Reusable Functional Composition Patterns for Web Services

Laurence Melloul and Armando Fox
Computer Science Department, Stanford University
{melloul,fox}@cs.stanford.edu

Abstract

Developers write Web service composition programs in terms of functionalities (e.g., "WebSearch") to postpone choosing which services of the same functionality to invoke (Google or Yahoo). We provide a higher level of abstraction than this for higher reuse. We express high-level "patterns" (e.g., "SearchAndCollectData") as both objects that can be "specialized" to particular applications ("SearchAndDownloadPapers" vs. "SearchAndAddBooksInCart") and objects that are reusable in the construction of higher-level ones. Our approach lets developers write patterns in terms of high-level functionalities (e.g., "CollectData") and later decide on services to compose that have lower-level functionalities (e.g., "DownloadPapers" or "addBooksInCarts"). We describe our prototype and show an example of nested pattern specialization. We also discuss a reuse trade-off, showing that too much abstraction makes the pattern less expressive. Rather, we suggest developers capture what must be guaranteed in every context of invocation, regardless of the service selection.

1. Introduction

When composing Web services as in [1], developers write programs in terms of *functionalities*, which are abstract functions having names describing their purpose, e.g., "WebSearch." The motivation is to postpone choosing which Web service to invoke, e.g., Google or Yahoo.

Consider that "SearchAndDownloadPaperAbstracts" is a similar activity to "SearchAndPutBooksInCart." Both require searching a database for a list of matching items and then acquiring them. In one case, the database is an online bibliographic database such as CiteSeer, and the acquisition process downloads the paper abstracts; in the other case, the database is a retailer's book catalog, and the acquisition process adds items to the shopping cart. We want the ability to express the higher-level pattern "SearchAndCollectData" that can be both *specialized* to particular applications ("SearchAndDownloadPaperAbstracts" vs. "SearchAndPutBooksInCart") and reusable in the construction of higher-level objects.

We provide a methodology and a framework to achieve these goals. Using our methodology, developers decide how abstract a pattern should be when they create it. Through the framework, they browse the functionalities and available services when creating and specializing a pattern.

In our approach, developers write patterns in terms of high-level functionalities, and *later* decide on the function signature they want to associate to a functionality appearing in a pattern. For example, in the case of the *SearchAndCollectData* pattern, a call to *CollectData* may be bound to *DownloadAbstract(PaperUrl,AbstractRegEx)* in one specialization or *PutInCart(ISBN, CartID)* in another.

Our system does not address automatic discovery or dynamic selection of Web services that match functionalities. Rather, we focus on how developers express compositions of functionalities. We do not require a machine-readable specification of service semantics; we assume that developers select services manually at pattern-specialization time.

The rest of the paper is organized as follows. In section 2, we explain our approach to writing functional composition patterns. In section 3, we formalize the conditions for binding services to the composed functionalities in a given pattern. In section 4, we expose a reuse trade-off that helps developers decide on the pattern abstraction level at creation time. We describe our system implementation in section 5 and provide examples of pattern specialization. Section 6 addresses the reuse of a pattern to match a functionality. Section 7 discusses our design limitations and the system's usability. We present related work and our conclusion respectively in sections 8 and 9.

2. Approach Overview and Definitions

Our goal is to help developers write high-level composition patterns that, when reused, are specialized through the choice of their components. There are two steps:

1. Writing patterns in terms of calls to "high-level" functionalities (e.g., "CollectData"). As we will show, there is a trade-off between how high-level (service-

independent) a pattern can be and how expressive the pattern logic can be.

2. Supporting the selection of Web services of "lower-level" functionality than the ones composed in the pattern (e.g., "DownloadPaperAbstracts" or "PutBooksInCart" for "CollectData").

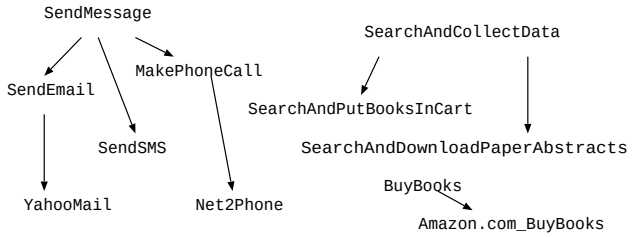

Figure 1. Functionality specialization.

