

Almost plain Prolog is Prolog with attributed variables (see for instance [5]): most Prolog systems have attributed variables these days. That renders our transformation useful for all those systems. We use a particular variant of attributed variables: dynamic attributes [2]. They were adopted by SWI-Prolog [7]. They are easy to work with because no declarations are needed. The transformation schema is actually not concerned with the attributed variables mechanism: only some Action Rules *built-ins* (which could be realised as library predicates) need to know about them. For Prolog systems needing attribute declarations, or using a different surface syntax, these can be adapted easily. Moreover, the transformation schema is not meant to be used as a full-fledged implementation: its value is mostly educational. However, this paper is not for the complete novice: the reader will benefit most if she was exposed to Action Rules before.

There are two separate issues when describing the semantics of Action Rules by a transformation to Prolog: (1) the transformation of the Action Rules themselves; (2) an implementation of built-ins related to Action Rules (*post\_event/2* ...). The former is described in Section 3, the latter in Section 4. Section 6 describes some optimizations to the basic transformation schema. Section 5 compares our transformation with the B-Prolog implementation at the semantics level, and shows how some variants could be explored.

We have not used our transformation to run benchmarks. The specification defined by our transformation is executable, and can be used to experiment with Action Rules, but it is not meant to be used instead of a more sophisticated implementation: it is only meant to make Action Rules more easily and more widely understood. We therefore start with a section on terminology.

## 2 Terminology

According to [9], the general form of an Action Rule is

`Agent, Condition, {Event} => Action.`

The *Agent* looks like the head of a Prolog clause, but the semantics is not unification, rather matching. We often use the word *head* instead of *Agent*. We use the word *Guard* instead of *Condition*, because this part of the Action Rule functions like the guard in committed choice languages: once a guard succeeds, execution commits to that rule. The part  $\{Event\}$  will be denoted as *EventPats* because [9] refers to its constituents as *event pattern*. If no event pattern is present, the rule is named a commitment rule. The *Action* looks like a Prolog clause body (and we will often refer to it as the *Body*). We will assume that the *Agent* has only variables as arguments, and that these are all distinct: this can be achieved always by moving the matching of non-variable terms to the *Guard*. [9] restricts the form of the guards, but we will not be bothered at this moment with the restriction.

The head of an ordinary Prolog clause has a principal functor that is a predicate symbol. Similarly, the head of an Action Rule is related to an Action Rule predicate symbol. A predicate symbol cannot appear in the head of both an ordinary Prolog clause and an Action Rule. An agent is a call to an Action Rule predicate: such a goal can be suspended and activated. Ordinary Prolog goals do not get suspended.

The words *event* and *agent* are overloaded in Action Rule speech and this contributes to difficulties in understanding them.

## 3 Transforming Action Rules to Almost Plain Prolog

### 3.1 The transformation

Assume we have the following set of Action Rules for  $p/n$ :

```
p(X1, ..., Xn), Guards_1, EventPats_1 => Body_1.  
p(X1, ..., Xn), Guards_2, EventPats_2 => Body_2.  
...  
p(X1, ..., Xn), Guards_n, EventPats_n => Body_n.
```

$Guards_i$  takes the form of a conjunction of guards in the  $i^{th}$  rule: B-Prolog restricts the set of allowed guards, but it is not important for our translation schema which goals are allowed as guards. In principle one could support also disjunctions, guards with side-effects, guards binding variables etc.

The event patterns that we explicitly support in this paper are  $event(X, M)$ ,  $ins(X)$  and  $generated$ . It will be clear how to support more events. We will denote with  $\{\}$  the absence of events - although syntactically, this is not accepted by B-Prolog.

The transformation schema generates two Prolog clauses for the Action Rule predicate  $p/n$  above: one clause for predicate  $p/n$  and one clause for predicate  $suspended\_p/(n+2)$ . Predicate  $p/n$  is responsible for selecting the rule that is allowed to register the events.

```

p(X1,...,Xn) :-
  Closure = suspended_p(Mess,Alive,X1,...,Xn),
  (Guards_1 ->
    register_events(EventPats_1,Closure),
    ((isin(generated,EventPats_1) ; EventPats_1 == {}) ->
      Body_1
    ;
    true
  )
;
...
;
Guards_n ->
  register_events(EventPats_n,Closure),
  ((isin(generated,EventPats_n) ; EventPats_n == {}) ->
    Body_n
  ;
  true
)
).

