

Information and interaction in MarketSpace and their implementation in Prolog (Position Paper)

Joakim Eriksson Niclas Finne
Telia Research AB, S-123 86 Farsta, Sweden
joakime@sics.se nfi@sics.se

Sverker Janson
Swedish Institute of Computer Science
SICS, Box 1263, S-164 28 Kista, Sweden
sverker@sics.se

August 9, 1996

1 Introduction

Market infrastructures

Current web-based commerce is by and large a replica of real-world commerce. Customers are supposed to visit various electronic storefronts, inspect the digital displays of goods and services, and, perhaps after some comparison shopping, place an order.

Unfortunately they have to do it in person. Attempts at supporting the customer through “intelligent agents” are guaranteed to be in vain, unless the underlying model is changed. It would have to be one bright agent to, using the web as is, find a few good deals on a fridge at nearby resellers, haggle prices and delivery conditions, and then present the three best alternatives. Which minute fraction of the 50000 hits on “refrigerator” in AltaVista is at all relevant? In these, what part of the text is the price, the model, the delivery conditions?

Needless to say, much better models are possible, in which the above would be a very basic service, and one such model will sooner or later supplant the web-based models. The web, and associated technologies such as Java, will remain as the user interface, but a new infrastructure will emerge that is targeted to the task of creating a near-perfect global market.

In this position paper, we present work towards such an infrastructure, characterised by *openness* and by being based on a paradigm of *interacting agents*.

We would like the market to be open like the web. Nobody should own it. Anyone should be able to enter it. Anyone should be able to offer any kind of service, such as brokering. This is typically not the case with current markets, where all information is owned by the market operator and not available for use by others [1, 2].

A market has to support more activities than just finding the desired product or customer. Well-defined interaction protocols are needed for negotiation. Ideally, these protocols should make sense for human-human, human-agent and agent-agent interaction alike, to make possible any mix of human and automated participants in the market. (For an agent-only market, see, e.g., Kasbah [3].)

Note that although the issues of security and payment are strongly emphasized in most works on electronic commerce, they are quite orthogonal here. Any future standards will do.

MarketSpace and logic programming

Our MarketSpace model is at the implementation level similar to other Internet-based infrastructures. Prolog is as always an efficient prototyping tool, but the application area at hand places entirely new requirements on languages and systems, such as a need for multithreading and persistence. Constraints would seem particularly useful for some purposes, but again new requirements affect their use. We will offer our preliminary thoughts on these issues in the conclusions.

Organisation of this document

In the remaining sections, we outline the information and interaction models that support our preliminary MarketSpace infrastructure, describe the design of a MarketSpace server prototype and its implementation in SICStus Prolog and Prolog Objects [4], and finally offer a few concluding remarks.

2 Information and interaction

Information

Interests are the essential pieces of information in a market. They describe what a participant (person, company, organisation, etc) is concerned about, curious about, intends to do or wants. There are several different kind of interests from simple forms of interests like “I am interested in cats” to more complicated like “I want to buy all things needed to build a spacecraft and hire personnel to build it”. Since the goal of a market participant is to make deals, an interest can be said to describe a set of potential deals. We will now try to make these concepts a little more precise.

What we will need to know about a *deal* is that it has a type, some participants involved, and a time when it is entered. As basic types we have (the names of) *participants* in the market, which are either human or programmed, and *time points*. The *deal type* includes everything else, such as what is sold, delivery conditions, etc. With each deal type is associated a set of *rôles*, which are the parts played by participants in a deal, such as buyer or seller. A deal assigns participants to all the rôles in its type.

An *interest* is a set of deals. An *expression of interest (eoi)* is a representation of an interest.

It is clearly desirable that expressions of interest are both expressive and computationally tractable, and using constraints springs to mind, but we have not yet explored this possibility.