We say that F1 is a *higher-level* functionality than F2 (and F2 a *lower-level* functionality than F1) if F1's purpose is more general than F2's. The lowest-level functionalities are Web service instances (e.g., Amazon.com_BuyBooks in figure 1). A functionality's purpose is a subjective notion; we assume developers or ontology creators decide on the functionality ordering.

If F1 is higher-level than F2, then F1 may be *specialized* into F2, that is to say, we may use F2 in place of F1 in a pattern. By transitivity, a higher-level functionality (e.g., SendMessage in figure 1) may be specialized into any Web service its lower-level functionalities specialize into (e.g., YahooMail). It is possible for a service to be usable in place of a given functionality in a pattern and not in another, because of the service constraints each pattern captures. We give shortly the conditions required to validate a functionality specialization in a given pattern.

A *pattern* is a program that embeds calls to functionalities, and it has no other distinctive characteristic. Since functionalities can be of different levels of abstraction, the pattern writer must decide on the level to use. For example, should the pattern call SendEmail, or call the more general functionality SendMessage and rely on binding-time specialization to select the email-sending function for binding to SendMessage? As we will show in section 4, the choice depends on the extent to which the inputs and outputs of the functionalities must be directly manipulated by the pattern logic. The more "visible" they are to the pattern, the more specific the functionalities must be, and in general the less reusable (*re-specializable*) the pattern will be. We discuss the trade-off arising from the policy decision of how high-level a functionality to use, while showing that our system provides the mechanism for handling binding at any level.

Such a general mechanism must be able to bind a wide selection of function signatures to a given functionality in the pattern, in order to encompass many different lower-level functionalities (e.g., SendEmail or SendSMS for SendMessage). To allow this, we let developers vary the number of parameters to the functionality call in the pattern, and we define the signature specified as the *minimal* required for a service to be selected. At specialization time, we can bind the functionality to any service that has *larger* sets of input and output parameters. For instance, a call to "SendMessage(MessageContent)" is compatible with services of the SendEmail or SendSMS functionalities, although these respectively require an email address and a phone number in addition to the message content. Because these additional inputs do not participate in the pattern's logic, we must provide their values outside the pattern. We can obtain them by calling services selected for other functionality calls in the pattern or for functionality calls in a code extension of the pattern. For an example of the latter, the phone number in the above example may be retrieved from a profile service.

To summarize, for a given functionality to be specialized into a given Web service in a pattern, the service must be of a lower-level functionality, it must have larger parameter sets, and all its inputs that are not part of the functionality call in the pattern must be provided outside the pattern.

3. Pattern Specialization Formalism

We now formalize these conditions. We thereafter specify the conditions needed to specialize the whole pattern.

Let M be a functionality and $S_M = \{m_1, m_2 ... m_k\}$ be the set of service functions of equal or lower-level functionalities than M. Referring to figure 1, if M is 'SendMessage', S_M is {YahooMail, Net2Phone}.

For all i in $[1 \dots k]$, we define $m_i : \{A_i^1, A_i^2 ... A_i^{n_i}\} \rightarrow R_i$, which are service functions with input parameters $A_i^1, A_i^2 ... A_i^{n_i}$ and result data R_i. All m_i's have fixed but likely different signatures, because they may be of different lower-level functionalities than M.

When used in the pattern P, M's call defines a minimal function signature in the context of P (M has no signature assigned to it outside a pattern). Let it be $M_P : \{A^1, A^2 ... A^{n_M}\} \rightarrow R$.

We address the following question: Given the functionality call statement $M_P : \{A^1, A^2 ... A^{n_M}\} \rightarrow R$ in P, is a service function $m_i : \{A_i^1, A_i^2 ... A_i^{n_i}\} \rightarrow R_i$ applicable for that invocation? That is to say, can we bind m_i to M in P?

We assume the pattern P has inputs $I_1, I_2 ... I_p$.

Definition 1. m_i is applicable to M in P if and only if:

1. $m_i \in S_M$ (m_i is of a lower-level or equal functionality than M).