```

Some explanation completes this:

- the goal `(isin(generated,EventPats_1) ; EventPats_1 == {})` can be evaluated to true or fail at transformation time, because the arguments are manifest
- also `register_events` has manifest arguments, and since we must have a definition for the predicate `register_events/2`, it can be evaluated at compile time to a point where only built-ins are called; later examples will show this by unrolling a call to `register_events/2` into a conjunction of calls to `register_event/2`: each of them deals with one event pattern
- `suspended_p/(n+2)` is a new predicate whose code will be shown shortly; its two new arguments are named Message and Alive:
  - Message indicates the variable for the message send in a `post_event(X,Message)` goal and thus corresponds to the second argument in an event pattern of the form `event(X,Message)`; an example will make this more clear in Section 3.2
  - Alive is a variable that represents the liveness of the agent: as soon as it is non-var, activating the agent has no more effect, i.e. amounts to true

`suspended_p/(n+2)` is executed when the agent is activated

The transformation schema is completed by one Prolog clause for the suspended agent:

```
suspended_p(Message,Alive,X1,...,Xn) :-
  (var(Alive) ->
    (Guard_1 -> kill(EventPats_1,Alive), Body_1
    ;
    ...
    ;
    Guard_n -> kill(EventPats_n,Alive), Body_n
    )
  ;
  true
).
```

The predicate `kill/2` has a very simple definition:

```
kill({},Alive) :- !, Alive = no.
kill(_,_).
```

Since the goal `kill(EventPats_1,Alive)` has a manifest first argument, it can be evaluated away at transformation time.

The life of an agent has three phases:

1. The life of an Action Rule predicate starts by a selection process in which the the guards select one rule which is allowed to register its event patterns, and (if *generated* is amongst them, or if there are no event patterns) executes its body.
2. The midlife of an agent consists in being activated any number of times: the guards select a rule which can execute its body. The event patterns in the selected rule are no longer relevant, except that if there are none, the agent is killed. During this phase, the agent is active, or sleeping.
3. The life of an agent ends when a rule is selected which has no event patterns. The agent becomes a zombie really, because posted events can still activate it. Killing an agent is done by binding the `Alive` variable. When a zombie agent is activated again, it simply succeeds.

### 3.2 A first example

Here follows a concrete example. Let the Action Rules be

```
p(A,B), f(A,B), {generated, event(A,M1), event(B,M2)} => b1(A,B,M1,M2).
p(A,B), g(A,B), {event(B,M3)} => b2(A,B,M3).
```

then our transformation generates:

```

p(A,B) :-
  Closure = suspended_p(Mess,Alive,A,B),
  ( f(A,B) ->
    register_event(event(A,Mess),Closure),
    register_event(event(B,Mess),Closure),
    b1(A,B,Mess,Mess)
  ; g(A,B) ->
    register_event(event(B,Mess),Closure)
  ) .

suspended_p(Mess,Alive,A,B) :-
  ( var(Alive) ->
    ( f(A,B) ->
      b1(A,B,Mess,Mess)
    ; g(A,B) ->
      b2(A,B,Mess)
    )
  ;
  true
) .

```

### 3.3 A second example

The next example shows the killing of an agent: killing is implicit when a rule is used which does not have any event patterns.

```

freeze(X,_), var(X), {ins(X)} => true.
freeze(_,Goal) => call(Goal).

```

results in

```

freeze(A,B) :-
  ( var(A) ->
    register_event(ins(A),suspended_freeze(Mess,Alive,A,B))
  ; true ->
    call(B)
  ) .

suspended_freeze(Mess,Alive,A,B) :-
  ( var(Alive) ->
    ( var(A) ->
      true
    ; true ->
      Alive = no,
      call(B)
    )
  ;
  true
) .

```

## 4 Action Rules Built-in Predicates

The event pattern *generated* has no explicit post associated to it. We show details related to two other event patterns: *ins/2* (the Herbrand solver) and *event/2* (B-Prolog events). Unification happens asynchronously, i.e. the Prolog unification routine intercepts the instantiation of a variable which has a goal waiting in its instantiation and puts the goal in a queue. Events are posted explicitly by calling the predicate *post\_event/2*.

