The VLDB Journal 10: 16–38 (2001) / Digital Object Identifier (DOI) 10.1007/s007780100045

Integrating and customizing heterogeneous e-commerce applications

Anat Eyal, Tova Milo

Computer Science Department, Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel; E-mail: {anate, milo}@math.tau.ac.il

Edited by F. Casati, M.-C. Shan, D. Georgakopoulos. Received: 30 October 2000 / Accepted 14 March 2001
Published online: 2 August 2001 –

Abstract. A broad spectrum of electronic commerce applications is currently available on the Web, providing services in almost any area one can think of. As the number and variety of such applications grow, more business opportunities emerge for providing new services based on the integration and customization of existing applications. (Web shopping malls and support for comparative shopping are just a couple of examples.) Unfortunately, the diversity of applications in each specific domain and the disparity of interfaces, application flows, actor roles in the business transaction, and data formats, renders the integration and manipulation of applications a rather difficult task. In this paper we present the *Application Manifold* system, aimed at simplifying the intricate task of integration and customization of e-commerce applications. The scope of the work in this paper is limited to web-enabled e-commerce applications. We do not support the integration/customization of proprietary/legacy applications. The wrapping of such applications as web services is complementary to our work. Based on the emerging Web data standard, XML, and application modeling standard, UML, the system offers a novel declarative specification language for describing the integration/customization task, supporting a modular approach where new applications can be added and integrated at will with minimal effort. Then, acting as an application generator, the system generates a full integrated/customized e-commerce application, with the declarativity of the specification allowing for the optimization and verification of the generated application. The integration here deals with the full profile of the given e-commerce applications: the various services offered by the applications, the activities and roles of the different actors participating in the application (e.g., customers, vendors), the application flow, as well as with the data involved in the process. This is in contrast to previous works on Web data integration that focused primarily on querying the data available in the applications, mostly ignoring the additional aspects mentioned above.

Key words: Electronic commerce – Data integration – Application integration

The work is supported by the Israel Science Foundation founded by the Israel Academy of Sciences and Humanities and by the Israeli Ministry of Science

1 Introduction

Electronic commerce applications support the interaction between different parties participating in a commerce transaction via the network, as well as the management of the data involved in the process [50]. The Internet provides access to a large and diverse body of such applications, e.g., book and record stores, supermarkets, train and airline reservation systems, etc. The variety of available applications makes electronic shopping appealing for both customers and entrepreneurs. From the customer's viewpoint, the provision of electronic stores makes comparative shopping possible, allowing customers to browse, compare, and order goods selectively. From the entrepreneur's viewpoint, the variety of available applications generates new business opportunities for providing new services by integrating, enhancing, or customizing existing e-commerce applications:

1. Comparative shopping services are obtained by integrating several stores, providing the user with a uniform interface for posing requests, and having the application interact with the different stores to find the best bargains (for example, see [22,18]).
2. Integration of complementary services may also be useful. For instance, existing airline, train, car rental, and hotel reservation systems can be combined within one application offering an integral traveling service.
3. Existing applications may also be customized for special needs. For example, a pornographic bookstore can be obtained from a regular bookstore by supplying a wrapper that provides the appropriate user interface, restricts the search to the above category, and possibly gives some additional facility for hiding the customer's identity.

Unfortunately, the diversity of applications in each specific domain and the disparity of interfaces renders the integration and manipulation of applications a rather difficult task: application data (e.g., store catalogs, form attributes) may have different formats, making the collection and comparison of data complex. In addition, the application flow, the roles of the various actors participating in the application, and the API may vary widely among applications, complicating the coordination of activities performed in the component applications.

To some extent, standardization may help here. Indeed, in recent years there has been an intensive effort to develop standards for e-commerce applications in specific domains [10,17,28,20]. This however provides only a partial solution: (i) customization of existing applications to the new standards faces essentially the same challenges as in item 3 above; (ii) although more uniform, the new standards obviously still leave some design freedom to the application developers; and (iii) one may still want to utilize existing useful applications even if not conforming to the new standards.

A satisfactory solution to the integration and customization of e-commerce applications has to deal with the application flow and interaction between the various parties participating in the business transaction, as well as with the data involved in the process. In recent years there has been a significant amount of research on *data* integration and customization (see for instance, for a very small sample, [11,14,12,37,32,39,21]). The focus of these works, however, has been on *querying* the data available in Web applications. The additional services offered by the applications, the roles of the various actors participating in the business transaction and the interaction between them, and the application flows, were mostly ignored (except perhaps for some description of the screen sequence needed to be traversed to obtain the searched for data). In contrast, the *Application Manifold* system (AM for short) presented in this paper offers an integral solution to the integration/customization problem, covering, in one framework, both the data *and* the operational aspects the of e-commerce applications. The system provides:

1. A novel declarative specification language for specifying the integration and customization task; then, acting as an application generator, the system generates a *complete* integrated/customized e-commerce application, with the declarativity of the specification allowing for the optimization and verification of the generated application.
2. An infrastructure based on the emerging Web standard, XML (the eXtended Markup Language [45]) and UML (the Unified Modeling Language [41]), for the modeling of the applications data and flow, respectively.
3. A modular approach where new applications can be added and integrated at will with minimal effort.

The specification of an AM application then has two parts: the first models the target integrated/customized application with which the user will interact. (We will refer to this application as *global*.) The second describes the actual underlying Web applications and their relationship to the global one. (We will call these applications *local*). A key observation is that local applications can be modeled as special *views* of the global application. This is in a sense an adaptation of the *Information Manifold* (IM) paradigm [37], used for data integration, to the context of e-commerce applications integration (hence the name *Application Manifold*). In IM, local data is modeled as a traditional view of the global data. Then, queries on the global data are processed by rewriting into queries over the local data. A significant difference here is that while IM focuses only on *data* and *queries*, AM covers the full profile of e-commerce applications: at execution time, *all* the operations/interaction of the various actors participating in the global application are rewritten into relevant processing in terms of the local applications, (possibly with some additional global processing). This, of course, requires the AM view specification language to go beyond the traditional data-oriented view languages and handle all aspects of the application and not just the data. Consequently, AM introduces a novel rewriting paradigm, enhancing the traditional query rewriting with a *refinement* mechanism dealing with the operational aspects of the application. A major advantage of the approach is that the addition of a new local source to an integrated application is rather convenient, typically requiring only the specification of the source view in terms of the global application (and the supply of a corresponding wrapper for the source).

Limitations. The scope of work in this paper is limited to web-enabled e-commerce applications. We do not support the integration/customization of proprietary/legacy applications. The wrapping of such applications as web services is complementary to our work. Choosing to describe local application as views over a global application (see also discussion in next section) has some tradeoffs. The main advantage is that by using the global application as a base model and describing local applications in the way they relate to this model, we actually define the semantic and glossaries of the integrated application in one coherent framework. No further mapping between data or methods is needed. However, a resulting limitation is that one can only use data and method of the local applications, that are specified in the global model. For example if a bookstore offers a special feature as publishing the TOC, which is not included in the global model, it will not be available for users of the global integrated application. Similarly, the granularity of the global application building blocks cannot be more refined than that of the local ones. For example, if the global application offers a purchasing process consisting of two separate steps – (i)submission of delivery information; and (ii) payment – it will not be possible to use a local application where the two actions can only be performed simultaneously in a single screen/command. Thus, to utilize as many local applications as possible, the structure and granularity of the global application should be specified carefully using the available knowledge on the relevant local applications.

The paper is organized as follows. We start in Sect. 2 by considering related work. Section 3 introduces our data model, query language, and application flow model on which we rely for the specification of global and local applications, and the system architecture. In Sect. 4 we introduce our running example and use it to illustrate various components of an AM application specification. The semantics of such a specification is then explained in Sect. 5. The system implementation is discussed in Sect. 6. Finally, we conclude in Sect. 7.

2 Related work

Our data model and specification language are inspired by [1], where a system called *ActiveViews* was proposed for the generation of new e-commerce applications. However, the goal of our AM specifications (hence, also the semantics of the specifications and type of the corresponding generated applications) is completely different: while ActiveViews specifications were used for constructing *new* e-commerce applications

from *scratch*, the AM system tackles the integration and customization of *existing* Web applications; this is not supported at all by ActiveViews.

Software integration vs data integration. In recent years, significant research effort has focused on software modules composition and reuse. Much work has been done in the context of object-oriented programming, mainly considering components and frameworks for semi-finished software architectures [31, 40]), including composition models, languages, tools, and methods for constructing flexible software systems. A higher-level approach, more suitable for the composition of Web applications, is Megaprogramming [48]. For instance, the CHAIMS project [13], handles the composition of remote services, typically provided by autonomous suppliers. As in AM, the architecture is based on a repository of services including locations, protocols, and data types, and of wrappers for services not supporting the CHAIMS protocol. However, the focus of work in this area is mainly on the client's developing environment, invocation of services, parallel execution, scheduling, and parameter handling, while the actual integration of data manipulated by the modules and the profile of the associated e-commerce applications are mostly ignored.

Data integration has been the focus of much recent research [11, 14, 12, 37, 32, 39, 21]. However, the emphasis was on *querying* the data available in Web applications. The application flow, when considered, is mostly limited to the navigation process needed for extracting dynamic web content (data extracted by filling multiple forms) [21]. Following these lines, many commercial Web sites, like comparative shopping services, that use the above technology, are not much more than search services; when shoppers wish to perform a purchase, they are directed to the merchant web sites (e.g., [18, 22]). Some sites do provide purchase capabilities (e.g., [49, 5]), but these are often based on pre-agreement with the merchants, using hard-coded interfaces or maintaining merchant catalogs, as opposed to our system which generates a full integrated Web application in a transparent way that does not necessarily demand the source site's cooperation. An exception is [7] that enables both product search and purchase in multiple sites, and which was released about the same time as our prototype. However, since no information is available regarding the internal structure of the product, one cannot tell whether the underlying technology is dedicated specifically to comparative shopping or provides a generic support for application composition.

Industrial standardization. As mentioned in the Introduction, there has been an intensive effort to develop XML standards for e-commerce. Some of these standards handle the protocols and the content of the data to be exchanged. This includes various efforts to move EDI vocabularies into XML (RosettaNet, X12/TG3 workgroup and UN/EDIFACT's Simple EDI) and the cXML [20] driven by Ariba which offers is a set of lightweight XML DTDs: a simple request/response protocol, definition of purchase order and purchase order response transactions, and the definitions for catalog data. However, each requires that all participating partners obey their unique single standard. There are also several competing efforts to develop frameworks that support multiple standards: BizTalk [10] driven by Microsoft focuses on an envelope to be used to describe XML contents, routing information and design guidelines for using XML. The eCo [17] framework driven by CommerceNet supports a discovery process of trading partners and the supporting description of processes, documents, and data elements. E'speak [28] driven by HP deals with the description of resources but not the contents of the transactions or the format of the data within it. Although the last three allow one to use a given set of different standards, they do not handle the integration of information nor do they support the integration of applications that do not conform to these standards. We will also mention several efforts, based on various standards, in the area of inter-enterprise workflows: eFlow [29] is a platform that supports the specification, development, and management of composite e-services. The platform provides a mechanism for defining services by composing basic services that conform to standards such as OTP, OBI, RosettaNet or e'speak. The CrossFlow [19] architecture, based on commercial workflow management systems, supports both contract making and services enactment by using service templates. The CMI [16] is a research effort aimed to expand and complement traditional workflow with a new collaboration technology that supports the coordination of activities and provides process-based application integration. These projects, like the frameworks described above, are also limited to integration of applications that conform to standards, and have no support for data integration.

Local-as-view vs global-as-view. The AM paradigm is an adaptation of the Information Manifold (IM) approach [37], used for data integration, to the context of e-commerce applications integration. In IM, local data is modeled as a traditional view of the global data. A significant difference here is that AM views cover the full profile of e-commerce applications, not just the data. As opposed to the IM paradigm for data integration, where local sources are modeled as views of the global schema, the TSIMMIS project [32] suggests an inverse paradigm, modeling the global data as a view (composition) of the local data. We have chosen the first approach for its suitability for modular application development, where the integration of new local applications into an existing system is rather convenient, requiring only the specification of the source view in terms of the global application and, when needed, the supply of an appropriate wrapper for the source. This is important in our context since new e-commerce applications are added to the Web every day and existing applications are often modified. Reducing the amount of work involved in incorporating new applications or adjusting to changes is thus an important factor.

It should also be noted that, in any case, no matter which specification direction is chosen (i.e., local as view or global as view), since the information flows in both ways (local data is integrated and passed to the global application, while updated global data and user input is sliced and distributed among the local applications), the inverse mapping will be needed as well. In our case this is generated automatically – AM introduces a novel rewriting paradigm, enhancing the traditional maximal view rewriting with a *refinement* mechanism dealing with the operational aspects of the application. In addition, for the data part, most of the previous work on maximal view rewriting has been in the context of relational data. Works dealing with more complex data models, like semi-structured and OODB

Application application_name

Actor actor-kind$_1$ **in application** application_name
data specification
methods specification
activities specification
application flow specification
...
Actor actor-kind$_n$ **in application** application_name
...

Fig. 1. Application specification

data, considered only exact query rewriting [8] (rather than maximal) and query containment [38]. A contribution here is the consideration of maximal view rewriting in the context of XML data, providing executable queries to combine data from several XML sources.