2. Considering inclusive polymorphism, R is a subset of R_i (all output elements of M_P are outputs of m_i).
3. For all j in $[1 \ldots n_M]$, A_i^j and A^j have the same type (all inputs of M_P are inputs of m_i).
4. For all j in $[n_M + 1 \ldots n_i]$,
   - either A_i^j and I_s have the same type, for some s in $[1 \ldots p]$
   - or there exist:
     – a functionality M' used in P or in an extended code of P
     – and a service function $m_u : X \rightarrow R_u$ in S_M',

     such that m_u is applicable to M' and A_i^j is a subset of R_u.

   That is to say, for every additional input of m_i not specified as argument to the call of M in P, this input is provided either as an input to the overall pattern, or from an output of another service. This service is bound to another functionality of the pattern or to a functionality used in a code extension of the pattern. Recursively, each input required by the binding of such an extension is provided either as an input to the overall pattern, or obtained by calling another service in the pattern or in another extension, etc.

Definition 2. The pattern P' is a code extension to P if P' makes a call to a functionality whose binding to a service function provides a piece of data needed by P.

Through conditions 2 and 3 in Definition 1, we guarantee the functionality's signature as specified in P is the minimal signature required for any service to be selected. Through condition 4, we guarantee all inputs required by the call to the service function that is bound to M will be provided at run-time. There is no risk of having cycles when binding additional service input parameters, because an input is either bound to a pattern's input or to another service's output, both of which do not need to be bound.

Two call statements of the same functionality M may specify different minimal parameter sets in different patterns. Hence, different subsets of service functions from S_M may be applicable to the uses of M in different patterns.

A pattern is specialized if all its functionalities are specialized, that is to say, if they are all bound to service functions and all necessary function inputs are also bound. The use of minimal signatures gives us the flexibility to select various combinations of services for the same pattern. The conditions listed above guarantee that every combination defines a complete (although likely different) data flow.

4. Pattern Abstraction

When developers create patterns, their objective will be to maximize the potential for specialization. For each call in the pattern they will need to decide on the sets of input and output parameters to specify and on the level of functionality abstraction to use. In this section, we expose a trade-off that will help them make these decisions.

The trade-off is as follows : *The fewer data references to input and output parameters a given pattern's functionality call has, the more flexibility there will be in selecting service function signatures to bind to this functionality, but the less expressive the pattern will be.*

Expressiveness is the qualitative degree of data flow that is explicit in a pattern. Figure 2 is a simple pattern that sequentially invokes two functionalities, one to search for a book and a second to reserve it. Since neither call statement explicitly names arguments, the potential set of services that we can select is large. However, we lose the ability to specify data flow in the pattern itself and must rely on the functionality names only to understand what the pattern does.

```
(define GetBookAbsolutely
  (lambda ()
    (GetLibraryBookRef)
    (ReserveLibraryBook)))
```

Figure 2. No expressiveness.

Now consider a slightly different version of the pattern (Figure 3) in which we want to specify an explicit dependency between what the pattern does and some piece of data provided by executing the body of the pattern. In this case, the pattern will explicitly output a message confirming the request date. Therefore, the selected SendMessage service must take as input a "MessageContent" parameter. This pattern specifies constraints on the services to be selected, but it is marginally more expressive than the previous one.

```
(define GetBookAbsolutely
  (lambda (date)
    (GetLibraryBookRef)
    (ReserveLibraryBook)
    (SendMessage (string-append "book requested on " date))))
```

Figure 3. Little expressiveness.

The next version (Figure 4) encapsulates explicit conditional execution in the pattern, allowing a warning message to be generated if the desired book is not found. By embedding this in the pattern, we are saying that any execution of this pattern must process services that satisfy the call dependencies of GetLibraryBookRef and SearchBook-

store. Specifically, GetLibraryBookRef must be bound to a service that returns a reference to the desired book, or the empty string if the desired book is not found; and SearchBookstore must be bound to a service that clearly returns SUCCESS if the search succeeds. Note that a service that is bound to SearchBookstore very likely also produces the title of the book found, but the pattern does not need it. Although this pattern specifies more constraints on the service selection, it has a much higher level of expressiveness because it performs substantial local data processing.

```
(defi ne  GetBookAbsolutely
  (lambda  ()
    (cond  ((not  (string=?  (GetLibraryBookRef)  ""))
              (let  ([confNumber  (ReserveLibraryBook)])
                    (SendMessage  (string−append
                                        "reservation confirmed "
                                        confNumber))))
           ((string=?  (SearchBookstore)  "SUCCESS")
            (BuyBook)  (SendMessage  "book bought"))
            (else  (SendMessage  "Book not found")))))))
```

Figure 4. Large expressiveness.

