HAL
open science

Anchoring modularity in HTML.

Claude Kirchner, Hélène Kirchner, Anderson Santana de Oliveira

To cite this version:

Claude Kirchner, Hélène Kirchner, Anderson Santana de Oliveira. Anchoring modularity in HTML.. 1st International Workshop on Automated Specification and Verification of Web Sites - WWV'05, Mar 2005, Valencia/Spain, pp.139-151. inria-00000750

HAL Id: inria-00000750

https://inria.hal.science/inria-00000750v1

Submitted on 16 Nov 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Anchoring modularity in HTML

Claude Kirchner

INRIA & LORIA

Hélène Kirchner

CNRS & LORIA

Anderson Santana [1]

INRIA & LORIA

Abstract

Modularity is a key feature at design, programming, proving, testing, and maintenance time, as well as a must for reusability. Most languages and systems provide built-in facilities for encapsulation, importation or parameterization. Nevertheless there exists also languages, like HTML, with poor support for modularization. A natural idea is therefore to provide generic modularization primitives.

To extend an existing language with additional and possibly formal capabilities, the notion of *anchorage* and *Formal Island* has been introduced recently. TOM for example, provides generic matching, rewriting and strategy extensions to JAVA and C.

In this paper, we show on the HTML example, how to add modular features by anchoring modularization primitives in HTML. This allows one to write modular HTML descriptions, therefore facilitating their design, reusability, and maintenance, as well as providing an important step towards HTML validity checking.

Key words: Modularization, parameterization, HTML, TOM, `MHTML`, formal island, feature anchorage

1 Introduction

Modularity is a key feature of programming environments at all stages of software development, from users requirements analysis to maintenance. It is of course the key feature of reusability policies and therefore a main concept in

[1] Supported by CAPES.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

any software library. With the raising time of safe and secure software, modularity appears also as a fundamental feature to make possible the construction of large certified software.

Modularity is thus present in many programming languages and proof environments and we have now a fairly good understanding of the semantics of the main modularity constructs. In particular in functional and algebraic programming, the notions of importation, parameterization and visibility have been given both categorical and operational semantics (e.g. [2,6,12]).

If from the theoretical point of view, the situation is satisfactory, this is not the case from the practical one, in particular because each language has its own modularity features and semantics. Clusters in CLU, packages in Ada, structures in ML, classes in C++ and Java are different constructs facilitating modular programming. Some languages have quite sophisticated modularity features, like CASL, OBJ and Maude, where the notion of *view* precisely formalizes the way parameters are instantiated or modules imported. Others, like ELAN have a more elementary approach. Object-oriented languages like Java take into account classes and inheritance. Functional languages, such as ML, have also evolved towards modularity. Face to this variety of approaches, we are thus in a situation where standard modularity features, mainly independent of the language, would be greatly appreciated.

But modularity is not a universal feature of programming languages and several of them lack of useful capabilities. For example, parameterization does not exist in ASF+SDF nor C. An extreme example in this case is HTML that has no importation nor parameterization capability at all.

So either for standardization or for needed improvement, it is desirable to have the capability of adding modularity features to an existing programming language.

While we understand its usefulness, we have now to address the feasibility of adding modularity primitives in a programming environment. This question has been explored in [12] where an SML-like module system is presented that can accommodate a variety of base languages, provided they satisfy mild assumptions.

Another approach, independently developed, is the formal island paradigm that comes into play in a simple and pragmatic context: indeed, it would be nonsense to throw away the billions of code lines that are in use today in all domains of human activities, nevertheless it is clear that all these software have to be considerably improved in their logic, algorithmic, security and maintenance qualities. As introduced in TOM [2] [13,8] in particular for matching, normalization and strategic rewriting, formal islands allow for the addition to existing programming languages, of formal features that can be compiled later on into the host language itself, therefore inducing no dependency on the formal island mechanism.

[2] `tom.loria.fr`

At the price of a rigorous anchoring, that provides the link between the host language data structure and the formal objects, the formal island approach gives the possibility (*i*) to extend the expressivity of the language with higher-level constructs at design time, (*ii*) to perform formal proof on the formal island constructions, (*iii*) to certify the implementation of the formal island compilation into the host language [11].

In addition to these benefits, what makes formal islands even more attractive is that they are shared between several implementations made in different programming languages. For instance, TOM provides matching, normalization and strategic rewriting in Java (this is jTOM), in C (this is cTOM) or in CAML (mlTOM).

To set-up the general definition of Modular Formal Island is a difficult goal and a first work towards making TOM modular for algebraic specifications in the vein of CASL has been done in [9].

The purpose of this paper is to present a first step for anchoring modularity in an existing language and to illustrate the approach with HTML. This allows writing modular HTML descriptions, therefore facilitating their design, reusability, and maintenance, as well as providing an important step towards HTML validity checking. While we only deal in this paper with the HTML case, one important interest of the proposed approach is to set-up the basis for a generic method.

In order to make precise our objectives, we use in Section 2 a running example of a one block HTML page and show how we would like it to be decomposed in significantly smaller pieces. We then present in Section 3 the modularity features added to HTML and give in Section 4 its operational semantics, thus making clear the compilation process. Related work and further extensions are addressed respectively in Section 5 and Section 6.

2 Motivating example

Let us first consider an illustrative example of how modularity can help for the task of construction and maintenance of WEB sites.