3 Preliminaries

Handling e-commerce applications from different sources requires a common framework in which the various applications and their data can be presented and modeled. As demonstrated in [1], a typical e-commerce application can be abstractly modeled by specifying the kinds of actors participating in the application and for each actor describing: (i) the data available for the actor and its access rights; (ii) the possible operations (methods) on the given data; (iii) the various activities that the actor may be engaged in, with the subset of data/operations relevant for each activity; and (iv) the application flow. To illustrate these components consider an example e-commerce application; an electronic bookstore. Typically such a store involves several types of *actors*, e.g, customers and vendors. It also involves a significant amount of *data*, e.g., the book catalog (typically searched by customers) or the promotion information (typically viewed by customers and updated by vendors). Observe that each of the actors may utilize different parts of the data (e.g., a customer can only see his/her own orders and the promotions relevant to his/her category, while vendors may view all the orders and promotions), each may have different *access rights* for the data (e.g., promotions can be updated only by certain vendors), and may perform different *operations*. Each actor typically performs several *activities*, e.g., a customer may be *searching* the catalog, *ordering* books, *changing* a passed order. Observe that in each of these activities, the actor may be facing a different Web page that possibly includes only part of the data and operations available to that given actor. Observe also that actions performed by an actor in a particular activity may initiate other activities/actions. For instance, when a customer orders a product, we may want to update the stock; when promotion is updated, we may want to refresh the customer's screen with the new data.

The form of the general specification of an application (both global and local) is illustrated in Fig 1.

The concrete syntax used for each part of the specification of global and local applications is given in the following sections. Readers familiar with [1] will notice that to some extent that our AM syntax resembles that of [1] – after all, both systems are used for constructing e-commerce applications. It is important to note however that the goal, hence semantics, here is different: in [1] specifications are used for constructing *new* e-commerce applications from *scratch*, with data and methods coming from a given database system (e.g., a new electronic bookstore with a catalog and customer information stored in a specific database). Hence the *data* part, for example, describes which part of the given database is viewed by each actor, and the *methods* part provides the code for the methods available for the actor. In contrast, AM specifications describe the integration and customization of *existing* Web applications (e.g., the integration of several existing electronic bookstores into one giant bookstore). Hence the data/methods/activities/flow parts of the specification describe how the data/methods/activities/flow of the integrated global application are realized in terms of the corresponding local ones. To prevent confusion it should also be noted that while both [1] and AM use the term *view*, the context and usage is completely different: while the first uses traditional *database views* for describing the portion of the database that is available for the actors participating in the application, AM uses *application views* for modeling the local applications as views of the global integrated application.

Before describing the syntax and semantics of the language, we give some minimal background on: (i) the *XML* standard used for the modeling of data; and (ii) *UML* and the *State Charts* formalism that will be used throughout the paper for the modeling of application flows. Then we present the architecture of the *Application Manifold* (AM) system which relies on the above.

3.1 Data model and query language

XML [45] is emerging as the new standard for data exchange on the Web. Since our goal is to support Internet applications such as e-commerce, we chose this emerging technology as the basis for our work. We will model the applications data using XML and use an XML query language for describing the data portion of the applications view. For lack of space, the presentation of XML here is rather brief, mostly via examples (that will also serve as a running example in the rest of the paper). Full definitions of XML and the XML query language can be found in [45,24].

An XML document features tags describing the logical structure of the document. To see an example, consider Fig. 2 which shows a fragment of an XML document describing a book catalog summarizing information from several Web bookstores. The <Category> and </Category> tags are used to delimit the information corresponding to one catalog category. Each category item consists of a sequence of tagged sub-items such as Category_Name, Book, etc.

An XML document can be typed. This is achieved by means of a Document Type Definition (DTD). Figure 3 shows a possible DTD for the document in Fig. 2. One can also define attributes for certain items (see, e.g., the Reviews item). In particular, we will use in the sequel the attribute name ACCESS to specify the allowed access rights for the given data items. (Here Reviews can be read or added but not deleted/modified.) For brevity we will assume that the default access right for an item is *read only*, and will explicitly

```
<Catalog>
  <Category>
    <Category_Name> Literature & Fiction </Category_Name>
    <Book>
      <ISBN>12445</ISBN> <Title>Bridget Jones's Diary</Title>
      <Authors><Author>Helen Fielding</Author></Authors>...
      <Stores>
        <Store> <Storename>BookStore 1</Storename> <Price>$10.39</Price> ...
                <Reviews> <Review> Screamingly funny - USA today </Review>
                          <Review> Unforgettably droll - Newsweek </Review> </Reviews> </Store>
        <Store> <Storename>E-BookStore</Storename> <Price>$12.95</Price> ...
                <Reviews> ... </Reviews> </Store>
      </Stores>
    </Book>
    <Book> ... </Book>
    ...
  </Category>
  <Category> ... </Category>
  ...
</Catalog>
```

Fig. 2. The global book catalog

```
⟨!ELEMENT   Catalog    (Category*) ⟩
⟨!ELEMENT   Category   (Category_Name,Book*) ⟩
⟨!ELEMENT   Book       (ISBN,Title,Authors,...,Stores)⟩
⟨!ELEMENT   Authors    (Author*)⟩
⟨!ELEMENT   Stores     (Store*)⟩
⟨!ELEMENT   Store      (Storename,Price,...,Reviews⟩
⟨!ELEMENT   Reviews    (Review*)⟩
⟨!ATTLIST   Reviews    ACCESS  (read,append)  #IMPLIED⟩
⟨!ELEMENT   ISBN       #PCDATA ⟩
...
⟨!ELEMENT   Review     #PCDATA ⟩
```

Fig. 3. The catalog DTD

specify the attribute only when different from the default. In general, typing is not a mandatory feature in XML, i.e., one can have documents, or document parts, without an associated DTD. However, since most optimization techniques rely on typing, it is realistic to assume that large XML applications will come with appropriate DTDs. In the sequel, we will denote element type definitions using the `Elem` suffix. For instance, `CatalogElem` will denote the type definition associated with the catalog element of the DTD given in Fig. 3.

So far, XML does not provide a standard query language. However, there is a major standardization effort in that direction [24,46,47]. Our goal here is not to propose a new language or to compete against the upcoming standard. Indeed, we intend to use this standard as soon as it becomes available. In the meantime, we will rely in this paper on the XML-QL language [24] to query XML documents.[1] For example, the query in Fig. 4 searches the global books catalog for books sold by "BookStore 1" and constructs the store's catalog: The `WHERE` clause describes the pattern to be searched for in the data. The relevant items are bound to the variables $X,$Y,$Z,$P,$Q. Then, for each such variable assignment, a book element with structure as is the CONSTRUCT clause is constructed. The nested query there basically "flattens" the Reviews item, obtaining book items with structure ⟨`!ELEMENT BookStore1_Book (ISBN,Title,Authors,Price,Category_Name,Review*)`⟩

XML-QL also supports regular path expressions in addition to constant labels. For this work we used only a fragment of the language, focusing on simple paths only. This simplifies the task of data integration (to be explained later) and seems sufficient for the applications that we considered. Another feature supported by XML-QL are Skolem functions in the CONSTRUCT clause, typically used for the grouping of several elements into one element. Since grouping can also be achieved using nested queries[24], we will assume below that our views use only nested queries.

3.2 Application flow

We use *State Charts* [33] to describe the application flow and to capture the fact that actions performed by an actor in a particular activity may initiate other activities/actions. [2] Besides providing a rich and flexible visual formalism where user activities can be modeled, State Charts support a *refinement* mechanism (to be described below) that provides convenient means for the modeling of the integration of applications. State Charts are also part of UML (Unified Modeling Language) [41], the emerging standard for the modeling of applications, hence, together with XML, provide a solid basis for our work.

We give below a brief overview of State Charts. For a detailed description see [33,34]. A state chart is a *higraph* (see [34]) consisting of rounded rectilinear blobs representing states, possibly nested and linked by transitions. A simple state chart is shown in Fig. 5. The state names, when significant, are written in the small rectangle attached to the top of the state (e.g., *Search_Activity, ShoppingCart_Activity, Search_Method*). The arrow emanating from the black dot signifies an initial state. A transition is denoted by an arrow and labeled by the name of the event causing the transition (e.g., *goto_Search_Activity, search, browse*). They can also have an associated *guarding condition* – a Boolean expression that evaluates to True or False. The transition occurs only if the guard is True. Guards are written within square brackets following the event name (e.g., *search*[*valid(Search_Params)*]). A state may have associated actions that are executed on entry to the state or upon exit from the state, e.g., when entering the

[1] Observe that [1] used the Lorel [3], rather than XML-QL, as query language. The change is not significant and is only for presentation convenience.

[2] In [1] active rules (triggers) were used instead. We have chosen here State Charts since they generalize the active rules in several ways and are more suitable for the modeling of applications integration.

```
<BookStore1_Catalog>
    WHERE <Catalog>
            <Category>
              <Category_Name> $C </>
              <Book>
                <ISBN> $X </> <Title> $Y </> <Authors> $Z </>
                <Stores>
                   <Store>
                     <Storename>"BookStore 1"</> <Price> $Q </> <Reviews> $P </>
                   </> </> </> </> </>
          IN "GlobalBookStore/catalog.xml"
    CONSTRUCT
        <BookStore1_Book>
          <ISBN> $X </> <Title> $Y </> <Authors> $Z </> <Price> $Q </> <Category_Name> $C </>
          WHERE <Review> $R </> in $P
          CONSTRUCT <Review> $R </>
      </>
</>
```

Fig. 4. XML-QL Query

Search_Method state, the method $search()$ is executed; then the state is exited, and the event *goto_ShoppingCart_Activity* is signaled (causing transition to the *ShoppingCart_Activity* state). Parallel execution of activities can be described using *orthogonal* states, graphically separated by dashed lines (see, e.g., Fig. 22, to be discussed in detail later on). State Charts also support a *refinement* mechanism that allows simple non-composed states (e.g., *Search_Method*) to be associated with a detailed state chart describing the actual implementation of the component, hence providing a modular description applications.

The UML State Charts semantics [41] that we use here assumes that a state machine described by a state chart reacts to events applied to it by some external objects. Event processing by the state machine is partitioned into steps, each of which is caused by an event directed to the state machine. The fundamental semantics assumes that events are processed in sequence, where each event stimulates a run-to-completion (RTC) step. The next external event is dispatched to the state machine after the previous step is completed. Once an event instance is dispatched, it may result in one or multiple transitions being enabled for firing. By default, if no transition is enabled, the event is discarded without any effect.

3.3 The application manifold architecture

The AM system is based on a client/server architecture. An AM application consists of several independent clients communicating with the AM server and possibly between them (via the server) using an XML-based API. Figure 6 shows the various components of one application (obviously, several such applications may run simultaneously on the server).

As mentioned above, an AM application consists of: (i) a (virtual) global application with which the user interacts; and (ii) the (actual) local applications implementing it. The task of the AM is to accept a request from clients of the first kind and process them by communicating with applications of the second kind. The communication with the underlying Web applications is done via wrappers that translate the standard AM API requests to application-specific *http* calls (and in the opposite direction mapping the obtained data to the corresponding AM XML-based representation).

The AM consists of five main components, sketched below.

The *request parser* receives user requests (messages in XML format), parses them and passes the constructed request objects to the *request rewriter*. This consults the specification of the given application, and rewrites the request into a corresponding state chart, expressing the request in terms of the local applications. The *validation module* verifies the correctness of the construction (e.g., the capabilities of the various sources and the adequacy of the passed parameters). The state chart is then passed to the *application manifold engine* for execution. The engine is in charge of communicating with the various sources, passing and receiving data and requests, and maintaining information about the current state of each application. The *results integrator* provides data integration services for the data accumulated in the processing. The engine is also in charge of communicating with the user, passing request results and relevant notifications. The request rewriter and the application manifold engine use an XML repository (omitted from the figure): the AM application specifications are stored in the repository and the request rewriter consults it for selecting the components relevant to the given request. Similarly, the repository is used by the AM engine to store and manipulate the data and state information accumulated in the processing.

4 Example

We will illustrate things throughout the paper using a simple running example – a global book store which integrates services offered by several bookstores on the Web. Note that although similar to some of the applications available on the Web for comparative book shopping (e.g., [22,49]), our emphasis here is different: our goal is not to present yet another comparative shopping application but rather to demonstrate the core technology behind the AM system, which enables a declarative specification and generation of arbitrary integrated/customized e-commerce applications (with comparative shopping being just one possible instance). We present below the application specification. The semantics of such a specification is then explained in the following section. Our AM specification will consist of two parts, the first describing the global bookstore application with which the user interacts and the second describing the (relevant parts of the) actual underlying local bookstore applications and their relationship to the global one. We start with the specification of the global application.