In essence, by specifying what goes in the pattern, we are specifying what elements must be common to any execution of the pattern; whereas by selecting specific services that are compatible with the functionality call constraints in the pattern, we may select elements that are found only in some possible executions of the pattern.

Once developers decide what they want to capture in a pattern, they determine the level of abstraction for each composed functionality, based on the data references in the functionality call. For instance, a set of inputs such as (MessageContent, MessageSubject, ToEmailAddress, FromEmailAddress) will require a SendEmail service, whereas a set such as (MessageContent) will tolerate any SendMessage service.

5. Composition Framework and Example

We have described our proposed methodology to create and specialize patterns. We now take the position of developers who want to build two similar applications, one that looks for books, filters them based on their reviews, and adds them in a shopping cart; another that looks for movies, orders them based on the country in which they were produced, and adds them to a queue. We can abstract these two applications into the SearchAndCollectData pattern, which *searches for items, processes them, and collects them.*

Another developer who has access to this pattern may wish to reuse it for another purpose: to search for papers, extract their abstracts, and store them in a remote server.

In the following, we show how the SearchAndCollectData pattern is specialized into two different applications: SearchAndDownloadPaperAbstracts and SearchAndAddBooksInCart.

5.1. Injecting Web services in the framework

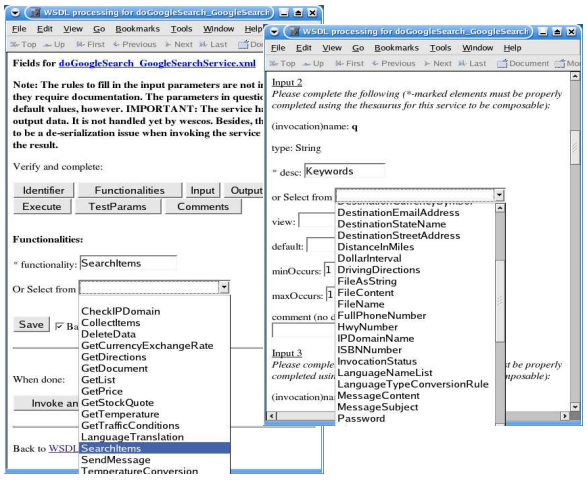

Figure 5. The SemMap tool.

To make a Web service available for composition, developers must map it to a functionality name and its inputs and outputs to common parameter names. The SemMap tool (Figure 5) parses the service's WSDL to extract the parameter information and lets developers browse existing common terms or add new ones to perform the mapping. For instance, the figure shows the selection of the "SearchItems" functionality for Google's search service as well as the mapping of its "q" parameter to "Keywords." Developers perform the semantics mapping once and may then reuse the services in many pattern specializations.

Common terminologies for functionalities and service parameters are useful for two reasons: first, to ease developers' understanding of the objects they want to reuse, and second, to permit the supporting system to provide optimizations. For instance, our current implementation deduces non-ambiguous data dependencies between the services selected during pattern specialization.

5.2. Pattern creation and specialization

Flatt's Units [2] are a Scheme language construct that provides for explicit and bi-directional binding. We express Web services, patterns, and functionality-to-service bindings as units. All these units import (require) and export (provide) functionalities. The bindings of one unit's exports to other units' imports take place in a *compound-unit*.