### 4.1 Registering events

The event patterns we describe have the form *event(X,M)* and *ins(X)* in Action Rules. Our code for *register\_event/2* is

```
register_event(event(X,_),G) :- add_attr(X,event,G).
register_event(ins(X),G) :- add_attr(X,ins,G).
register_event(generated,_).    % ignore

add_attr(X,Mod,A) :-
    (get_attr(X,Mod,Old) ->
     New = [A|Old]
    ;
     New = [A]
    ),
    put_attr(X,Mod,New).
```

### 4.2 Posting an event and activating the agent

Posting a Herbrand event (corresponding to *ins/1*) consists in unifying a variable. Posting an event corresponding to the pattern *event/2* is done by calling the predicate *post\_event/2*:

```
post_event(X,Mes) :- get_attr(X,event,Gs), !, activate_agents(Gs,Mes).
post_event(_,_).
```

Assuming that *X* is a suspension variable and that it is unified with *Y* (which can be a term), a handler is called at some point for each attribute of *X*. For the *ins* attribute this means that activating the agent amounts to calling *ins:attr\_unify\_handler/2* which is defined as<sup>2</sup>:

```
ins:attr_unify_handler(AttrX,_) :- call_list(AttrX).

call_list([]).
call_list([X|R]) :-
    call_list(R),
    call(X).
```

<sup>2</sup> Our the handler deals only with the case of unification of a variable with a non-variable term: it is easy to extend the definition to deal with the other case.

Activating the agents on event/2 can be implemented as:

```
activate_agents([],_).
activate_agents([G|Gs],Mes) :-
    G =.. [N,M|R],
    NewG =.. [N,Mes|R],
    activate_agents(Gs,Mes),
    call(NewG).
```

Note that `call_list` and `activate_agents` are not tail-recursive: in this way it observes the B-Prolog order of activating agents.

The above is certainly not the most efficient way to implement all this: see [3] for a partial treatment of the efficiency issue. Also, B-Prolog distinguishes currently between breadth-first and depth-first event posting: we come back to that issue in Section 5.

## 5 Comparison with B-Prolog

- in B-Prolog, the body of an Action Rule cannot leave behind any open alternatives; in fact, the implementation itself makes sure there aren't any by putting a cut after the body; our transformation does allow for backtracking back into an Action Rule body, but it is of course easy to generate the extra cut as well; [9] says: *At a point during execution, there may be multiple events posted that are all expected by an agent. If this is the case, then the agent must be activated once for each of the events. If an agent is found to be active already when the system tries to add it into the active chain, the system makes a copy of it and adds the copy into the chain.* Detecting whether an agent could be reactivated by backtracking is expensive, and that could be a good reason to prefer deterministic Action Rule bodies
- the moment of activation of guards is not fixed: what we presented for the ins/1 event relies entirely on how the underlying system deals with Herbrand instantiation and attributed variables; the main point that [9] makes about the activation moment is related to the wish that events do not *get lost* because a choice point is created between the moment that the event is posted and the moment that the activated agent is scheduled; Prolog implementations handle this correctly for attributed variables
- the author of Action Rules says that the user should not rely on a particular scheduling strategy for activated agents for the correctness of the program; our transformation does not enforce a particular strategy
- we have mentioned already that B-Prolog restricts the guards and events: our transformation is in some sense more generic and it puts no such restrictions; the restrictions on the guards stem from the wish to compile efficiently and to let the guards not bind exterior variables; this could be relaxed, while keeping efficient compilation in the case the guards are *simple in-line goals*; also disjunctive conditions in guards could be allowed
- extending Action Rules with new event patterns requires that one specifies what it means to register a new event, how to post a new event, and how to handle it; depending on the flexibility of the low-level support, this can be done in a very modular way and remain efficient.
- the head in an Action Rule performs matching, not full unification: our transformation schema relies on matching being explicit in the guards; some Prolog systems

support matching; the ones that don't would need to get around it somehow in order to support the same semantics as B-Prolog