DEAL HEAD	
TIME	now until now+T
PARTICIPANTS	A=P1, B=P2
DEAL TYPE	"product trade..."
DEAL CONTENT	
ROLES	Seller(P1),Buyer(P2)
TRANSACTION	T1(P1->P2:X, P2->P1:Y)
ITEMS DESCRIPTION	
X:	book{Title = "...", ...}
Y:	money{Currency = SKr, Amount = 100}
TRANSACTION INFORMATION	
T1.delivery(X)	"mail"
T1.payment(X)	"cash on delivery (COD)"
T1.warranty(X)	none
T1.delivery(Y)	"postal giro"

Figure 1: Example of a possible EOI

The interaction model

Based on the interests, we define a very simple speech-act style interaction protocol. It consists of *information messages*: ask and tell, and *negotiation messages*: propose, reply, accept, and reject.

$ask(A, B, eoi, id)$

A asks B for information on interests, giving an eoi (possibly but not necessarily its own interests) as guidance. Replies can be given by tell.

$tell(A, B, eoi), tell(A, B, eoi, id)$

A tells B about some interests (possibly its own). If this is intended as an answer, the question can be referred to by its id.

$propose(A, B, eoi, id)$

A initiates negotiation with B, giving an interest as guidance. Replies are given by reply, accept, or reject.

$reply(A, B, id, eoi, id)$

A replies to B's latest bid. The interest given need not have any special relation to the bid. (It is up to the participants involved to assess the progress of the negotiation and interrupt it when necessary.)

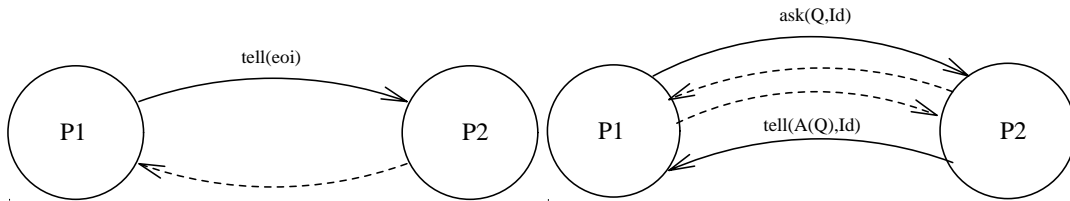


Figure 2: Tell and Ask

accept(A, B, id, id)

A accepts B's latest bid. This is equivalent to signing a contract. The reply is to accept or reject.

reject(A, B, id)

A aborts the current negotiation with B.

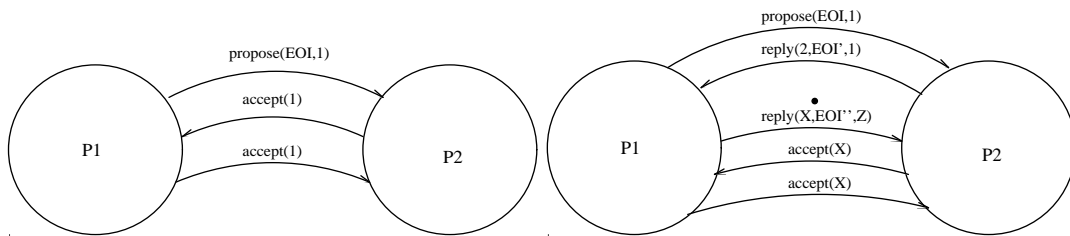


Figure 3: Negotiation

This simple protocol can be regarded as a subset of a richer more KQML-like protocol, other elements of which will be needed [5]. We are currently exploring conversational styles and idioms using these basic messages, rather than integrating capabilities through complex message types. This will allow some, simpler, participants to have a more naive view of interaction, while more advanced participants can recognize and enjoy the benefits of more elaborate patterns of interaction.

3 A MarketSpace server

We have developed a MarketSpace server architecture and prototype implementation based on the ideas presented above. The prototype was developed using SICStus Prolog (version 3) and Prolog Objects.