While processing a given WSDL, the SemMap tool also extracts other pieces of information relevant to the service's invocation. For each operation in the WSDL, the system automatically generates a Scheme unit (e.g., "doGoogleSearch" in Figure 6). The unit has the capability to invoke the actual Web service, by calling a Java servlet that makes the HTTP or SOAP request and returns the results in XML form.

```
; unit for doGoogleSearch_GoogleSearchService.xml
; automatically generated by the system
(define doGoogleSearch_GoogleSearchService.xml@
  (unit
    (import GetLicenseKey GetKeywords ...)
    (export SearchItems GetListOf_Title_URL_Snippet)
    (define _GetListOf_Title_URL_Snippet_var "")
    (define SearchItems
      (lambda ()
        ; invoke actual google Web service
        ; extract xml results
      ))
    (define GetListOf_Title_URL_Snippet
      (lambda () _GetListOf_Title_URL_Snippet_var))))
```

Figure 6. The doGoogleSearch unit.

The doGoogleSearch unit exports the functionality the service was mapped to, SearchItems. The unit also imports the service's inputs (in the example, "GetLicenseKey, "GetKeywords") and exports the service outputs ("GetListOf_Title_URL_Snippet"). This way, we can express service data dependencies in the compound unit once the service selection is made. (Each service parameter name has the "Get" prefix because we get the data through accessor functions.)

In figure 7, we show the SearchAndCollectData@ pattern unit. It is a simple pattern consisting of a sequence of four calls to imported functionalities: SearchItems, ProcessItems, CollectData, and VerifyCollectedData. It exports the "SearchAndCollectData" functionality.

```
(define SearchAndCollectData@
  (unit
    (import SearchItems ProcessItems CollectData VerifyCollectedData)
    (export SearchAndCollectData)
    (define SearchAndCollectData
      (lambda ()
        (SearchItems)
        (ProcessItems)
        (CollectData)
        (VerifyCollectedData)))))
```

Figure 7. Pattern SearchAndCollectData@.

To specialize this pattern unit, we bind its functionalities to equal or lower-level functionality services in a compound unit. Figure 8 shows one specialization of the pattern into the "SearchAndDownloadPaperAbstracts" application. In the compound's code, we assign the SearchAndCollectData@ pattern the variable Y (at the bottom of the unit). In the same statement, we bind its four imported functionalities to other services' exported functionalities. For instance, the first parameter of SearchAndCollectData@ in that statement, "(S0 SearchItems)," means that we bind the first import of the SearchAndCollectData@ pattern to the "SearchItems" export of the service that we have bound to S0. This service is doGoogleSearch. We use it to look for publications with user-submitted keywords on Citeseer. The second functionality that we bind in the Y statement is ProcessItems and we assign it to variable S3 (for the ordering of the imported functionalities, see the pattern itself in figure 7). We map S3 to a local function, "ExtractDocumentElements," used to extract abstracts from the result papers and concatenate them in a document string. We implement the third functionality, CollectData, by saving the abstracts on the remote XMethod's file server (variable S4). Last, for verification, we read back the stored file through XMethod's readFile service (S5).

We must specify in the compound all data dependencies that are not captured by the pattern. For instance, we bind S0's exported GetListOf_Title_URL_Snippet output to S3's input in line 9, and we pass S3's output, GetResultAsString, to S4. Also, we bind the "GetKeywords" parameter of doGoogleSearch in line 6 to the compound's GetKeywords import in line 3. A compound's import is similar to a main program's command line argument and is submitted when the compound is invoked. To bind a service input parameter to a code extension of the pattern, we would need to add another variable in the compound and bind the code extension functionality to a service that implements it. Then we would bind the required input to the service's corresponding output.

Because the pattern calls high-level functionalities with no input parameters, we can select many combinations of services to bind to these functionalities. Figure 9 shows a second specialization that searches for Amazon.com's books for a given query, filters them, and adds them to a shopping cart.

Looking at the two compound units, we see that we have built two different applications of the SearchAndCollectData pattern: "SearchAndDownloadPaperAbstracts" and "SearchAndPutBooksInCart." In most cases, the services selected in the two compounds for the same functionality have totally different function signatures. For instance, the two services that perform the CollectData functionality are writeFile_XMethodsFilesystemService.xml@ (Figure 8) and createAndAddItemsToShopping-

```
(defi ne SearchAndDownloadPaperAbstracts@
  (compound−unit
   (import GetLicenseKey GetKeywords GetFileName ...)
    (link
     [S0 (doGoogleSearch_GoogleSearchService.xml@
          GetLicenseKey GetKeywords ...)]
     ...
     [S3 (ExtractDocumentElements_local@
          (S0 GetListOf_Title_URL_Snippet) ...)]
     [S4 (writeFile_XMethodsFilesystemService.xml@
          (S3 GetResultAsString) GetFileName ...)]
     [S5 (readFile_XMethodsFilesystemService.xml@
          GetFileName ...)]
     [Y (SearchAndCollectData@
         (S0 SearchItems) (S3 ExtractDocumentElements)
         (S4 WriteData) (S5 GetDocument))]
     ...
     )
    (export)))
```