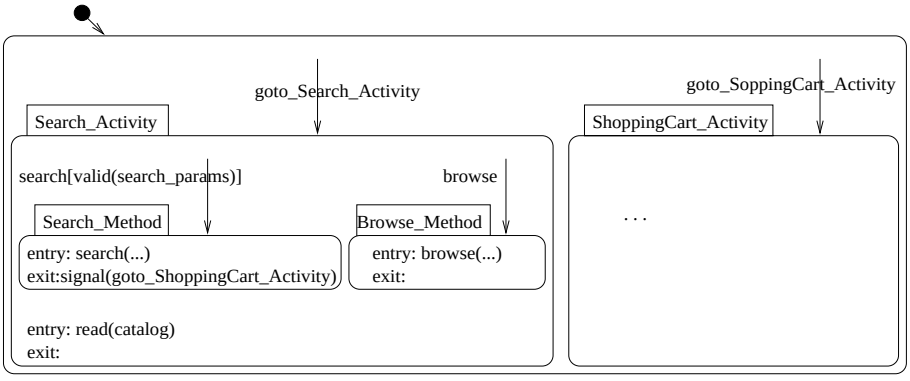

Fig. 5. Example of state chart

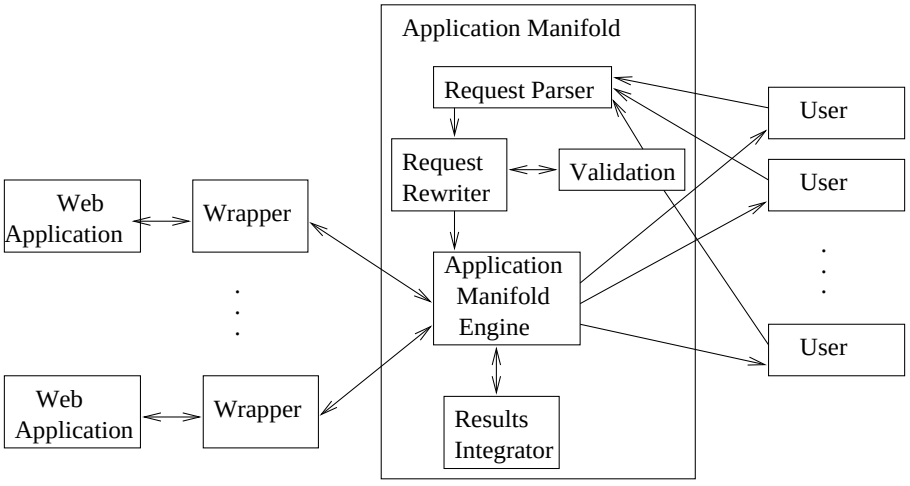

Fig. 6. Application manifold architecture

4.1 The global bookstore

We define for each actor type participating in the global application: (i) the data available for the actor; (ii) the possible operations on the data; (iii) the activities that the actor can be engaged in; and (iv) the application flow. We detail below the global *Customer* specification. Other actors (e.g., suppliers, publishers) can be defined similarly.

Data and methods. Assume that the data available to a customer includes: (i) a catalog with a structure as in Fig. 3 which can be browsed and searched using the methods *browse* and *search*, respectively; (ii) the search parameters (used as input to the search method); (iii) the search result; (iv) a shopping cart where items can be added or deleted; and (v) a list of the customer pending orders with the methods *order* to place a new order for the items accumulated in the shopping cart and *status* to check the delivery status of a pending order. The data and operations specification for the customer actor is partially given in Fig. 7. All the data variables except the second one are *common*, in the sense that their values correspond to (possibly integrated) values originating in the underlying local applications (e.g., the catalog presented to the user will in fact be the integration of catalog data from the local bookstores. Similarly the global shopping cart is the union of the customer's carts in the various stores.) In contrast, *search_params* is *private* in the sense that its value is supplied by the user (and may be passed as input to the underlying applications). Similarly, the *browse*, *search*, and *order* methods here are defined as

common	catalog : CatalogElem
private	search_params : SearchParamsElem
common	search_result: (BookElem)*
common	shopping_cart : (ItemElem)*
common	orders : (ItemElem)*
...	
common method	search(search_params,search_result)
common method	browse(...)
common method	order(...)
...	

Fig. 7. Data and methods of the global customer

common: their actual code is not specified in the ActiveView; instead they will be implemented by communicating with the underlying applications, issuing (possibly a sequence of) local methods. In addition to the methods listed explicitly in the specification, we also have read/write/append/... methods for the various data elements, according to the access rights specified in the element DTD.

Activities and flow. Assume that a customer can be engaged in four activities: (i) login into the system; (ii) searching the catalogue; (iii) adding/removal of books from the shopping cart; and (iv) placing an order. From an end-user viewpoint, each activity corresponds to a Web page with some data and buttons. For instance, the *Search_Activity* page will show the catalog and the search criteria (to be filled in by the user), and some buttons allowing the user to call the search and

activity	Search_Activity	**includes**
	catalog, search_params, search_result	
	search(), browse(), goto_ShoppingCart_Activity(), goto_Order_Activity(), quit()	
activity	ShoppingCart_Activity	**includes**
	search_result, shopping_cart	
	add_to_cart(), remove_from_cart(), goto_Order_Activity(), quit()	

Fig. 8. Activities of the global customer

browse methods, change activity or quit the application. The *ShoppingCart_Activity* page will show the search result and the shopping cart, and some buttons allowing the customer to add/remove elements from the cart or switch to the ordering activity. This is specified as in Fig. 8.

Finally, the application flow is defined using a state chart: the default state chart associated with an actor has one state per activity, entered when *goto_ActivityName* signal is signaled. Upon entry the values of the activity variables are read. Each activity has substates representing the activity methods. They are entered when the relevant method button is pressed, and the relevant method is executed upon entry. One can customize this default flow by adding components or refining the flow. For example, Fig. 5 shows part of the default customer state chart (for brevity some of the methods and activities are omitted), customized by adding a guarding condition to the search method (verifying before execution that the search parameters are valid). In addition, once the search is executed, the customer is automatically switched to the shopping cart activity to be able to add some of the selected items.

4.2 Local bookstores

Local applications are defined as views over the global application. Then, at execution time, the user requests and flow of the global application will be rewritten to actions in terms of the local applications, (possibly with some additional global processing).

Assume we are given several local applications which we will call in the sequel BookStore1,..., BookStoreN. We detail below BookStore1 (a somewhat simplified version of the Barnes&Nobel Web site). The other applications (corresponding, for example, to Amazon.com and so on) are treated similarly. Upon entry to Bookstore1, the user can search the catalog by one of three criteria: title, author, keyword. To activate other types of searches, the user clicks on an 'advanced search' button getting a Web page where he/she chooses (again, by clicking on the relevant buttons) between two possible Web pages: the first allowing a search by ISBN, and the second allowing a search by one or more of the criteria, author title, and keyword, optionally restricted by price, format, age, and subject. The various searches return a search result including only a short description of the retrieved books. To get more information on a specific book, users need to click on a 'read more' button next to the book, switching to a page containing the full book data, where they can click an "add to shopping cart" button, adding the book to their cart.

We next define this local application as a view over the global application. It should be noted that *only the relevant parts of the local application need to be specified*: the application may feature other data/activities that are not covered by the global application; these need not be specified.

let	bookstore1_catalog: BookStore1CatalogElem
be	Query1
let	quick_search_params: QSearchParamsElem
be	Query2
let	ISBN_search_params: ISBNSearchParamsElem
be	...
let	full_search_params: FSearchParamsElem
be	...
...	

Fig. 9. Variable definitions in the local application

```
<QSearchParams>
   WHERE <SearchParams> <$P> $Q</> </>
         IN "GlobalBookStore/SearchParams.xml"
         $P in {Title,Author,Keyword}
   CONSTRUCT  <Criteria>
                 <$P> $Q </>
              </>
</>
```

Fig. 10. Query2

Data. The local customer data here includes, among others, the catalog, the three types of search criteria, the search result, and the shopping cart. The structure and access rights are defined by a local DTD (omitted here) and may differ from that of the corresponding variables in the global application. The variables value is defined as a view (query) over the global data, basically projecting out the data originating from the given application. This is illustrated in Fig. 9 above, where Query1 is the query of Fig. 4, retrieving the BookStore1 portion of the global catalog, and Query2 is the query in Fig. 10, retrieving the 'quick search' parameters (title,author,keyword), if they exist, from the global search criteria. In general each local variable is defined as a query over one variable of the global application. Recall that we distinguished here between *common* and *private* variables (e.g., *catalog* vs *search_params*). Following the same terminology we will refer to the local variables defined by queries over common(private) variables as common(private).

In addition to the above data variables, which correspond to variables of the global application, the local application may use some additional local variables: Web applications often maintain information about the user's state/context in cookies and variables that are passed to the user's browser and sent back to the application server with each user request. To continue with our example, assume, for example, that BookStore1 also contains (as in the original Barnes&Noble application) the local variables *userid: UidElem* and *log:LogElem* (here we do not have associated queries).

Methods, activities, and flow. Once the local data and its relationship to the global data is specified, we continue with the methods, activities, and flow. The specification of a local application (just like that of the global one) lists the applica-

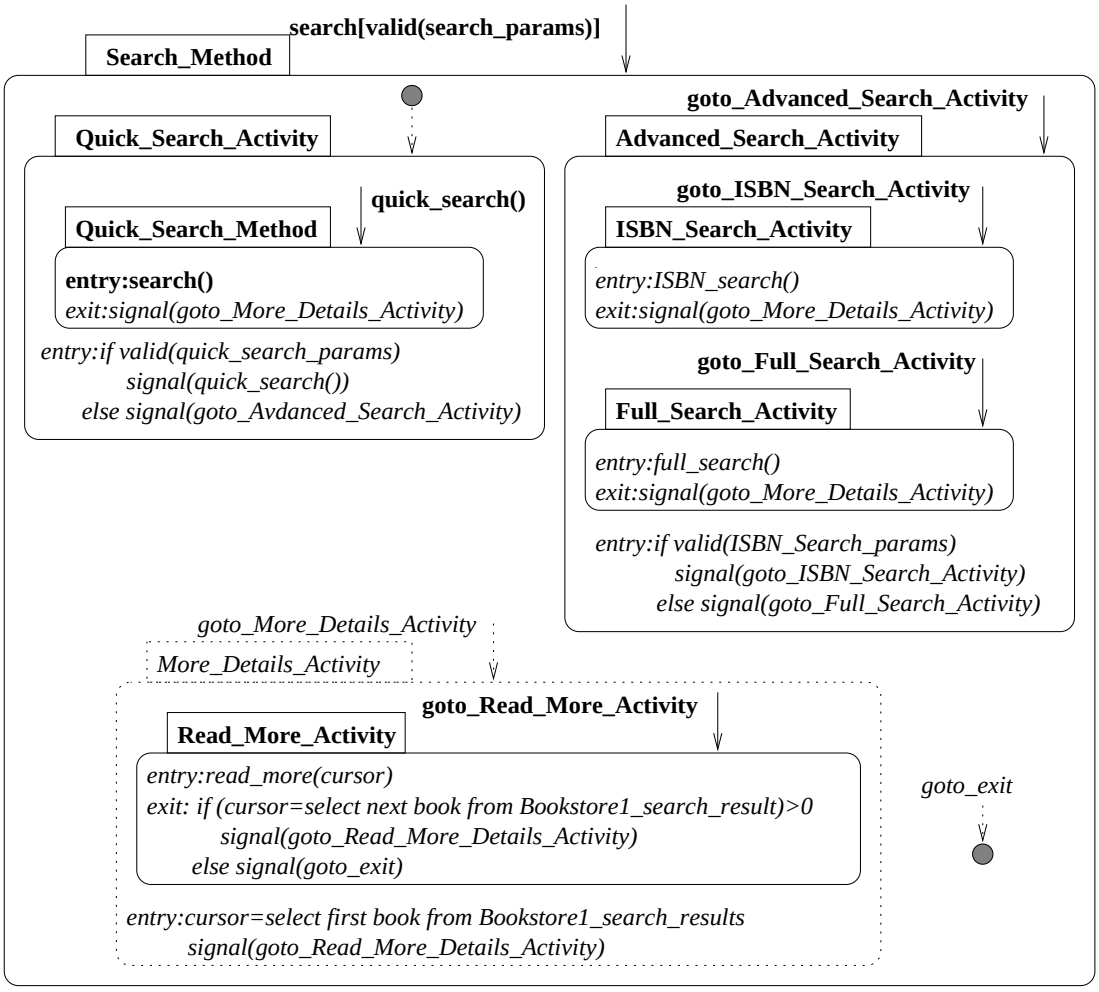

Fig. 11. Refinement for search_method

tion's methods, activities and flow, in a similar syntax. (As in the case of data, only the methods/activities/flow relevant to the global application need to be specified.) The relationship between the global and the local ones is specified via the State Charts *refinement* mechanism: first, with each global activity A we associate some local activity $l(A)$. Now, recall that each of A's methods was represented in the state chart of the global applications by a (simple) state A_m. These states are now refined to include the relevant subset of the local application's state chart, augmented with some additional control describing the specific flow required for capturing the global method, and having $l(A)$ as the initial and final states.

Example 1. To illustrate this consider, for example, the *search* method of the global *Search_Activity* in Fig. 5. Figure 11 above shows a possible refinement of state *Search_Method*, with *l(Search_Activity) = Quick_Search_Activity*. The bold letters and solid lines of the refinement describe the original relevant portion of BookStore1's state chart, while the dotted lines and italic letters describe the additional specific control implementing the local version of the global method. Recall from the beginning of the subsection that BookStore1 has three possible search activities (quick, ISBN, and detailed), with the user starting at the quick search screen and, if desired, moving to the other two types. In addition, recall that the three corresponding search parameters were defined above as views (queries) over the global search parameter. The added control here basically validates the value of the three possible local search parameters (i.e., the result of the corresponding view queries) with respect to their DTDs, and executes the first applicable local search. Now, since the search in BookStore1 returns only a short description of the retrieved books, and since further details can be obtained only for one book at a time, the added control also includes a new state (*More_Details_Activity*), besides the original BookStore1 states, realizing a 'read more' iteration over the retrieved books.