The rôle of this server in the MarketSpace infrastructure is both to support human participants, with whom it will interact by emulating a web server, and programmed participants (agents), for which it will serve as a scheduler and communication mediator.

Since this paper is intended for a readership interested in logic programming and its use for Internet applications, various aspects of the implementation will be described in more detail than otherwise called for. The purpose is to emphasize advantages of the object-oriented programming model, as well as the roundabout coding style needed in a single-threaded system.

The marketserver architecture

The MarketSpace server architecture has three main components (see Figure 4):

- the *kernel*, which handles the events that schedule activities in the system and communication with the outside world;
- the *protocol handlers*, which register the protocols they listen to and get the corresponding events from the event handler when data arrives;
- the *agent environment*, which implements the runtime environment for agents.

Two event types are built-in: The *time* events are used by system components (objects) that wish to be scheduled at regular time intervals and the *stream* events signal the arrival of data from the outside world to system components such as the protocol handlers.

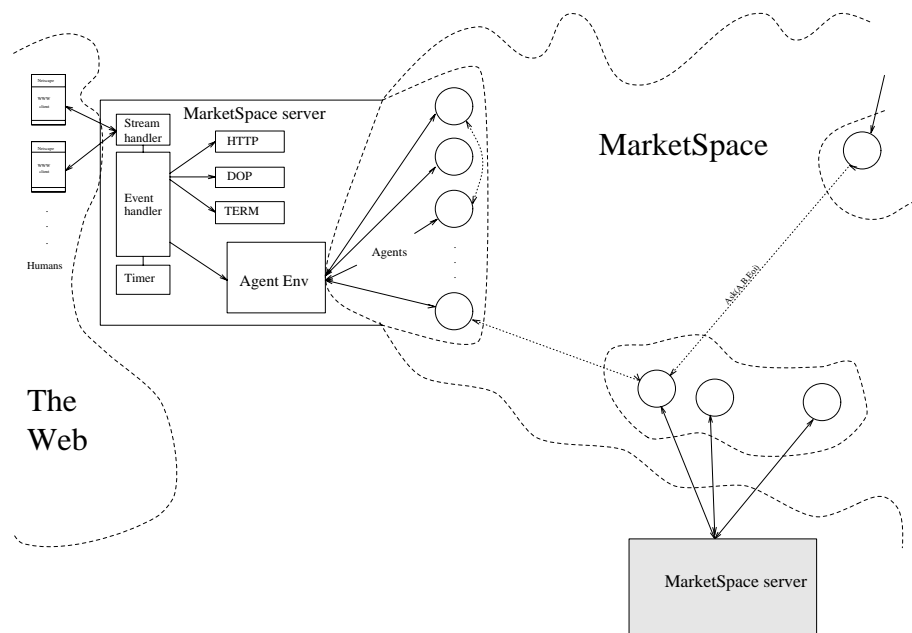


Figure 4: The basic MarketSpace architecture

The object model

The implementation relies heavily on the Prolog Objects extension of SICStus Prolog. The basic objects (prototypes) are *root*, which adds initialisation and finalisation capabilities, and *persistent*, which also adds the ability to save (checkpoint) and recreate state from secondary storage. The persistence mechanism is also used for migration of objects.

Since objects and object implementations should be mobile there is a mechanism to load required object implementation files. When new object implementations are added to the system, it loads the implementation file and detects new object types. New object types are registered in a database enabling the implementation file to be restored when the object types are needed.

The kernel

The kernel is the main engine of the system and has two important rôles: one is to generate stream and time events, the other is to distribute events among subscribing objects. The kernel consists of the following objects:

- the *stream handler* object handles the communication from the outside world with the system using socket and stream communication;
- the *time handler* object handles generation of time and delay events;
- the *event handler*, which is the main kernel object, and handles scheduling of events to the system objects.

The functionality of the kernel can be seen as two systems, one is the event system and the other is the stream and protocol system.