Figure 8. A pattern specialization.

```
(defi ne SearchAndPutBooksInCart@
  (compound−unit
   (import GetGroupID GetLicenseKey GetKeywords
           GetProductLine GetItemMaxPrice ...)
   (link
    [S0 (keywordSearch_amazon.com.xml@
         GetGroupID GetLicenseKey GetKeywords GetProductLine ...)]
    [S1 (FilterOutItems_local@
         (S0 GetListOf_Identifi erNumber_ Title_ ReleaseDate_ Price)
         GetItemMaxPrice)]
    [S2 (createAndAddItemsToShoppingCart_amazon.com.xml@
         GetGroupID GetLicenseKey
         (S1 GetListOf_Item_Output) ...)]
    [S3 (getItemsInShoppingCart_amazon.com.xml@
         GetGroupID GetLicenseKey
         (S2 GetCartIdentifi erNumber)
         (S2 GetCartSecurityRule))]
    [Y (SearchAndCollectData@ (S0 SearchItems) (S1 FilterOutItems)
                              (S2 CollectItems) (S3 CollectItems))]
    ...
```

Figure 9. Another specialization.

Cart_amazon.com.xml@ (Figure 9). They respectively take as inputs (GetFileContent, GetFileName, GetUserID and GetPassword) and (GetGroupID, GetLicenseKey, and GetListOf_IdentifierNumber).

Both compounds are valid because all their services' inputs are bound. Any other service combination that provides a complete data flow would also be valid.

We have developed a second graphical tool, the SpecTOOL, that lets developers specialize patterns without writing any code. They select services to bind to the composed functionalities of the pattern, and specify the data dependencies between these. When there is no ambiguity, the tool deduces these dependencies, thereby reducing developers' effort. In any case, it generates the corresponding compound and the application is ready for execution.

6. Building-up Concepts

In this section, we show how to bind *patterns* to composed functionalities in a larger pattern and hence reuse them as objects in the overall pattern.

6.1. The pattern-to-functionality binding process

Patterns, like services, export functionalities. They also have signatures. These consist of the pattern's exported functionality name (e.g., SearchAndCollectData) and its sets of data imports and exports.

The conditions required for binding patterns to functionalities in larger patterns include the ones we specified for binding services to functionalities: the pattern must be of an equal or lower-level functionality than the given functionality in the overall pattern, it must have larger input and output parameter sets, and any input parameters not specified in the functionality call statement in the overall pattern must be bound to external sources.

An additional condition is that the pattern that is selected for binding must also have all its imported functionalities bound, either to services or other patterns. In the latter case, these patterns too must have their imported functionalities bound, and so on. The process stops when all imported functionalities are eventually bound to services.

Web services are special cases of patterns which do not import functionalities. Therefore, binding functionalities to patterns is a generalization of the functionality-to-service binding process.

6.2. Example

We now show an example of nested pattern reuse. We want to do a better paper search through various lists of keywords: "Web service composition," "high-level composition patterns," and "abstract web services." Papers that would come out of all three searches would likely be more relevant to our work and we should begin by reading those.

We compose the SearchAndCollectData functionality in a new pattern, WeighCollectedData@: we call this functionality on each of our keyword set, and then call RankData on the result list of abstracts, assigning each abstract a weight according to its number of appearances (Figure 10).