Upon completion we go back to the initial state, ready to execute additional activity methods.

Example 2. To see a less 'query-oriented' example, consider the *ShoppingCart_Activity* and a method *add_to_cart* that adds a list of books to the cart. Since in BookStore1 only one book at a time can be added (and this can be done only once the book was searched for and its full details requested and displayed), the refinement will include an iteration (similar to the one above) over the sub-list relevant to BookStore1 (defined as explained above as a view over the global list), with each loop realizing a sequence of BookStore1 activities: retrieving the given book (e.g., via an ISBN search), obtaining its full details, and then adding it to the actual (local) BookStore1 cart (see Fig. 12).

Example 3. Note that while the examples above relate local and global actors playing essentially the same role (customer in both cases), this is not obligatory: one can use the same mechanism to relate global and local data/activities of arbitrary types of actors. For example, in our global application a customer may add reviews to the catalog. If in BookStore1 only certain vendors are allowed to modify the reviews, the customer's global 'add reviews' method can be associated with the relevant local vendor's activity, and refined by a state chart describing the necessary flow to be performed on behalf of the vendor. Furthermore, one can realize a global method by a combination of activities involving several types of local actors.

Example 4. Our framework can also be used to *enhance* exiting applications with new services. For example, assume that to assist users in deciding where to buy their books we want to: (i) give the users a 'cheapest bargains' summary of the search result; and (ii) record for each user his/her order history, allowing him/her to annotate the list with comments regarding the (dis)satisfaction of the given services. Now our global application has *real* coded methods in addition to those defined by refinement (e.g., a method *cheapest_books*, calling *search* and then querying the obtained *search_result* to find the cheapest stores), and *real* data stored in the system's XML repository, in addition to the virtual data coming from the underlying applications. Variables in the global application can now be defined as views over this repository data, e.g.,

```
let   customer_order_history: AnotatedOrderElem*
be    WHERE <AnotatedOrd><Usid> $U </></> ELEMENT_AS $X
            IN XML_repository/OrdHist.xml
          $U =self.id
      CONSTRUCT <AnotatedOrd> $X </>
```

User requests for real data/methods are then served by the XML repository, while the virtual ones are delegated by the AM engine to the underlying Web applications.

Remark. One may wonder how difficult is it to model local application and define the above refinements. It is important to note that specifications do not require any knowledge of underlying implementation of the local applications, but only a rather simple analysis of their Web interface: Web pages or frames are naturally modeled as activities, page data and form fields as data variables, and buttons as methods. The flow in the refinement is then essentially a description of a typical usage of the Web interface. In the next two sections we explain the semantics of the specification. Then we give a detailed example of the modeling process and the runtime operation in Sect. 6.

5 How things work

Now that we have illustrated how AM applications are specified, we need to explain the semantics of such a specification. As shown above, the specification of the local applications consists essentially of two parts, the first modeling the local data variables as views (queries) over the global variables, and the second modeling the local application flow as a refinement of the global one. We will start by considering the more standard part – the data variables. Next we will consider the flow and explain how the two parts work together.

5.1 Data

Applications contain two types of variables, *private* and *common*. The value of private variables (e.g., *search_params*) in the global application is supplied by the user. The value of the corresponding variables in the local applications (e.g., *quick_search_params*, *ISBN_search_params*, *full_search_params*) is computed by simply evaluating the queries associated with the variables.

For common variables (e.g., *catalog*), the computation flows in the other direction: their values originate from the corresponding variables in the local applications (e.g., *bookstore1_catalog* in Bookstore1). Then, following the Information Manifold approach [37], the value of the corresponding variable in the global application is defined to be the *maximal view rewriting* for the variable, given the views in the local applications (which basically means that the global variable contains as much relevant information as can be obtained from the corresponding local variables.)

Before explaining this further, let us first recall the standard definition of *maximal view rewriting* from the relational data model [26]. Given a relation R, some views $V_1, ..., V_n$ over R, and a query Q over R, the maximal view rewriting for Q is a query Q' that: (i) uses only the views as EDB relations; (ii) is contained in Q; and (iii) contains any other query Q'' that satisfies (i) and (ii).

Intuitively, Q' computes as much as possible of the answer to Q based on the information given in the views. To see the connection to our XML context observe that: (i) XML documents are often modeled as edge-labeled graphs (with nodes representing the document elements, edges representing the "component of" relationship among elements, and with the edges labeled by the type of the pointed item, or with a data value, for the atomic leaf elements) [3,24]; and (ii) such graphs can be described by an *edge* relation $R(from, label, to)$, with queries being modeled as mappings of such input edge relations to output ones.[3] Now, let u be some common variable in the global application (with corresponding edge relation R_u) and let $u_1, \ldots, u_n$ be the corresponding source variables in the local applications, described as views over u (with edge relations $R_{u_1}, \ldots, R_{u_n}$). The value of u can be defined to be the maximal view rewriting of R_u using $R_{u_1}, \ldots, R_{u_n}$.

Note however that we are not interested here in constructing arbitrary maximal XML documents for our global variables: first, we are primarily interested in *valid* documents – recalling that each variable has an associated DTD, we want the obtained documents to be valid with respect to this DTD. Second, we may be interested in documents that obey some additional structural restrictions – for example, in our global book catalog, we want to have a unique category entry for each category name. (i.e., no two $\langle Category \rangle$ elements can have the same $\langle Category\_name \rangle$ value). Consequently, what we really want is a rewriting that is maximal with respect to documents satisfying the constraints.

Again, to understand this, lets go back to the relational model, where maximal view rewriting has been studied for databases obeying constraints in the form of functional depen-

[3] The order among sibling elements can be captured by an additional order relation. To simplify the presentation we ignore this here.

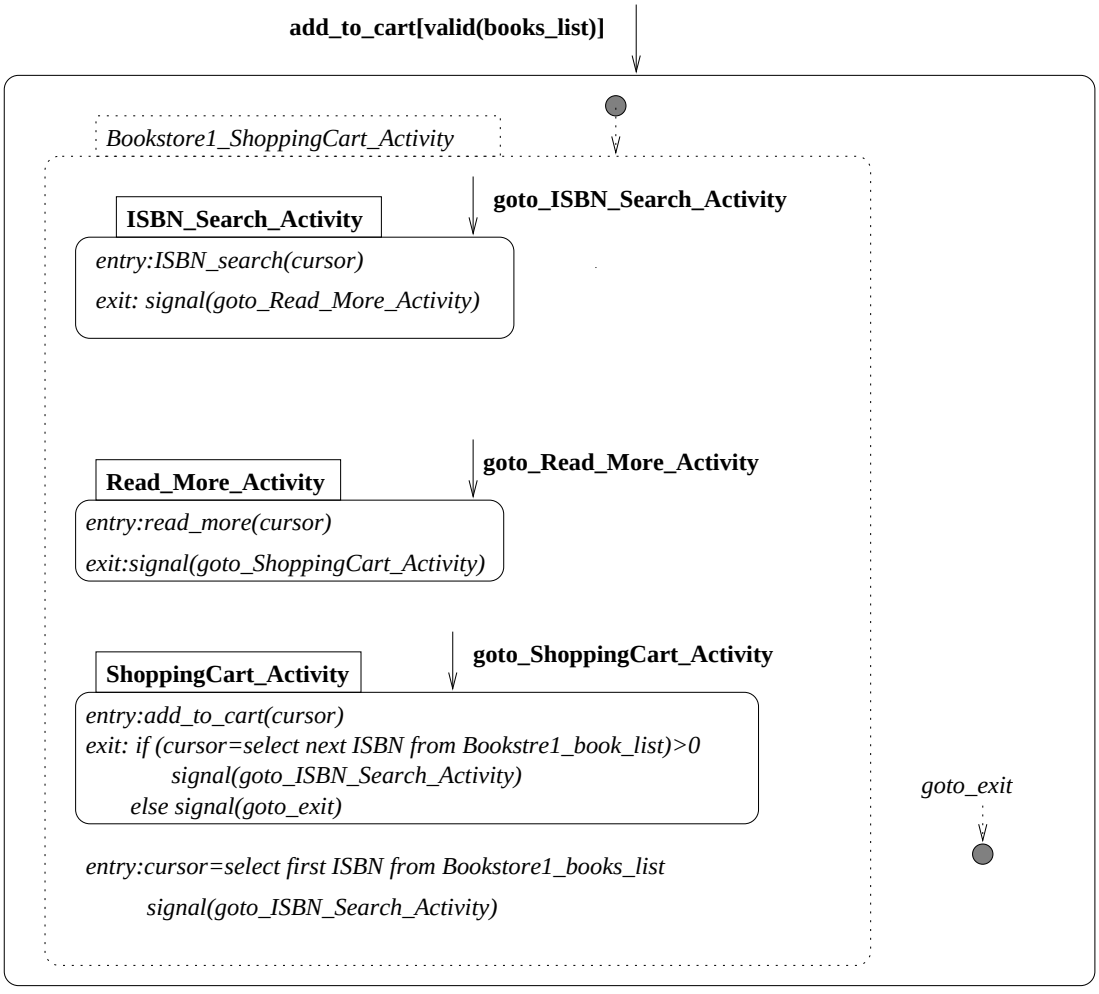

Fig. 12. Refinement for add_to_cart_method

```
ALGORITHM   Global variable computation
INPUT       AM specification, set of functional dependencies, local variables u_1,...,u_n
OUTPUT      global variable u

 Step 1 For each variable u_i defined by a query Q_i, construct an inverse query Q'_i.
 Step 2 Annotate items in Q'_i with Skolem functions to capture the functional dependencies.
 Step 3 For i = 1...n, compute the annotated Q'_i on u_i.
 Step 4 Merge the results to obtain a canonical value for u.
```

Fig. 13. Global variable computation

dencies [27]. To capture this, the definition of maximal view rewriting was adjusted so that query containment is tested only with respect to databases obeying the given dependencies. Interestingly, many of the constraints we are interested in here can be viewed as functional dependencies between document elements. For example, a DTD requirement, say, that a $\langle Book \rangle$ element contains one $\langle ISBN \rangle$ sub-element, amounts to the fact that the identity of the ISBN element in the document graph functionally depends on the id of its parent book item. Similarly, in our catalog, the identity of the $\langle Category \rangle$ node functionally depends on the value of its $\langle Category\_Name \rangle$ child.

Modeling our document constraints as functional dependencies allows us to harness the techniques from the relational world for computing our global variables. We will focus here on the above two types of functional dependencies, the first modeling a DTD requirement for a given element to contain only a single occurrence of a sub-component of a certain type (hence the identity of the subcomponents functionally depends on the id of its parent), and the second, modeling cases where the id of an element functionally depends on the id of some of its subcomponents. Figure 13 describes the algorithm for computing the value of a global variable. We explain it below and illustrate it with an example.[4]

Step 1: inverse queries. It was observed in [26] that, in the case of relational conjunctive views and queries, the maximal view rewriting can be obtained by generating for each view some *inverse rules* computing the portion of the global relation that is captured in the view. The key observation is that since we use here only a fragment of XML-QL including, in particular, only simple paths, our views are essentially conjunctive queries over the edge relation, and, hence, similar inversion principles as in [26] apply. Adapting this to our

[4] Clearly not all constraints imposed by a DTD can be modeled by such functional dependencies. The problem of maximal view rewriting that fully captures document validity is an open problem, see Sect. 2.

```
WHERE <BookStore1_Catalog>
        <BookStore1_Book>
           <ISBN> $X </> <Title> $Y </> <Authors> $Z </> <Price> $Q </> <Category_Name> $C$ </>
        </> CONTENT_AS $P'
      </> IN "BookStore1/bookstore1_catalog.xml"

CONSTRUCT <Catalog ID=S_Catalog()>
            <Category ID= S_Category($C)>
              <Category_Name ID=S_Category_Name(S_Category($C))> $C </>
              <Book ID=S_Book($X)>
                <ISBN ID=S_ISBN(S_Book($X))> $X </>
                <Title ID=S_Title(S_Book($X))> $Y </>
                <Authors ID=S_Authors(S_Book($X))> $Z </>
                <Stores ID=S_Stores(S_Book($X))>
                  <Store ID=S_Store(S_Stores(S_Book($X)),''BookStore1")>
                    <Storename ID=S_Storename(S_Store(S_Stores(S_Book($X)),''BookStore1"))>
                                                                    "BookStore 1"</>
                    <Price ID=S_Price(S_Store(S_Stores(S_Book($X)),''BookStore1"))> $Q </>
                    <Reviews ID=S_Reviews(S_Store(S_Stores(S_Book($X)),''BookStore1"))>
                      WHERE <Review> $R </> in $P'
                      CONSTRUCT <Review> $R </>
                    </> </> </> </> </> </>
```

Fig. 14. The inverse query for Query1 (appearing in Fig. 4)

```
<BookStore1_Catalog>
     ...
     <BookStore1_Book>
        <ISBN> 0735710201 </>
        <Title> Inside XML (Inside) </>
        <Authors> Steven Holzner </>
        <Price> $39.99 </>
        <Category_Name> Computers </>
        <review> Great Comprehensive Guide to XML, November 27, 2000 </>
     </>
     <BookStore1_Book>
        <ISBN> 1861003412 </>
        <Title> Beginning XML </>
        <Authors> Kurt Cagle, et al </>
        <Price> $31.99 </>
        <Category_Name> Web Development </>
        <review> This book is definitely very good for beginners. </>
     </>
     ...
</>
```

Fig. 15. BookStore1's catalog variable

context we generate for each local variable u_i defined by a query Q_i an *inverse query* Q_i' by basically switching the roles of the WHERE and CONSTRUCT clause of the query. There are two delicate points here.