- the place to kill the agent is just after the guard succeeds; this is in line with the B-Prolog implementation; it surprised us that even if the only event pattern is *generated* the agent is not killed; this is just a design choice and not a fundamental issue
- when an event is posted to a channel variable by `post_event/2` B-Prolog activates agents in the same chronological order in which the events were registered; all activated agents are put in the active chain and this takes time linear in the number of activated agents before any agent can be executed; our implementation has the same characteristic; recently a new built-in `post_event_df/2` (and `/3`) was introduced in B-Prolog, which does an incremental addition to the active chain, and which does no longer respect the chronological order; it is not clear whether the semantics of these primitives is fixed, but as far as we know, our approach can handle them
- when none of the rules of an Action Rule predicate have event patterns, i.e. every rule is a commitment rule, the semantics of our transformation is exactly as intended by matching clauses
- the variable Message is (made) the same in all `event/2` event patterns: B-Prolog gives a warning or an error message when the user has specified two different message variables in the same clause, but our translation schema simply unifies them; code that is correct in B-Prolog behaves the same in our approach; B-Prolog requires that the message variable in an event pattern is a first occurrence: our transformation would also deal with non-first occurrences in a reasonable way, but there might be design decision not to allow it
- when the EventPats in an Action Rule is for instance  $\{event(X, M), generated\}$ , B-Prolog seems to ignore the *generated*; our transformation does not, but could be easily made to do so; a similar observation applies to other combinations of event patterns as well
- apparently, the goal `post(ins(X))` with X still free has a meaning in B-Prolog (and so have some others); we have not catered for its semantics, but it is clear that it is easily possible

## 6 Optimizations

We have already indicated the obvious optimizations related to inlining the calls to `register/2` and `kill/2`. A few other optimizations are also quite easy to perform:

- if none of the events is of the form `event/2`, the extra Message argument in the suspended agent predicate is not needed
- if two guards contain common parts (or opposite parts like `var(X)` and `nonvar(X)`), they can be factored out or removed; e.g.

```
var(X), var(Y) -> bla
;
var(X), nonvar(Y) -> foo
```

can be reduced to

```

var(X) ->
    (var(Y) -> bla
    ;
    foo
    )

```

This can be done at transformation time, or left to the underlying Prolog compiler in case it implements such optimizations

- the Alive argument can also be removed in some cases, e.g. if every rule has event patterns (and thus none of the rules kills the agent), or if the agent can only be activated once (as is the case with `freeze/2` - see the example in Section 3.3)

The above optimizations can be generalized as follows: suppose we make a new agent for each rule, i.e. for the  $i$ -th rule instead of `register_event(EventPatsi,suspended_p(Mess,Alive,X1,...,Xn))` we generate `register_event(EventPatsi,suspended_pi(Mess,Alive,X1,...,Xn))` where `suspended_pi` is a copy of `suspended_p`. We can now exploit the fact that `suspended_pi` is only activated as a result of one of the posting of an event in `EventPatsi` and that `Guardi` succeeded, but none of the earlier guards. This can lead to better code for `suspended_pi`. We illustrate this on a small example. For

```

p(A,B), var(A), {ins(A)} => true.
p(A,B), var(B), {ins(B)} => true.
p(A,B) => writeln(ok).

```

we start of with the generated code:

```

p(A,B) :-
    (
        var(A) ->
            register_events([ins(A)],suspended_p1(Alive,A,B))
        ;
        var(B) ->
            register_events([ins(B)],suspended_p2(Alive,A,B))
        ;
        writeln(ok)
    ).

```

```

suspended_p1(Alive,A,B) :-
  ( var(Alive) ->
    (
      var(A) ->
        true
      ;
      var(B) ->
        true
      ;
      Alive = no,
      writeln(ok)
    )
  ;
  true
).

suspended_p2(Alive,A,B) :-
  ( var(Alive) ->
    (
      var(A) ->
        true
      ;
      var(B) ->
        true
      ;
      Alive = no,
      writeln(ok)
    )
  ;
  true
).

```

It is clear that if suspended\_p2 is activated, it is because at the moment of the goal p(X,Y), X was nonvar, so the test var(A) equals fail in suspended\_p2. Furthermore, since suspended\_p2 was activated, it was because Y became nonvar, so the var(B) equals fail in suspended\_p2. This reduces suspended\_p2 to:

```

suspended_p2(Alive,A,B) :-
  ( var(Alive) ->
    Alive = no,
    writeln(ok)
  ;
  true
).