The WeighCollectedData@ pattern imposes constraints on the pattern signatures that can be selected for binding to its SearchAndCollectData functionality call, in order to gain in expressiveness. The system will not accept the functionality's binding to the SearchAndCollectData@ pattern that we discussed earlier (Figure 7). The reason is that this pat-

```
(define WeighCollectedData@
  (unit
    (import SearchAndCollectData GetListOf_KeywordSet RankData)
    (export WeighCollectedData)
    (define WeighCollectedData
      (lambda ()
        (set! keywords_sets (GetListOf_KeywordSet))
        (for-each (lambda (set) (SearchAndCollectData set))
                  keywords_sets)
        (RankData)))))
```

Figure 10. WeighCollectedData@.

tern has a smaller set of inputs (empty set) than the functionality call's one (see the KeywordSet "set" input). However, many patterns may export the same functionality. A valid pattern for binding is SearchAndcollectData_2@ in figure 11 because it has an equal set of inputs (KeywordSet).

```
(define SearchAndCollectData_2@
  (unit
    (import SearchItems ProcessItems
            CollectData VerifyCollectedData GetKeywordSet)
    (export SearchAndCollectData)
    (define SearchAndCollectData
      (lambda ()
        (SearchItems (GetKeywordSet))
        (ProcessItems) (CollectData) (VerifyCollectedData)))))
```

Figure 11. SearchAndCollectData_2@.

The compound SearchAndWeighPaperAbstracts@ (Figure 12) shows how this larger application is created by specializing functionalities at two levels, for WeighCollectedData and SearchAndCollectData. In the compound's code, the WeighCollectedData@ pattern (variable Z) binds its SearchAndCollectData imported functionality to the functionality of the same name exported by the FromLambdaToImport unit (W), instead of binding it directly to the one exported by SearchAndCollectData_2@. The W unit hence implements a level of indirection necessary for binding: it exports the KeywordSet parameter (which was passed through the SearchAndCollectData call in the WeighCollectedData@ pattern) so that it can be bound to the import of SearchAndcollectData_2@. The system can automatically generate W.

7. Discussion

We discuss successively the limitations of our design and the framework's usability.

We have shown how we specialize patterns and reuse them as higher-order objects. However, we have an inherent paradox in our design, which makes the way we build up higher-level concepts suboptimal. The first pattern, SearchAndCollectData@ (Figure 7), can be specialized in many ways because it does not have any data imports. However, this also reduces its potential for reuse as part of another pattern. Indeed, its signature has fewer inputs than the SearchAndCollectData functionality call in the WeighCollectedData@ overall pattern, as we previously explained. Hence, the SearchAndcollectData@ pattern cannot be selected for that functionality in the overall pattern, although there exists a specialization that would let it consume the "KeywordSet" parameter (through the SearchItems binding to "doGoogleSearch" that we have specified in figure 8). We are currently investigating this problem and revising our design in order to address it.

```
(define FromLambdaToImport@
  (unit
    (import SearchAndcollectData_Pat)
    (export SearchAndCollectData GetCurrentKeywordSet)
    (define _set "")
    (define SearchAndCollectData
      (lambda (set) (set! _set set) (SearchAndCollectData_Pat)))
    (define GetCurrentKeywordSet (lambda () _set))))

(define SearchAndWeighPaperAbstracts@
  (compound-unit
   (import GetLicenseKey GetListOf_KeywordSets GetFileName ...)
    (link
     ...
     [S6 (rankData_local@ ...)]
     [W (FromLambdaToImport@ (Y SearchAndCollectData))]
     [Y (SearchAndCollectData_2@
          (S0 SearchItems) (S3 ExtractDocumentElements)
          (S4 WriteData) (S5 GetDocument)
          (W GetCurrentKeywordSet))]
     [Z (WeighCollectedData@ (W SearchAndCollectData)
                             GetListOf_KeywordSets
                             (S6 RankData))]
     ...
```

Figure 12. The SearchAndWeighPaperAbstracts application.

Regarding the usability of the system, there are several improvements we should make. A first area of improvement is to increase the potential of reuse of the available services and patterns in the system (let us note that discovering a pattern reduces to discovering the functionality it exports). To that purpose, we must help developers detect useful functionalities when they create patterns, and choose the most appropriate names when they add a service in the system. For instance, how would they know which functionality term to look for if they wanted to add items in a shopping cart? AddInCart? ShoppingCart? We plan to benefit from ontology work to find solutions to these issues.