- First, the WHERE clause may contain some variable names that do not appear in the CONSTRUCT. (This is analogous to the case where the body of a conjunctive query contains some variables no appearing in the head.) As in the case of conjunctive queries, when the query is inverted and the roles of the WHERE and the CONSTRUCT clauses are switched, those variables are simply replaced in the new CONSTRUCT clause by a corresponding Skolem function of the remaining variables[26].
- Second, we need to handle nested queries. Recall from Sect. 3.1 that a nested query is essentially composed of two parts. A variable in the WHERE clause that defines the scope of the query (e.g., the variable $\$P$ in the query in Fig. 4), and nested query over this variable in the CONSTRUCT clause. When inverting the query we also invert the role of these two parts: the nested query in the old CONSTRUCT clause (which now becomes a WHERE clause) is replaced by a new variable name whose scope is the item where the nested query appeared, and the (inverted) nested query is moved to the new CONSTRUCT (previously the WHERE), replacing the old scope variable.

To continue with our example, the inverse of Query1 from Fig. 4 is given in Fig. 14 above. Ignore for now the ID attribute of the elements. (this will be explained below). The WHERE and the CONSTRUCT clauses of Query1 are switched, and the nested query (with its WHERE and the CONSTRUCT clauses also switched) is moved to the (new) CONSTRUCT clause, being switched with its source variable $\$P$.

Step 2: Skolem functions. To capture the restrictions imposed by the given functional dependencies, we associate with each restricted element $\langle Ename \rangle$ in the CONSTRUCT clause a Skolem function S_{Ename} determining the element ID as a function of the elements on which it depends. Again, this is an adaptation of the relational treatment for functional dependencies of [27], equating elements that functionally depend on the same values: Skolem functions are used in XML-QL to control how the result is produced and grouped. In Fig. 14, whenever a $\langle Category \rangle$ is produced, its associated ID is $S\_Category(\$C)$. $S\_Category(\$C)$ is a Skolem function, and its purpose is to generate a new identifier for every distinct values of $\$C$. If, at a later time, the query binds $\$C$ to the same value again (e.g., by finding another book of the same category), then the query will not create another $\langle Category \rangle$ element, but instead will append information to the existing $\langle Category \rangle$. Thus, all the $\langle Book \rangle$s of that category will be grouped under the same $\langle Category \rangle$ element. The $\langle Category\_Name \rangle$ element has a single occurrence since its id depends on the category id. Note that in this example the only non-constraint elements are the $\langle Review \rangle$s. They thus

```
<BookStore2_Catalog>
    WHERE <Catalog>
            <Category>
              <Category_Name> $C </>
              <Book>
                <ISBN> $X </> <Title> $Y </> <Authors> $Z </>
                <Stores>
                   <Store>
                     <Storename>"BookStore 2"</> <Price> $Q </>
                   </> </> </> </> </>
          IN "GlobalBookStore/catalog.xml"
    CONSTRUCT
        <BookStore2_Book>
          <ISBN> $X </> <Title> $Y </> <Authors> $Z </> <Price> $Q </> <Category_Name> $C </>
        </>
</>
```

Fig. 16. Query 2: BookStore2's catalog as a query over the global catalog

```
WHERE <BookStore2_Catalog>
        <BookStore2_Book>
           <ISBN> $X </> <Title> $Y </> <Authors> $Z </> <Price> $Q </> <Category_Name> $C$ </>
        </>
      </> IN "BookStore2/bookstore2_catalog.xml"

CONSTRUCT <Catalog ID=S_Catalog()>
            <Category ID= S_Category($C)>
              <Category_Name ID=S_Category_Name(S_Category($C))> $C </>
              <Book ID=S_Book($X)>
                <ISBN ID=S_ISBN(S_Book($X))> $X </>
                <Title ID=S_Title(S_Book($X))> $Y </>
                <Authors ID=S_Authors(S_Book($X))> $Z </>
                <Stores ID=S_Stores(S_Book($X))>
                  <Store ID=S_Store(S_Stores(S_Book($X)),''BookStore2")>
                    <Storename ID=S_Storename(S_Store(S_Stores(S_Book($X)),''BookStore2"))>
                                                                    ''BookStore 2"</>
                    <Price ID=S_Price(S_Store(S_Stores(S_Book($X)),''BookStore2"))> $Q </>
                    </> </> </> </> </> </>
```

Fig. 17. The inverse query for Query2

have no associated Skolem function and a new $\langle Review \rangle$ is produced for each distinct assignment for the query variables.

Steps 3 and 4: variable computation. Now, given the inverse queries for all the variables in the local applications, we construct a single canonical value for the global variable u. This is achieved by evaluating the inverse queries for all the u_i's and "fusing" all the elements with the same id into a single element. For instance, in the above example, for each book, all the catalog data coming from the various local applications will be grouped under the same $\langle Book \rangle$ element (since the id of the book element is defined to be a Skolem function of the book's ISBN, which is the same in all sources).

Example of data integration. To illustrate the actual data integration process, assume we have two book stores, BookStore1 and BookStore2, each with its own books catalog, and consider their integration into a global catalog. As before, assume that the relationship between the global catalog variable and that of BookStore1 is defined using Query1 (appearing in Fig. 4), and let Fig. 15 describe part of the actual catalog content of BookStore1.

Next, consider the catalog variable of BookStore2. Naturally, it may have a different structure than that of the first book store, so assume for example that it does not contain book reviews. The relationship between the global catalog variable and that of BookStore2 is defined using the query in Fig. 16. As we did for BookStore1, we define a corresponding inverse query for the variable (see Fig. 17). Notice that as this source contains no Review items, the query (and thus its inverse too) is not nested. Finally, Fig. 18 describes part of the actual content of BookStore2's catalog.

To conclude the example we need to see how the two catalogs are integrated, namely, execute steps 3 and 4 of the algorithm. We first apply the two inverse queries on the corresponding local catalog variables. The results are given in Figs. 19 and 20, respectively. Notice that the Skolem-based ids assigned to the items are important even before the results are integrated: for instance, the output of each of the two queries contain only one *Catalog* element rather than the two that would be generated if no Skolem functions where used. Similarly, the result of the second inverse query (Fig. 20) contains a single *Category* element with the name *Computers* rather than the two that would be created in the absence of Skolem functions.

Finally, we run the forth step of the algorithm and merge the results of the queries into an integrated global catalog by fusing elements with identical id's. The result is described in Fig. 21, where, for clarity, the identifiers are omitted from the integrated catalog, (as they are not really part of the data and are only used for grouping and fusion.) Intuitively, the integrated catalog is the maximal one could obtain given the sources – it contains all the available books information without redundancies.

Remark on consistency. The above algorithm assumes the consistency of the data sources with respect to the given set of functional dependencies. In practice, although each source may individually obey the constraints, (e.g., books' ISBNs determine their title), they may not be preserved globally due to errors or different writing standards, (e.g., a book with the same ISBN may have distinct titles in different stores due to spelling mistakes or shorthand notations.) Fusion (in step 4) of such inconsistent data may produce multiple values for an item, rather than the single one required by DTD. When this is the case, we present the alternative values to the user and let her/him chose. Automatic resolution of inconsistencies[42], based, for example, on linguistics analysis and source reliability, is planned to be incorporated in the future.

```
<BookStore2_Catalog>
     ...
     <BookStore2_Book>
        <ISBN> 0735710201  </>
        <Title> Inside XML (inside)</>
        <Authors> Steven Holzner </>
        <Price> $34.99 </>
        <Category_Name> Computers </>
     </>
     <BookStore2_Book>
        <ISBN> 0672320541 </>
        <Title> Applied XML Solutions</>
        <Authors> Benoit Marchal </>
        <Price> $35.99 </>
        <Category_Name> Computers </>
     </>
     ...
</>
```

Fig. 18. BookStore2's catalog variable

Remark on optimization. The current default in the system is that the global variables are fully computed when a *read* for the variable is requested (see below). For certain applications this may be redundant: consider for example the global book catalog. When browsing the catalog, the user will typically visit only a few entries and then will stop either because the needed data is found or because he/she decides to call the search method for issuing a particular search. In this case it is better to view the above variable value definition as virtual and employ lazy evaluation [8], computing the relevant elements only as the user navigates into the data. To overrule the default, one can use two specific kinds of read modes, namely, *deferred read* or *immediate read*. The first instructs the system to compute elements only on demand, while the latter indicates that elements should be computed immediately when encountered.

5.2 Activities and application flow

Recall from Sect. 3.2 that the activities and flow of each actor in the global application are modeled by a state chart, with user requests interpreted as events in the corresponding state machine. Now, at execution time, the State Charts refinement mechanism is employed: on each user request (method call), rather than activating the relevant global state, its local refinements are executed. Observe, however, that typically each global method (state) may have several such refinements, one per local application. In the computation, the multiple refinements are viewed as orthogonal states and performed in parallel.

To continue with our example, consider the *search* method of the global bookstore (modeled by the simple state *Search_Method* in Fig. 5). A method call results in running the compound state machine in Fig. 22 which includes n orthogonal states representing the refinements of *Search_Method* for the local applications BookStore1,...,BookStoreN. Note that, in general, some methods may not be supported by *all* the local applications. For example, BookStore1 may support a catalog search but not an arbitrary browsing of its catalog (hence its specification may contain no refinement for the *Browse_Method* of BookStore1). The constructed state chart contains only the applicable refinements.

To see the connection with the AM system architecture, recall from Sect. 3.3 that the *Request Rewriter* module of the AM is responsible for constructing the state chart portion to be executed upon user request. The state chart being built for each method call is precisely the refinement described above, with each orthogonal state corresponding to one local application. In principal, rather than running all local applications in parallel, the request rewriter could decide, for optimization reasons, to serialize the requests to certain web sites as a result of validation checks or optimization considerations. For example, a serialization policy may be adopted to eliminate redundant execution if the necessary information is already fetched from other sites. While the current implementation is fully parallel and does not support this type of optimization, we plan to further investigate the issue and add this in the next version of the system.

Once constructed, the refined state chart is passed to the AM engine for execution. Recall that in each such refinement, part of the state chart describes actual activities of the local applications (the solid lines and bold text in Fig. 11), while part describes the added control needed for implementing the global method (the dotted lines and italic text in Fig. 11). When the AM engine runs the state chart, the bold-text parts are executed by communicating with the local applications, while the control in the italic-text parts is executed by the AM engine itself.

In between consecutive user requests, the AM system maintains the *application context*, for both the global and local applications. This includes the application state and the current value of all variables for all the actors in the application. The variables and states are augmented with expiration time and stored in the XML repository. When a user issues a new request within a reasonable time to the given application, the relevant state machines resume the states they last had at any depth within the composite states and all variables values are restored. Then the request state machine is run starting from this point.

5.3 All together

To conclude, we explain how the the flow part and the data part work together. Recall from the previous subsection that we distinguish between two types of variables, *private* and *common*, the first originating in the global application, while the second originates from the local ones. When a private variable needs to be passed to a local application, its value is computed (as explained above), from the corresponding global variable, and vice versa, when a read request for a common variable is issued by the global application, its value is computed from the corresponding local variables. The data in the global application is obtained by check-in/check-out, e.g., explicit calls to the global *read* and *write* methods. Thus, in general changes are not immediately propagated. (However, the freshness level of data can be controlled by the programmer, specifying in the DTD the desired read/write frequency.)