The *event system* generates stream and time events using the time handler, time agent and stream handler objects. If there is a need for more event types, the *register event* and *add event* methods can be used. An example of this is how time events are generated.

1. At initialisation, the timeagent registers the time event, adds a time event and subscribes to time events.
2. Each time the timeagent receives a time event, it makes a request for the next time event at the time handler.

Objects that subscribe to events should implement the method *event*, by which they are delivered to the object by the event system. As seen in Figure 5 there are two

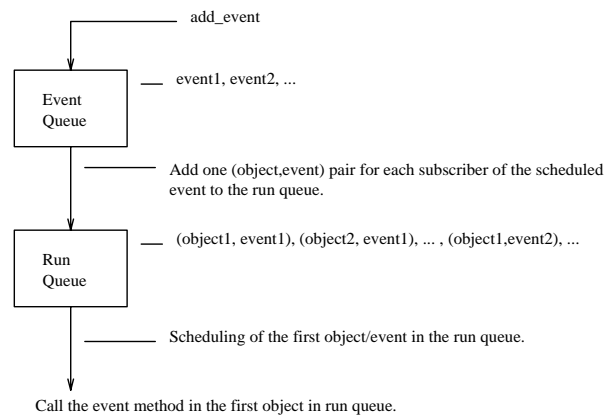


Figure 5: The event loop - event scheduling in the event handler

queues in the event system. One queue is for queueing incoming events and the other acts like a run queue. Each time the event loop is executed, the first event in the event queue is dequeued and all subscribers to that event is placed in the run queue. After the event has been dequeued, the event loop will schedule the first subscriber in the run queue.

```

eventhandler::{
  [ ... ]
  add_event(Event):-
    event_db(Event,_Y),
    get(events(Q)),
    queues:queue_last(Event,Q,NewQ),
    set(events(NewQ)) &
  register_event(Agent,Event):-
    \+ event_db(Event,_Y),
    assert(event_db(Event,[Agent])) &
  [ ... ]
  subscribe_event(Agent,Event):-
    subscription_db(Event,List),
    lists:non_member(Agent,List),
    update(subscription_db(Event,[Agent|List])) &
  [ ... ]
  scheduler:-
    schedule_event,
    schedule_agent,
    streamhandler::select(Reads),
    handle_reads(Reads),
    timehandler::get_next_event,
    scheduler
}.

```

The Prolog code above is part of the *event handler*. The *scheduler* method is the event loop which keeps the system running, using the stream handler and the time handler to generate events.

The *stream and protocol system* generates the stream events and takes care of stream communication with the outside world. The main component of this system is the stream handler which sets up a port using the socket library through which it receives all incoming data. When a new stream is opened through the port, the event handler will read data from the stream until it can determine which protocol is used. Then, a stream event is generated marked with a protocol tag (e.g. *http(Stream)*).

The protocol database is empty at initialisation time. For each protocol, its id and header have to be registered.

```

streamhandler::{
  [ ... ]
  init(Port):-
    sockets:current_host(Host),
    sockets:socket('AF_INET', Socket),
    user:on_exception(system_error(X)
      ,sockets:socket_bind(Socket, 'AF_INET'(Host,Port))
      ,(utils:lformat(5,'*** Error binding socket: ~w~n',[X])
      ,user:fail)),
    sockets:socket_listen(Socket,5),[ ... ] &
  [ ... ]
  select(ReadStream):-
    get(socket(Socket)),
    get(streams(Streams)),
    sockets:socket_select(Socket,New,Client,0:200,Streams,ReadStream),
    [ ... Handle new stream ... ] &
}.

```

The above Prolog code is part of the *stream handler*, which handles all socket communication. The *init* method shows the initialisation of the communication socket and the *select* method shows the listening for new data. The *socket_select* predicate both check if there is a new stream opened and if there is new data on the old streams.