A second area of improvement is to reduce the relative

amount of code developers have to write in the compound, compared to the pattern's code. Indeed, it currently appears that most of the work is done writing the compound as opposed to writing the pattern. We first plan to tackle more complex examples of nested patterns. Second, we intend to assist developers by refining the set of "boundable" services for the composed functionalities in a pattern. For instance, after the first service is selected, the tool could adjust the choices for the second service based on the data dependencies with the previously selected service, and so on. Last, we would like to use the pattern as a *functionality guide* for dynamically selecting Web services. The main problem we foresee involves safely binding missing inputs to external sources without compromising the semantics of the application. For instance, if a service available for selection requires a "phoneNumber" not provided by the pattern, in what conditions can this piece of data be retrieved from an external service S?

8. Related work

Our work differs from related work in that it permits developers to specialize patterns with services of different lower-level functionalities than the composed ones. In contrast, [1] provides support to select different Web services that map to the same functionality.

Adding a service in the system requires the mapping steps defined in section 5.1. The system then generates the code to perform the Web service call. Despite the fact that the reusability of the service depends on its functionality name in our approach, there is no code-writing effort required to reuse a pattern, which contrasts with creating a wrapper to perform a call redirection in the middleware [3]. Besides, reusing a program with services of different functionalities for the same call seems just insuitable with proxies or interceptors, because of the discrepancy between the code that is visible and the executed one.

Although we do not use classes and derivation explicitly, our functionality hierarchy suggests an object-oriented model. A service may be seen as an object of a functionality class that has one method only (the service's invocation function). However, the flexibility to reuse and extend a pattern doing late binding in such an explicit way as we have done with units is a fair advantage towards the OO tools currently available. Indeed, we offer a clear separation between the code for the pattern logic and the code specific to a given service selection.

We plan to benefit from current effort on Web service descriptions [4] and ontologies [5] once these technologies or tools are mature for Web services.

Finally, we distinguish our work from automatic Web service composition such as the Semantic Web [5] and AI planning [6] where the goal is to produce a composition plan. Rather, we start with a (high-level) plan and focus our work on its reuse.

9. Conclusion

Through this work, we have shown how developers can create high-level patterns that get specialized in many ways through component selection. Specialization may occur with a selection of services whose parameter requirements were not anticipated at the time the pattern was written. We have also described how developers can build up concepts by reusing patterns.

We have provided a methodology and framework to realize our approach. We have exposed a trade-off in deciding on the abstraction level with which to create patterns: the fewer parameter references a given functionality call makes in a given pattern, the more flexibility there will be in selecting services of various lower-level functionalities for that call, but the less expressive the pattern will be.

The examples suggest that it is easy to specialize patterns by creating new compound units. However, they also underline an issue of usability that we look forward to address. Our goal is to investigate this approach further and verify its viability.

Acknowledgments We would like to deeply thank John Mitchell for pointing out Matthew Flatt's work and for his helpful advice. We are grateful to the following persons for their time and comments: the reviewers, Charles Petrie, Constantine Sapuntzakis, Emre Kiciman, George Candea, Iddo Lev, James Cutler, John Davis, and Shankar Ponnekanti. Finally, we thank France Telecom and our sponsors at the Stanford Network Research Center for supporting this research work.

References

[1] B. Verheecke, M. A. CibrnVerheecke. AOP for Dynamic Confi guration and Management of Web Services. *The International Conference on Web Services - Europe*, 2003.

[2] M. Flatt. Programming Languages for Reusable Components. *Thesis*, 1999.

[3] N. Wang, K. Parameswaran, D. Schmidt. The Design And Performance of MetaProgramming Mechanisms for ORB Middleware. *Conference on Object-Oriented Technologies and Systems*, 2001.

[4] V. Tosic et al. Web Service Offerings Language (WSOL) and Web Service Composition Management (WSCM). *Workshop on Object-Oriented Web Services*, Seattle, 2002.

[5] S. McIlraith et al. Semantic Web Services. *IEEE Intelligent Systems, Special Issue on the Semantic Web*, Volume 16, No 2, pp46-53, March/April 2001.

[6] C. Petrie et al. Adding AI to Web Services. *Agent Mediated Knowledge Management*, LNAI 2926, Springer 2004, 322-338.