Read: observe that the data in local applications may be updated from outside the AM system, e.g., via its direct Web interface. Thus, each global *read* request is refined to a fresh local read of the corresponding local variables. Then the system computes the value. There are two possible optimizations that we plan to incorporate in the next version of the system: (i) some Web applications (e.g., [1]) support a *notification*

```
<Catalog ID=S_Catalog()>
     <Category ID=S_Category(‘‘Computers")</>
         <Category_Name ID=S_Category_Name(S_Category(‘‘Computers"))> Computers </>
         <Book ID=S_Book(0735710201)>
             <ISBN ID=S_ISBN(S_Book(0735710201))>0735710201 </>
             <Title ID=S_Title(S_Book(0735710201))>Inside XML (inside)</>
             <Authors ID=S_Authors(S_Book(0735710201))>Steven Holzner</>
             <Stores ID=S_Stores(S_Book(0735710201))>
              <Store ID=S_Store(S_Stores(S_Book(0735710201)),‘‘BookStore1")>
               <Storename ID=S_Storename(S_Store(S_Stores(S_Book(0735710201)),‘‘BookStore1"))>
                                                              ‘‘BookStore1"</>
                <Price ID=S_Price(S_Store(S_Stores(S_Book(0735710201)),‘‘BookStore1"))> $39.99 </>
                <Reviews ID=S_Reviews(S_Store(S_Stores(S_Book(0735710201)),‘‘BookStore1"))>
                  <Review>Great Comprehensive Guide to XML, November 27, 2000</>
         </> </></>
     <Category ID=S_Category(‘‘Web Development")</>
         <Category_Name ID=S_Category_Name(S_Category(‘‘Web Development"))> Web Development </>
         <Book ID=S_Book(1861003412)>
             <ISBN ID=S_ISBN(S_Book(1861003412))>1861003412 </>
             <Title ID=S_Title(S_Book(1861003412))>Beginning XML </>
             <Authors ID=S_Authors(S_Book(1861003412))>Kurt Cagle, et al</>
             <Stores ID=S_Stores(S_Book(1861003412))>
              <Store ID=S_Store(S_Stores(S_Book(1861003412)),‘‘BookStore1")>
               <Storename ID=S_Storename(S_Store(S_Stores(S_Book(1861003412)),‘‘BookStore1"))>
                                                              ‘‘BookStore1"</>
                <Price ID=S_Price(S_Store(S_Stores(S_Book(1861003412)),‘‘BookStore1"))> $31.99  </>
                <Reviews ID=S_Reviews(S_Store(S_Stores(S_Book(1861003412)),‘‘BookStore1"))>
                  <Review>This book is definitely very good for beginners.</>
</> </> </> </></>
```

Fig. 19. The result of applying the first inverse query on BookStore1's catalog variable

```
<Catalog ID=S_Catalog()>
     <Category ID=S_Category(‘‘Computers")</>
        <Category_Name ID=S_Category_Name(S_Category(‘‘Computers"))> Computers </>
        <Book ID=S_Book(0735710201)>
          <ISBN ID=S_ISBN(S_Book(0735710201))>0735710201 </>
          <Title ID=S_Title(S_Book(0735710201))>Inside XML (inside)</>
          <Authors ID=S_Authors(S_Book(0735710201))>Steven Holzner</>
          <Stores ID=S_Stores(S_Book(0735710201))>
             <Store ID=S_Store(S_Stores(S_Book(0735710201)),‘‘BookStore2")>
               <Storename ID=S_Storename(S_Store(S_Stores(S_Book(0735710201)),‘‘BookStore2"))>
                                                              "BookStore2"</>
               <Price ID=S_Price(S_Store(S_Stores(S_Book(0735710201)),‘‘BookStore2"))> $34.99 </>

        </> </>
        <Book ID=S_Book(0672320541)>
          <ISBN ID=S_ISBN(S_Book(0672320541))>1861003412 </>
          <Title ID=S_Title(S_Book(0672320541))>Applied XML Solutions</>
          <Authors ID=S_Authors(S_Book(0672320541))>Benoit Marchal</>
          <Stores ID=S_Stores(S_Book(0672320541))>
             <Store ID=S_Store(S_Stores(S_Book(0672320541)),‘‘BookStore2")>
               <Storename ID=S_Storename(S_Store(S_Stores(S_Book(0672320541)),‘‘BookStore2"))>
                                                              ‘‘BookStore2"</>
               <Price ID=S_Price(S_Store(S_Stores(S_Book(0672320541)),‘‘BookStore2"))> $35.99 </>
</> </> </> </>
```

Fig. 20. The result of applying the second inverse query on BookStore2's catalog variable

mechanism which allows notifying the AM engine on changes to the local data. Recording such notifications will allow the AM engine to avoid redundant reads; (ii) rather then fully recomputing each time the global value, we can compute the delta between the new and old local values (using e.g., a diff algorithm as in [44]) and only propagate the change. Recall from Sect. 5.1 that we construct an inverse query for each local variable and compute the value of global variables by merging the results of those queries. This allows us to view the global variable as a view of the local data (although the original specification was reversed – local variable were defined as view of global ones!). Thus, the problem at hand is that of propagating updates from the database (the local data) to the view (the global variable), which can be performed in the style of [2].

Write: when a variable in the global application is modified and a *write* is requested, we have to propagate the change from the global application to the local ones. The local data is defined as a view over the global data. By default, a global write is refined to a local write with the new value for the view. Again, this can be optimized. If the system maintains the delta between the old and the new value for the global variable, an incremental evaluation of the views can be performed in the style of [2]. Besides computational efficiency, incremental evaluation has two additional advantages here:

1. Since we are in a client server architecture, sending the delta instead of the entire value may result in large saving in communication.
2. The second issue is access rights. Consider for example the variable *catalog* in the *Customer* global application, and assume that some reviews are appended to a given book and a *write* is issued for the catalog. Assume that the access rights of $BookStore1\_catalog$, as declared in the

```
<Catalog>
    <Category>
      <Category_Name> Computers </>
      <Book>
         <ISBN> 0735710201 </>
         <Title> Inside XML (Inside) </>
         <Authors> Steven Holzner </>
         <Stores>
           <Store>
               <Storename> BookStore1 </>
               <Price> $39.99 </>
               <review> Great Comprehensive Guide to XML, November 27, 2000 </>
           </>
           <Store>
               <Storename> BookStore2 </>
               <Price> $34.99 </>
           </>
         </>
      </>
      <Book>
         <ISBN> 0672320541 </>
         <Title> Applied XML Solutions</>
         <Authors> Benoit Marchal </>
         <Stores>
           <Store>
               <Storename> BookStore2 </>
               <Price> $35.99 </>
           </>
         </>
      </>
    </>
    <Category>
      <Category_Name> Web Development </>
      <Book>
         <ISBN> 1861003412 </>
         <Title> Beginning XML </>
         <Authors> Kurt Cagle, et al </>
         <Stores>
           <Store>
               <Storename> BookStore1 </>
               <Price> $31.99 </>
               <review> This book is definitely very good for beginners. </>
           </>
         </>
      </>
    </>
</>
```

Fig. 21. An excerpt of the global catalog after the integration)

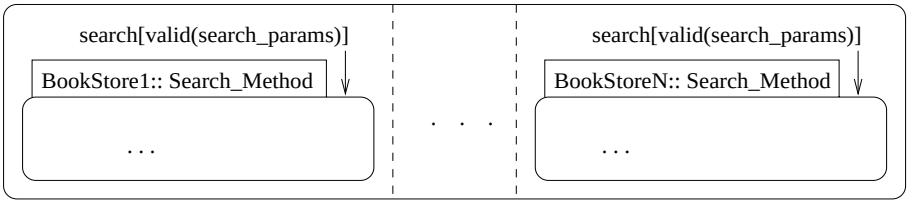

Fig. 22. Compound refinement of search_method

DTD, is *read* for all elements but the $\langle Reviews \rangle$ which can also be updated. An attempt to propagate the write by replacing the whole $\langle BookStore1\_Catalog \rangle$ element by a new one will be rejected by the local application due to access rights violation, whereas an incremental update replacing only the $\langle Reviews \rangle$ will be accepted.

Updates that cannot be propagated due to violation of access rights in the local applications are rejected by the system. Another difficulty is updates that are not relevant to any of the local applications. To see an example, assume that our application allows certain users to update all the catalog data. Such a user could in principle add book elements sold by a $\langle Storename \rangle$ other than any of the given local applications. However, now when a *read* is issued for the global catalog, a recomputation of its value from the local applications will cause the update to disappear. This could be anticipated by rejecting non-propagative updates and is planned to be incorporated in the next version of the system.

Remark. We conclude this section with a remark on transactions. The transactional support that a global application can give depends on the capabilities of the underlying local applications. By default, a global application is not in a transaction mode: even if one assumes that method calls in local applications are atomic, the local refinement of a global method may consist of a sequence of such calls. Transactional behavior can be achieved only if the local applications support explicit methods to start a transaction and terminate it with an abort or commit, with the level of global atomicity depending on the level of the available local support [4].

6 Implementation

The AM system is implemented as a mediator system that simultaneously plays two roles: a server for the global applications, accepting and handling user/application requests, and a client of the local applications, imitating a browser when connecting to remote services. [5] In this configuration we can easily implement a broker for different applications. We are currently experimenting with a broker for bookstores and are next planning to implement a broker for auctions.

To keep the application simple and open and to enable portability we mostly use public domain tools. In particular, we run the *Apache Web Server* [6] with *mod_perl* [6] on a *Windows NT* machine (these tools were also tested on a Linux machine and are also applicable to different Unix machines). The implementation work consists of three parts:

(1) Implementation of the AM system.
(2) Writing the specifications of the global and local applications.
(3) Writing of wrappers.

We will describe each of these below.

6.1 The AM system

As explained in Sect. 3.3, the system consists of five modules (see the center part of Fig. 6). They are implemented in Perl, C++, and Java in a Windows environment, and activated by a CGI script upon a user request. The communication with the users and the local applications is managed by the Apache Web server and proxy server, respectively. Recall that communication with the local applications may require the activation of an appropriate wrapper. For that we have replaced one of the modules of the proxy server with a mod_perl script of our own, in charge of wrappers activation.

Request parser. Implemented in Perl. Parses the AM XML-based API (global) request and, using the stored applications specification, attaches the list of relevant local sources. The resulting object is passed to the validation module.

Validation module. The first version of AM implements a simplified version of the validation module which includes only DTD validation checks for each local application. (Implemented using the Xerces Java parser [xml.apache.org]). In the next version we plan to add consistency checks (e.g., validity of request according to application state). Valid requests are passed to the request rewriter, otherwise a user notification is issued.

Request rewriter and AM engine. The two are combined together into one executable. The current version of the request rewriter implements parallel activation of requests to all relevant sites. In the next version we plan to experiment with different optimization strategies (e.g., serialization of requests, adaptive request's cost management). The AM engine consists of a main program, which generates events according to the accepted request, and a collection of modules which include the local application's state machines and are activated by those events. These modules are automatically generated from the specification, as described below, using Rational's *Rose RealTime* tool [43] and include local state charts definition as well as state machine activation code. The AM engine communicates with each local application (using threads) via the wrappers and writes the response to a file (a distinct XML file for each local source.) These files are passed to the result integrator.

Results integrator. Written in Java and uses a DOM interface to read the XML local data. The integrator reads and parses each file (for validity check and DOM tree creation), thus creating one tree per file, and then integrates them into one tree (using the algorithm of Sect. 5.1).

6.2 Specification and code generation

For the specification of applications we use a UML compliant modeling tool – Rational's *Rose RealTime* tool – which provides a convenient graphical interface for writing the specifications and automatic code generation from the specifications (plus debugging and tracing facilities for the code).[6] The generated modules realize the state machines described in the specifications and are used by the AM engine as explained above.

Modeling considerations and structures. In Sect. 3 we described the use of State Charts to specify the activities, the methods, and the flow. Recall that activities corresponds to web pages while methods represents the operations that can be activated using buttons within these pages. In our specification language, we have used states to specify both methods and activities. As different activities (pages) can be activated in parallel, they are described as orthogonal states according to [33] (see Fig. 5). Since our specification language follows the UML standard, the system implementation could be simplified by using a UML compliant tool for the specification and code generation. There are several such tools on the market (e.g., [35,43]). We have chosen to use *Rose RealTime* as our implementation tool. This specific choice forced the implementation to deviate a bit from the specification language introduced in previous sections, to match the tool's capabilities. Nevertheless, the changes are very minor and affect only the graphical syntax of the specification and not the semantics. We detail the changes below. *Rose RealTime* implementation of orthogonal states is based on the notion of capsule, which are a UML stereotype of a class. Therefore we use capsules to describe the activities, and their associated state chart to describe the methods and the flow. Figure 23 demonstrates the visual notation of capsules. Capsules are represented by rectangulars, while the small black or white squares are ports which are used by capsules for a signal-based communication (events transmission). Due to the use of capsules, the refinement process, used for specifying the local application, differs

[5] Other alternatives could be implementing it, for example, as a plug-in for a user browser or as a software agent. We chose the above architecture for modularity reasons, but similar design principles could apply for the other configurations.

[6] This in fact illustrates one of the main advantages of basing our framework on standards like XML and UML – the ability to capitalize on enhanced tools developed for the standard.

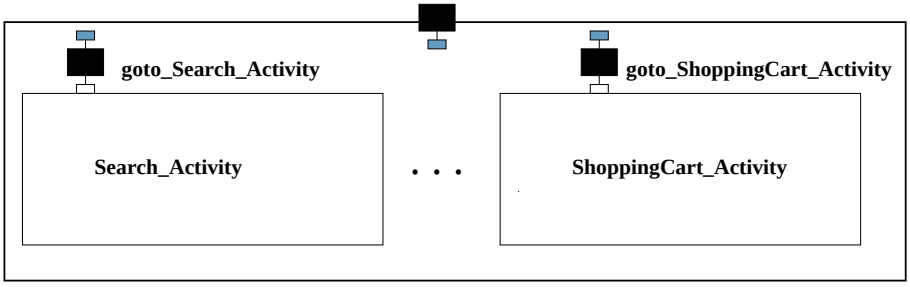

Fig. 23. Global application (with capsules)

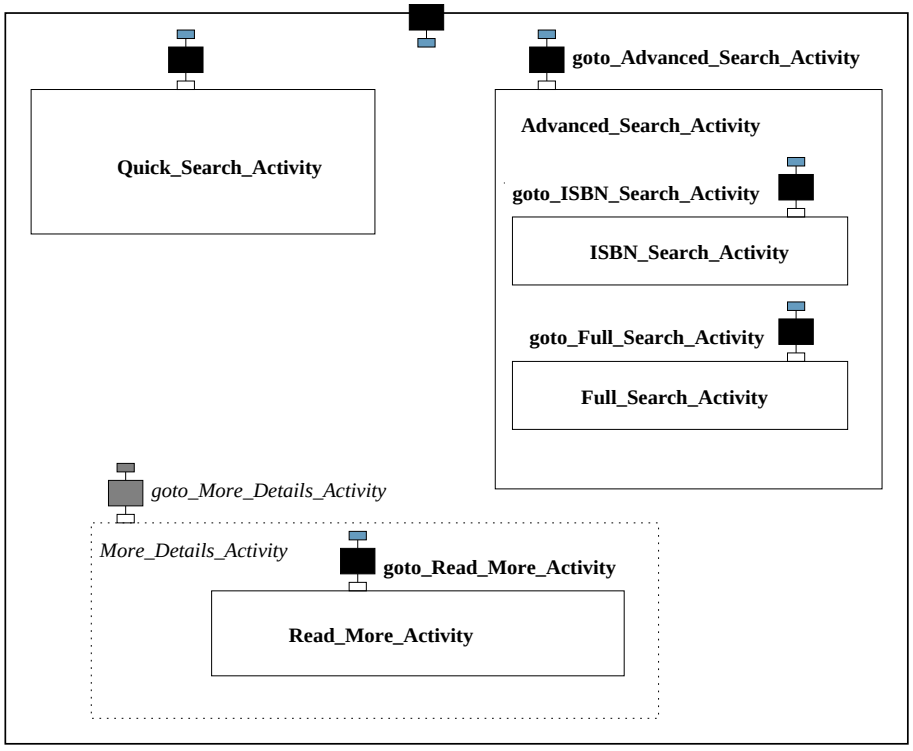

Fig. 24. Local application (with capsules)

a little from the one described in Paragraph 4.2. First, local activities (capsules) are derived from global activities (capsules) by inheritance association. In this way, the local activity inherits the state chart (and sub-capsules) of the corresponding global activity. From this point, we continue with the state chart refinement steps as described above.