```

A similar reasoning reduces suspended\_p1 to

```

suspended_p1(Alive,A,B) :-
  ( var(Alive) ->
    (
      var(B) ->
        true
      ;
      Alive = no,
      writeln(ok)
    )
  ;
  true
).

```

By reasoning on the Alive argument, we can reduce all code to:

```

p(A,B), var(A), {ins(A)} => true.
p(A,B), var(B), {ins(B)} => true.
p(A,B) => writeln(ok).

p(A,B) :-
(
    var(A) ->
        register_events([ins(A)],suspended_p1(A,B))
    ;
    var(B) ->
        register_events([ins(B)],suspended_p2(A,B))
    ;
    writeln(ok)
).

suspended_p1(A,B) :-
(
    var(B) ->
        true
    ;
    writeln(ok)
).

suspended_p2(A,B) :- writeln(ok).

```

Note that the above Action Rules predicate is not intended to be covered by the B-Prolog semantics. In fact, Neng-Fa Zhou writes himself: *When a predicate contains multiple Action Rules, the events set watched by the first one must subsume the second one, and the second one must subsume the third one, and so on. This restriction is imposed because event registration (attaching the agent to suspension lists [of] a channel) is done only once for each agent.* While this subsumption restriction is not necessary to make sense of the compilation schema, it might make sense to impose it: B-Prolog gives a warning for the above example, but compiles it in line with our transformation.

*Disclaimer:* we do not claim that any of the above described optimizations is new; one can check that B-Prolog implements at least some of them and possibly more. In particular, B-Prolog doesn't generate the code for the guards twice. A straightforward compilation of the generated Prolog clauses would.

## 7 Conclusion

Our main aim was to make Action Rules more accessible. Our understanding and the subsequent description presented in this paper was derived from the B-Prolog manual, experimenting with B-Prolog, e-mail exchanges with their designer Neng-Fa Zhou and reading his papers. We realise that our interpretation might differ in some details from what he intended. Still, we hope that our description of Action Rules is more accessible and can form the basis for a more refined semantics of Action Rules. This could result in their wider acceptance and implementation in other Prolog systems as well. Also, their surface syntax might not be ideal, but they offer a much needed functionality for

writing constraint propagators. Action Rules already surpassed that niche and it is also used for interactive graphics applications. It could widen its scope to become even more general purpose, just like CHR [4] evolved from a constraint specification language to a general purpose language. Based on our transformation, it is easy to integrate Action Rules in any Prolog system, based on term\_expansion and a small library of Action Rules related built-ins: this caters for a suboptimal implementation, but it would be very useful. An implementation along those lines is underway for SWI-Prolog.

## Acknowledgements

We have benefited from e-mail exchanges with Neng-Fa Zhou. Part of this work was conducted while the first author was a guest at the Institut de Mathématiques Appliquées of the Université Catholique de l'Ouest in Angers, France. Sincere thanks for this hospitality.

## References

1. B. Demoen. hProlog. <http://www.cs.kuleuven.be/bmd/hProlog/>.
2. B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Computer Science, K.U.Leuven, Belgium, Oct. 2002.
3. B. Demoen and P.-L. Nguyen. Delay and events in the toam and the wam. In *Proceedings of CICLOPS - Colloquium on Implementation of Constraint and Logic Programming Systems*, 2006. accepted.
4. T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37:95–138, 1998.
5. C. Holzbaur. Meta-structures vs. Attributed Variables in the Context of Extensible Unification. In M. Bruynooghe and M. Wising, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 260–268. Springer-Verlag, Aug. 1992.
6. T. Schrijvers, N.-F. Zhou, and B. Demoen. Translating Constraint Handling Rules into Action Rules. Third Workshop on Constraint Handling Rules, Report CW 452, K.U.Leuven, Department of Computer Science, July 2006.
7. J. Wielemaker. SWI-Prolog release 5.4.0, 2004. <http://www.swi-prolog.org/>.
8. N.-F. Zhou. A Novel Implementation Method for Delay. In *Joint International Conference and Symposium on Logic Programming*, pages 97–111. MIT Press, 1996.
9. N.-F. Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming (TPLP)*, 6(5):483–508, 2006.