The protocol handlers

The protocol handlers handle data arriving from the outside world. They register the protocols and subscribe to stream events from the event handler. The following Prolog code from the term handler object illustrates the interaction between the protocol handler and the event handler.

```
termhandler::{
    super(root) &
    [ ... ]
    init:-
        self(Self),
        protocols:term_header(Term),
        eventhandler::register_protocol(term,Term),
        eventhandler::subscribe_event(Self,term(_))&

    finish :-
        self(Self),
        eventhandler::unsubscribe_event(Self,term(_)) &

    event(term(Stream)):-
        protocols:read_term_message(Stream,Term),
        [ ... Execute the Prolog term and get a result ... ]
        protocols:send_term_message(Stream,Result),
        eventhandler::end_stream(Stream)
}.
```

The *init* method takes care of the registration of the protocol and the subscription to stream events, the *finish* method the unsubscription, and the *event* method the reception of stream events.

The *HTTP handler* handles World Wide Web communication and forwards parsed HTTP messages to other objects. (A DCG parser could easily be derived from the HTTP specification [6].) Objects can subscribe to a specific path whereby web accesses matching that path will be forwarded to the subscriber. The following system objects subscribe to HTTP events:

- *htmlserver* - Subscribes to the empty path and receives all web requests not caught by other subscribers. Replies with files matching the web request.
- *cgibinserver* - Subscribes to the path "cgi-bin" and handles user registration and authentication.
- *cgiobject* - Subscribes to the path "cgi-object" and provides a way to inspect Prolog Objects from the web.

The *DOP handler* handles communication with the DOP, Distributed Object Protocol, which is a simple protocol (defined by us) for distributing objects. When an

object is sent to another server, the state and a list of required object types are extracted from the object and sent in a DOP message. When a message arrives the DOP handler parses it and creates an object of the specified object type. If that object type or any other required object type is unknown, the creation is suspended until the missing type becomes known. After the object has been created, it must self take care of whatever it is supposed to do. For example, deliver itself to an agent.

The *TERM handler* handles communication with the TERM protocol. TERM messages are parsed into Prolog terms and executed within the server context. The result is returned to the caller. This makes it possible to examine and control a running server which is very useful for debugging.

The agent environment

At the top level of the architecture we have built a simple agent environment in which programmed participants in the marketplace can reside. The agent environment implements a small set of features useful for agents. These are

- Registration of requirements and description
- Agent communication
- Retrieval of information about other agents

The agent environment can be divided into two components. One is the *agentenv* object which implements a set of features for agent communication and registration. The other is the administrator agent, which handles simple questions about the local (and in the future also the distributed) agent environment.

```

collectoragent::{
  super(persistent) &
  [ ... ]
  init(state(none)) :-
    self(Self), agentenv::register_agent(Self,[],[time]) &

  finish :-
    self(Self), agentenv::deregister_agent(Self) &

  time:- [ ... ]

  incoming_message(Msg,From) :-
    Msg::class(MsgType),
    incoming_msg(MsgType,Msg,From) &

  incoming_msg(interact_aipv1,Msg,From) :-
    [ ... Handle the interaction message ... ]
}.

```

The above Prolog code is part of a simple agent and illustrates how an agent interacts with the agent environment. The first method, *init*, shows the initialisation of the agent when it is recreated and *finish* the clean up before the agent is abolished. The *time* method is called at regular intervals since the time event was subscribed to in *init*. The *incoming message* method exemplifies how the agent handles reception of messages.

Interests and interactions

For the present, the interests of the general model have a fairly simple representation:

- Items, like products to sell, are represented as objects with a set of methods for accessing information about concepts and instance values.
- Deals are represented as different objects according to deal type.
- EOIs, Expressions Of Interest, are represented as objects with a list of deal objects. Methods for accessing EOI information (like deals) exists as well as set operations like *union*, *intersect* and *partition*

The interaction model is implemented using interaction (or message) objects. They inherit from the persistent object (enabling migration) and have methods for accessing message information and sending the message to another agent.