Modeling example. Now we return to our running example of a global bookstore which integrates services offered by several bookstores on the Web, and follow the modeling steps:

Modeling the global application. First we define the global data structure (see Fig. 3 for the dtd), then we define the global application interface (Fig. 25) which actually defines the API. Recall that forms and buttons correspond to data and methods. We create four pages. For each page we define an AM activity: Search, SearchResults, ShoppingCart, and AddReviews. The search activity enables us to submit a query to multiple bookstores (Search_Method) and browse the integrated catalog (Browse_Method). (goto_BookPrices). On search activation, a new page appears, containing the search results. Each book description includes title, author,ISBN, list of the bookstores it was found in, and price and availability details. The methods in this page are Add_to_Cart_method and goto_Add_Reviews. The next page shows the shopping cart's content in all relevant bookstores and allows to delete_from_cart, change_quantity and initialize all carts. The last page (omitted from the figure) includes an add_review method. Figure 23 is a partial UML representation of this interface in *Rose RealTime*.

Modeling local applications. Now we come to model the local applications. The UML modeling of the application flow and method is quite straightforward and derived from the local site interface. Once the global application is defined, it is used as a base class for defining of the local applications by inheritance association. Then the specific behavior is defined by refinement of the global state charts. We also separate the handling of the communication from the modeling process. Calls to local sites are implemented in one separate general module, with one public method which accept the local sites parameters (requested site, methods, and data to be passed) and supports issuing http calls and handling of cookies. In that way the code involved in activating a local site's method within a state is minimal and merely includes calling the above method.

Figure 26 is a screenshot of three pages of a local bookstore. The specification of the according state chart through refinement was explained in detail in Sect. 4 and illustrated in

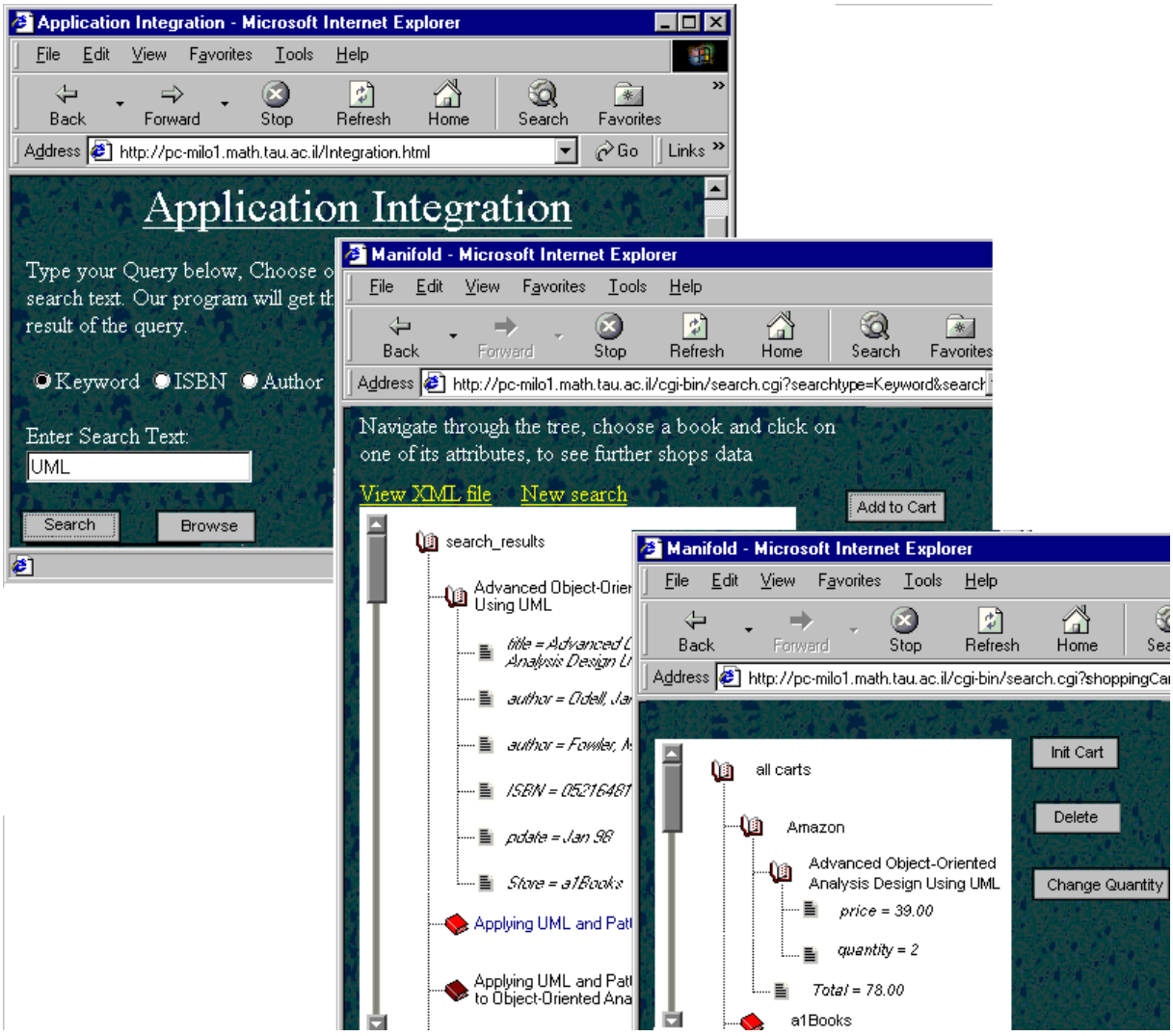

Fig. 25. The global application interface

Fig. 11. Figure 24 shows the equivalent UML model in *Rose RealTime*, using capsules. The definition of the dtd is quite simple. We can easily derive from the visual representation of the pages the dtd for Amazon's local catalog. Each book item includes title, one or more authors, list price, Amazon's price, availability etc. However, sometimes there is additional hidden information. For example, from looking at the html page source of the search results we can extract the ISBN of each book, which is a unique identifier for books and is important for data integration. After we have the local dtd's, we define the mapping between them and the global data as queries over the global data (see Figs. 4–10).

Code generation. Finally, after all this is complete, we can build the local bookstore module. We do this by creating a component which contains the local bookstore capsules, selecting the component in the browser, and clicking compile. *Rose RealTime* then creates the make file based upon the platform, generates the C/C++ code for the application (based upon structure, state transition diagrams, and transition and state action logic), and initiates the C++ compiler to generate the code. Now that we have compiled, we can run the application from *Rose RealTime*, set breakpoints, animate the RoseRT model so that we can watch how the application is running. Then, we deploy the dll file to the target machine to be activated by the AM Engine to communicate with the local applications.

6.3 Wrappers

A wrapper has two parts, the first in charge of the translation of the standard AM API requests to application-specific *http* calls, and the second in charge of mapping the returned data to the corresponding AM XML-based representation. The first direction can be implemented in two ways: one is rather straightforward and amounts to running an application Web browser in the background, mapping each API request to a corresponding action on the browser (typing or buttons activation). This is naturally facilitated by the fact that the local applications modeling is tight to the actual screens, hence the mapping is immediate. The second alternative is an actual formatting of corresponding http calls (with the appropriated parameters). While the first approach is applicable in most cases,

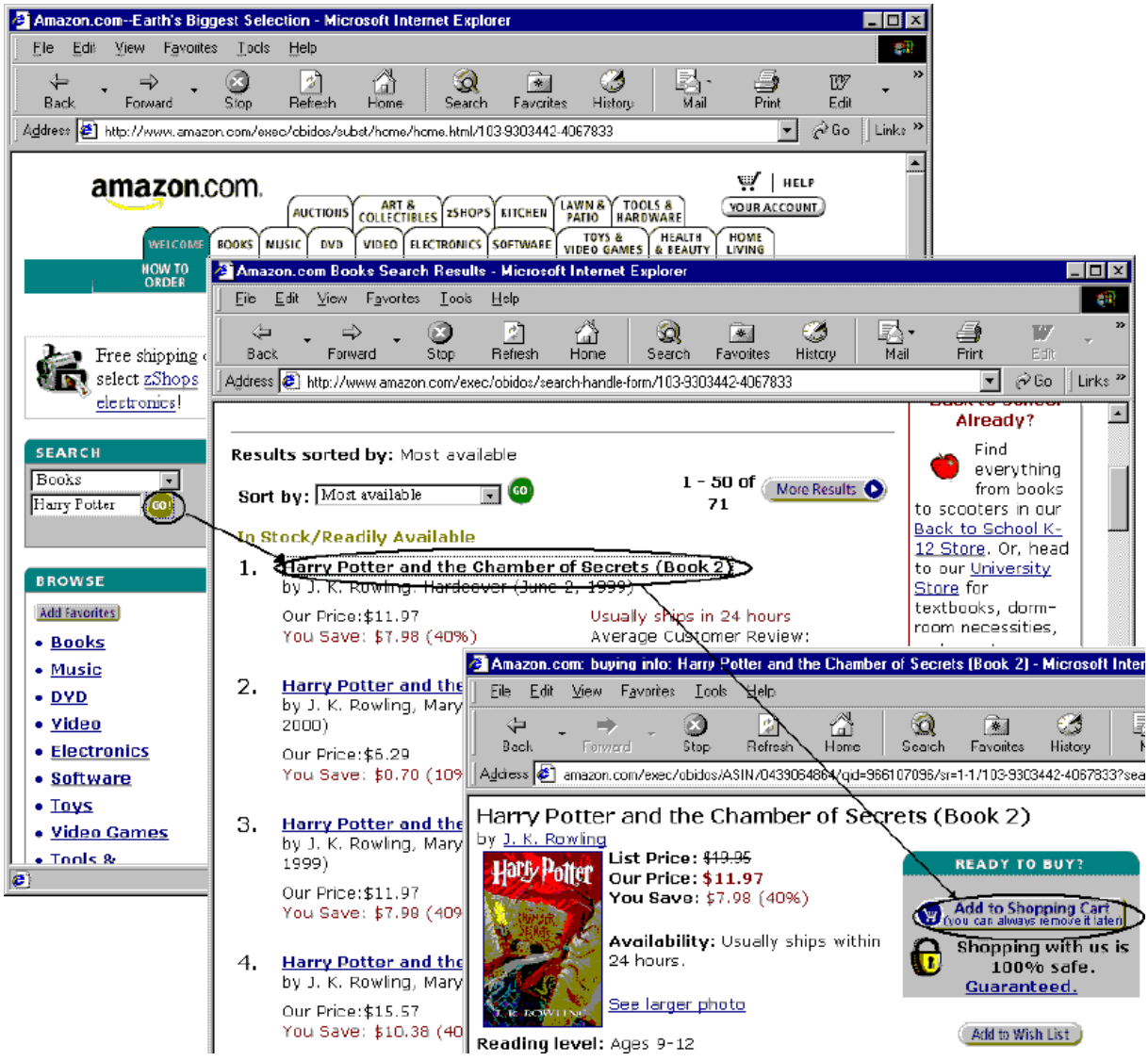

Fig. 26. Local application example

```
<entry>
      <site>www.amazon.com>
      <method>search</method>
      <action>POST</action>
      <uri>www.amazon.com/exec/obidos/search-handle-form/!session_id!</uri>
      <content>index=books&field-keywords=!search_str!</content>
</entry>

<entry>
      <site>a1books</site>
      <method>add_to_cart</method>
      <action>GET</action>
      <uri>www.a1books.com/cgi-bin/a1books/a1Front?act=addCart&WVSESSION_ID=
          !WVSESSION_ID!&ISBN=!ISBN!</uri>
</entry>
```