To send an interaction object to another agent, the sender call the corresponding method.

```
ia::{
  super(simpleagent)&
  [ ... ]
  rcv_ask(Eoi,Id):-
    get(general_eoi(MyEoi)),
    Eoi::intersect(MyEoi),
    tell(Eoi,Id)&

  rcv_reply(Eoi,_Ref,Id):-
    get(status(proposed)),!,
    update_latest_eoi(Eoi),
    get(precise_eoi(MyEoi)),
    (
      Eoi::intersect(MyEoi) ->
      set(status(dealing)),
      update_latest_eoi(Eoi),
      reply(Eoi,Id,_NewRef)
    );
    reject(Id),
    set(status(none)
  )&
  [ ... ]
}.
```

This Prolog code is part of the agent, *ia*, which performs simple price negotiation. The methods *rcv_ask* and *rcv_reply* are called when the agent receives ask and reply messages. The ask method intersects the incoming EOI object with its own interest, *MyEoi*, to find out if they have something in common. If so, the resulting EOI will be sent back as a reply using the *tell* method.

Example

Finally, we illustrate the use of this architecture and system by an interaction between two servers: the *blue server* and the *red server*.

The blue server provides a web interface to interact with human participants, who can create, change and remove interests from their interest store, and engage in negotiations, all through dynamic web pages (generated using the package by Cabeza and Hermenegildo). The blue server hosts a collector agent, which looks for new interests and forwards them to a broker agent at the red server with the purpose of matching interests.

The red server lets human participants specify their interests as URLs to web pages in which the interest is given in a special syntax. The red server fetches the specified web page and parses the interest. The red server also hosts a broker agent to which a human participant can delegate the task of negotiating a deal, giving upper and lower price limits.

In addition, a *log server* subscribes to log events from the other servers, and displays what is going on in the marketplace.

Example scenario

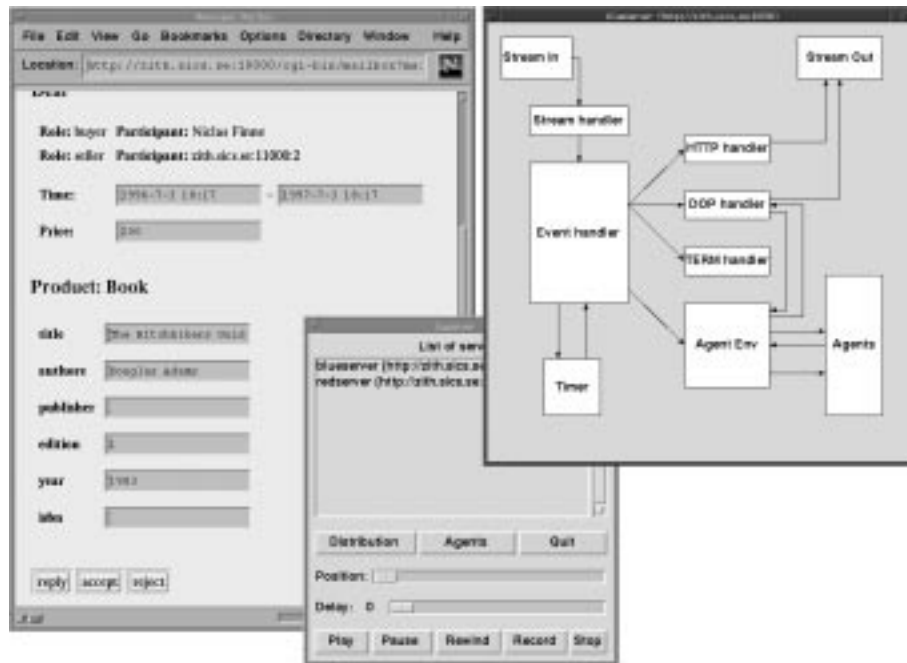


Figure 6: Snapshot of a proposal given to a user (left window) and the log handler

The following simple scenario illustrates a possible interaction in the marketplace.