Fig. 27. Wrapper entry for search method

the latter is possible only for http-based requests (e.g., not for Java rmi calls), and is feasible only when the correspondence between the API variables and the http call parameters is clear. Nevertheless, we decided to use the latter in our implementation: a proxy tracing of the http calls of the local bookstores showed a straightforward correspondence between the parameters, hence, writing the request translator was quite trivial. We plan to implement the former method in the next version of the system to support a wider range of applications. The request transformer is table-based Perl script which reformats the general request according to the requested local site and method. In this version the table is stored as an XML file and will be managed in an XML repository in next versions. For each method in each local bookstore we need to add the an entry to this table. Figure 27 is an excerpt of this table. Notice that each entry defines the http request method (POST/GET), URL

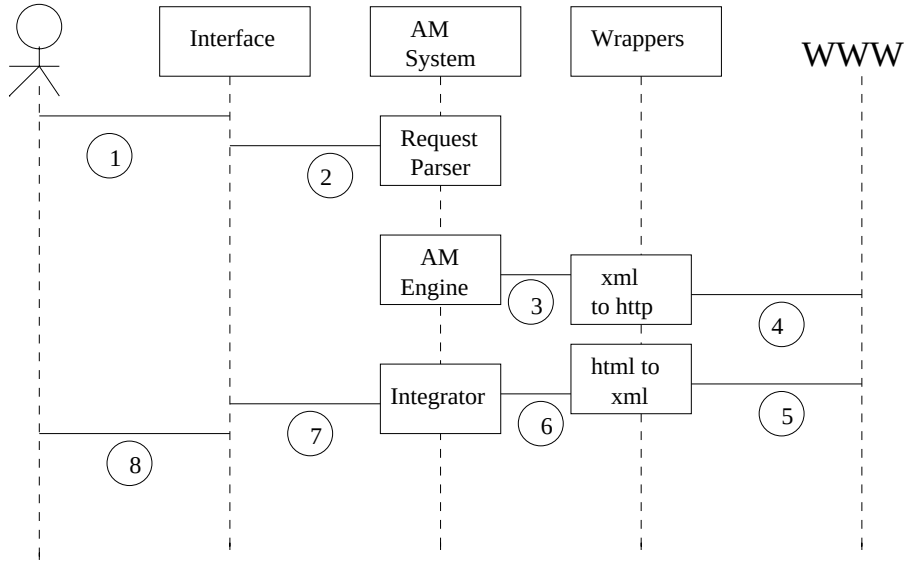

Fig. 28. Request processing

```
<request>
        <site>*</site>
        <method>search</method>
        <params>
               <criteria>author</criteria>
               <search_str>Rowling</search_str>
       </params>
</request>
```

Fig. 29. Search request

and the content to be send in case of a POST action. Strings between exclamation marks are names of variables should be extracted from the original XML request and replaced by their actual values.

The opposite direction, i.e., mapping html response to XML-based representation, requires parsing of the response. For that we used simple scripts written in Perl that parse a specific page content and perform the above translation.

During the implementation we encountered the problem of web-page structure being periodically changed. We thus consider adopting methods for change detection and automatic wrappers development [36].

6.4 Runtime operation

Now, after the specifications are defined and the appropriate wrappers are built, we will describe what happens during the runtime operation. Figure 28 illustrates the steps that a user request follows: (1) The user activates one of the global application methods (e.g., search) by clicking on one of the global application interface buttons; (2) the interface, via the user's browser, issues a request to the AM system (recall that the AM system is implemented as a cgi program and runs under a web server). For example, Fig. 29 demonstrates a request for a book search to all sites that support such methods. The AM parses the request, decomposes it to several requests (e.g., one for each bookstore), validates and rewrites them. Each request is then handled (in a different thread) by the appropriate local store's module as part of the AM engine. These modules send one request or a sequence of requests, according to the local application flow, via the proxy server (3), which is in charge of wrappers activation. These requests are a site-specific request in a unified format and contain the local application's cookies and variables maintained by the AM engine. The proxy server activates a wrapper, which transforms the request according to an XML file (Fig. 27) to the specific format of the local application (see Fig. 30) and sends it (4). Then the proxy server captures the response (5), parses it to the right XML format using an appropriate wrapper and returns it to the sender (6). The AM integrator integrates all responses, creates a unified XML response, and sends it back via the web server (7) to the user (8).

7 Conclusion and future work

As the number and variety of e-commerce applications on the Web grow, the need for convenient tools to support the integration and customization of existing applications increases. This paper presents the *Application Manifold* system which, based on the emerging Web standards XML and UML, offers a novel solution for the problem. The system supports a declarative specification language for specifying the integration and customization task, covering the full profile of the integrated/customized e-commerce applications: the various services offered by the applications, the activities and roles of the different actors participating in the application, the application flow, as well as the data involved in the process. Then, acting as an application generator, the system generates a full integrated/customized e-commerce application.

While the work described in his paper was targeted at supporting the integration and customization of e-commerce applications, the developed solution can also be used for other types of Web applications with similar characteristics, including in particular: (i) sharing of data; and (ii) cooperative work by a number of actors connected via the network. These are typical features also found, for instance, in digital libraries or manufacturing information systems.

The declarativity of the specification allows for much freedom in the optimization the generated application. Only little

```
POST http://www.a1books.com/cgi-bin/a1books/a1Search HTTP/1.0
User-Agent: AM browser
Host: www.a1books.com
Content-Length: 83
Proxy-Connection: Keep-Alive

act=search&WVSESSION_ID=30509&matchCriteria=keyword&searchBy=author&searchStr=Rowling
```

Fig. 30. Local request

of that was incorporated in the first prototype of the system. We are currently working on extending the prototype along the lines mentioned in Sect. 5, supporting optimized reads and writes and conditional/serial execution of the refined state machines.

We are currently examining the ability of the system to scale both in terms of the number of applications that can simultaneously be integrated and the number of users that can be supported. We started with a few applications due to computation power limitations. Keeping track of several web applications may be easy, but doing so at the Internet scale is not feasible. This is the reason we also conducted research on change detection of Web sites along the directions mentioned in the previous section. We are also studying further optimizations based on offline preprocessing of statecharts refinements and rewriting of queries. The relevant preprocessing will be re-activated when our system detects changes in one of the local sites. Another interesting line for future research is the automatic integration of applications. There are two main issues here. Given a target global application, the first problem is finding the relevant local applications among the many applications available on the Web. A possible solution is to use software component brokerage [23,30,9]. Based on ontologies and problem-solving methods, brokering services of Web repositories establish customer requirements, match a software model from a component library, and supply the glue logic according to the customer's specifications. Combining our system with these techniques for obtaining an automatic matching of existing applications to the target application could be very beneficial, and we plan to study the issue in future work. Once the relevant local applications are found, the next issue is writing the specification of the mapping between the global and the local applications. Some of this can be automated as well. In recent years there has been a significant amount of work on automating the translation of data among different sources and formats, based on source and target schema/DTD [15,39,25]. We plan to study how this can be incorporated here to deal with the data part of the specification, and furthermore, how it can be extended to also handle (at least some of) the dynamic part of the applications, namely, the mapping between the activities and flow.

Acknowledgements. We thank Ossnat Avihu, Vladimir Gamaley, and Avi Telyas for their help in the implementation of the AM system.

References

1. Abiteboul S., Amann B., Cluet S., Eyal A., Milo T., Mignet L. Active views for electronic commerce. In: Int. Conf. on Very Large DataBases (VLDB), Edinburgh, Scotland, September 1999
2. Abiteboul S., McHugh J., Rys M., Vassalos V., Wiener J.L. Incremental maintenance for materialized views over semistructured data. In: Int. Conf. on Very Large DataBases (VLDB), New-York, August 1998
3. Abiteboul S., Quass D., McHugh J., Widom J., Wiener J.L. The Lorel query language for semistructured data. International Journal on Digital Libraries, 1(1), 1997
4. Alonso G., Fessler A., Pardon G., Schek H.J. Correctness in general configurations of transactional components. In: PODS, 1999 /CHAIMS
5. Amazon.com. Earth's biggest selection. http://shoptheweb.amazon.com
6. Apache. The apache software foundation. http://www.apache.org
7. AristoCart. Aristocart home page. http://www.aristocart.com
8. Papakonstantinou Y., Ludascher B., Velikhov P. A framework for navigation driven lazy mediators. In: WebDB'99, 1999
9. Benjamins V.R., Plaza E., Motta E., Fensel D., Studer R., Wielinga B., Schreiber G., Zdrahal Z., Decker S. Ibrow3: an intelligent brokering service for knowledge-component reuse on the World-Wide Web. In: The 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW98), Banff, Canada, 1998
10. BizTalk. Biztalk home page. http://www.biztalk.org
11. Buneman P., Davidson S., Hillebrand G., Suciu D. A query language and optimization techniques for unstructured data. In: Proc. of the ACM SIGMOD Conf. on Management of Data, San Diego, Calif., USA, 1996
12. Carey M.J. et al. Towards heterogeneous multimedia information systems: the Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994
13. CHAIMS. Compiling high-level access interfaces for multi-site software. http://www-db.stanford.edu/CHAIMS/
14. Chang T.-P., Hull R. Using witness generators to support bi-directional update between object-based databases. In: Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS), San Jose, Calif., USA, May 1995
15. Cluet S., Delobel C., Simeon J., Smaga K. Your mediators need data conversion! In: SIGMOD'98, to appear, 1998
16. CMI. Cmi home page. http://www.mcc.com/projects/cmi
17. CommerceNet. Commercenet home page. http://www.commercenet.com/
18. CompareNet. Interactive buyers guide. http://www.compare.com
19. Crossflow. Crossflow home page. http://www.crossflow.org
20. cXML. cXML home page. http://www.cxml.org
21. Davulcu H., Freire J., Kifer M., Ramakrishnan I.V. A layered architecture for querying dynamic web content. In: Proc. of the ACM SIGMOD Conf. on Management of Data, Philadephia, Pa., USA, 1999
22. Dealpilot. The ultimate comparison shopping engine. http://www.dealpilot.com
23. Decker K., Sycara K., Williamson M. Middle-agents for the Internet. In: Proc. of the 15th Int. Joint Conference on Artificial Intelligence, IJCAI, pp. 578–583, Japan, 1997
24. Deutsch A., Fernandez M., Florescu D., Levy A., Suciu D. XML-ql: a query language for xml. http://www.w3.org/TR/NOTE-xml-ql/

25. Doan A., Domingos P., Levy A. Learning source descriptions for data integration. In: WebDB'00, 2000
26. Duschka O., Genesereth M. Answering recursive queries using views. In: ACM PODS, 1997
27. Duschka O., Levy A. Recursive plans for information gathering. In: Proc. 15th International Joint Conference on Artificial Intelligence, IJCAI-97, 1997
28. E-Speak. E-speak home page. http://www.e-speak.hp.com
29. Fabio C., Ski I., Li-Jie J., Vasudev K., Ming-Chien S. eFlow: a platform for developing and managing composite e-services. HP Labs Technical Reports, http://www.hpl.hp.com/techreports/2000/HPL-2000-36.html
30. Fensel D. An ontology-based broker: making problem-solving method reuse work. In: Workshop on Problem-Solving Methods for Knowledge-based Systems (W26), pp. 23–29, Japan, 1997
31. Fowler M. Analysis patterns – reusable object models. Addison-Wesley/Longman, 1997
32. Garcia-Molina H., Papakonstantinou Y., Quass D., Rajaraman A., Sagiv Y., Ullman J., Vassalos V., Widom J. The tsimmis approach to mediation: data models and languages. In: Journal of Intelligent Information Systems, 1997
33. Harel D. Statecharts: a visual formalism for complex systems. Science of Computer Programming, 8:231–274, 1987
34. Harel D. On visual formalisms. Comm. Assoc. Comput. Mach., 31(5):514–530, 1988
35. Ilogix. Ilogix homepage. http://www.ilogix.com/
36. Kushmerick N. Regression testing for wrapper maintenance. In: AAAI-99, 1999
37. Levy A., Rajaraman A., Ordille J. Querying heterogeneous information sources using source descriptions. In: VLDB, 1996
38. Levy A., Suciu D. Deciding containment for queries with complex objects. In: Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS), 1998
39. Milo T., Zohar S. Using schema matching to simplify heterogeneous data translation. In: Int. Conf. on Very Large DataBases (VLDB), New York, August 1998
40. Niesrstrasz O., Tsichritzis D (eds.). Object oriented software composition. Prentice-Hall, Englewood Cliffs, N.J., USA, 1995
41. OMG. Uml notation guide, version 1.1, 1 september 1997, 1997. www.omg.org/techprocess/meetings/schedule/Technology_Adoptions.htm
42. Papakonstantinou Y., Abiteboul S., Garcia-Molina H. Object fusion in mediator systems. In: VLDB, 1996
43. Rational Software. Unified development solutions & programming tools. http://www.rational.com
44. XML TreeDiff, 1999. http://www.alphaworks.ibm.com/tech/xmltreediff.
45. W3C. Extensible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml
46. W3C. Extensible stylesheet language (xsl). http://www.w3.org/Style/XSL/
47. W3C. The W3C query languages workshop, Dec. 1998, Boston, Mass., USA http://www.w3.org/TandS/QL/QL98/cfp.html
48. Wiederhold G., Wegner P., Ceri S. Towards megaprogramming: a paradigm for component-based programming. Communications of the ACM, 11:89–99, 1992
49. Yahoo. Shopping. http://www.shopping.yahoo.com
50. Yesha Y., Adam N. Electronic commerce: an overview. In: Adam N., Yesha Y (eds) Electronic commerce, Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York, 1996