- User *A* on the blue server specifies that she wants to buy a book. The user does this by creating an interest and specify the rôle to act (buyer, seller) and the product to buy or sell. This interest is stored in a user store which is accessible by all agents in the marketplace.
- User *B* on the red server specifies that she wants to sell a book by specifying a URL to a web page (in the special format). The web page is fetched and information about the interest is extracted and put in the user store.
- User *B* on the red server delegates her interest to a broker agent through an agent interface to which the broker agent has registered. When delegating the interest, the

user specifies an acceptable price and a suitable starting price. During negotiation, the broker agent starts with the starting price and slowly goes to the acceptable price trying to get the best price.

- A collector agent in the blue server collects all local interests and sends them to the broker agent. At the moment this is done at regular intervals, but in the future it will probably be some form of subscription. To collect all local interests, the collector agent starts by asking the administration agent for all local agents (which also includes users). Then the agent asks each such agent for their interests and when the agent happens to be a user, the user handler catches the message and answers it by fetching the users interests from the user store. The union of all interests found are sent to the broker agent at the red server.
- The broker agent receives the interest and checks for a matching interest by taking the intersection between the received interest and the delegated interest. If the intersection is not empty, the broker agent changes some information as for example price and sends a proposal to the agent that owns the interest. In this example it is user *A* at the blue server but it could just as well been a programmed participant.
- User *A* receives a message from the broker (see Figure 6) and the message is converted into a web page when the user reads it. The web page describes who the sender is, the type of message and its contents together with buttons to do various actions with it. For example, the user can choose between *accept*, *reject* and *reply* when replying to a proposal. The user chooses to negotiate by changing the price and selecting *reply*. A reply message is created and sent to the broker agent.
- The user and the broker agent negotiates until both accepts a deal (or someone rejects in which case the broker agent tries to find another agent to negotiate with).
- When the broker agent finally settles a deal, the user *B* is informed.

4 Conclusions

In summary, focusing on the logic programming aspects of our work,

- Prolog (and Prolog Objects) is, as usual, an efficient prototyping and implementation tool, and also for Internet applications;
- constraints might offer the expressiveness needed for representing and manipulating interests in MarketSpace;
- the need for a multithreaded Prolog becomes painfully apparent;
- although it is easy enough, Prolog is not suitable for migrating code in an open system due to security problems.

There are nice solutions to many of the security problems in various languages and implementations, but to be successful a market infrastructure cannot depend on the quirks of minor programming languages.

The same holds for the use of constraints. If they are to be used, this is either invisibly, or in a way that is easy to reconstruct by independent developers of software components for the market infrastructure.

Multithreading is a more neutral issue. In this, and other, Internet-applications, (pre-emptively scheduled) light-weight threads are clearly useful. The work-around

with tasks scheduled “by hand” employed here is very painful. We expect that most Prolog implementations will offer multithreading in the very near future.

Finally, for those who are interested in exploring systems of this kind using Prolog, we offer our prototype as a starting point (see <http://www.sics.se/isl/commerce/>).

References

- [1] Internet Shopping Network. URL: <http://www.internet.net>.
- [2] Polycon AB. The webra marketplace. URL: <http://www.polycon.fi/webra>.
- [3] Anthony Chavez and Pattie Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proceedings of PAAM'96*, 1996. Also available as <http://lcs.www.media.mit.edu/groups/agents/Publications/kasbah-paam96.ps>.
- [4] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 1995. Also available via <http://www.sics.se/isl/sicstus.html>.
- [5] Tim Finin and Richard Fritzon. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management*, 1994. Also available via <http://www.cs.umbc.edu/kqml/papers/kqml-acl.ps>.
- [6] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. available at <http://www.w3.org/pub/WWW/Protocols/HTTP1.0/draft-ietf-http-spec.html>.