Commonly, WEB pages composing a WEB site share some contents. This is related either to standard information that must appear on each page, or to navigability issues, for example, sets of links repeatedly presented to site visitors leading to other pages on the site.

The current trend among WEB designers is to drop the use of frames[3], which allows shared content between pages to be repeated in every page of the site. Especially for WEB sites built without the use of a script language, the webmasters have literally to "copy and paste" the static information from one page to another. Consequently, updates become an annoying task.

[3] http://www.w3c.org/TR/REC-html40/present/frames.html

A typical WEB site layout is shown in Figure 1. All pages that follow the home (index) page would share the same code for page header, menu, and page footer. The interaction of a visitor with the links contained on the list of pointers on the left, brings a similar page where the "content" box, as indicated in the figure, displays varying information, according to the subject of that page.

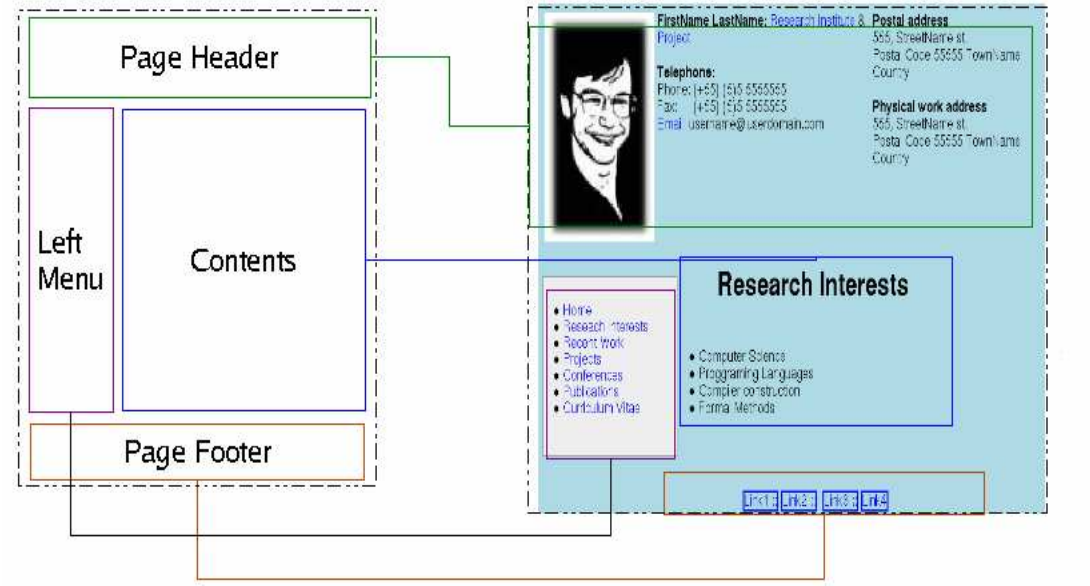

Fig. 1. A typical WEB site layout

Having the capability to isolate each piece of repeated code, WEB sites writers could (semi-)automatically compose a new WEB page from separate sources. In the example above, the page should be broken into reusable named modules for the page heading, navigation menu, and for the page footer.

Moreover, introducing parameters would allow reusing a similar page format for another user, facilitating uniform presentation of WEB sites.

As a third concern, in this context, each repeated piece of code could be checked only once to be well-formed from an HTML point of view.

Our goal is now to give a practical answer to these concerns, through the proposal of a formal language and its anchoring process in HTML.

3 The Modular HTML language

In this section we present the implementation of Modular HTML (`mHTML`). Basically, we use the algebraic specification formalism to express syntax and semantics of the language. Similar approaches have been used in the definition of Full Maude [5] as well as a module algebra for ASF+SDF [1]. Nevertheless, no sophisticated knowledge about algebraic specifications is required to understand the mechanism on which this execution model is based.

The mHTML language is a combination of the regular HTML markup with structuring primitives that allows the composition of documents through the reuse of existing code fragments. The principles that have guided its design are simplicity and genericity.

Modules

A first concern in modularization is how to define a self-contained piece of reusable code that represents a module. Our approach leaves to the user the task of defining the desired degree of granularity, since we do not restrict the modules to be exactly valid HTML code. Actually, any well-formed set of HTML markup can be considered as a module. For example, by using the `%module` primitive, we can define a reusable "menu" available for all pages in a web site in the following manner:

Example 3.1 The left menu of the web page in Figure 1 is described in the following module.

```
%module menu
 <ul>
      <li><a href="index.html">Home</a></li>
      <li><a href="researchInterests.html">Research Interests</a></li>
      <li><a href="work.html">Recent Work</a></li>
      <li><a href="projects.html">Projects</a></li>
      <li><a href="conferences.html">Conferences</a></li>
      <li><a href="publications.html">Publications</a></li>
      <li><a href="cv.html">Curriculum Vitae</a></li>
 </ul>
```

By convention and in order to facilitate access to modules, we restrict to one module per file, and impose that module and system file containing it have the same names. Thus a module called `foo`, will be found in a system file called `foo.mhtml`.

Imports

Secondly, a common need in the context of reuse is to access symbols declared by other modules. This can be achieved through the `%import` primitive, The effect of the import primitive at compile time is a textual expansion of the document. The imported module is inserted at the position the `%import` mark is found, repeated headers will be ignored, only the main module header is kept. In the case where the imported module has itself importations, this process is recursively performed.

Another important concern is to force the imported modules to be well-formed HTML. Alternatively it would be desirable to import *valid* HTML code, what would mean that the imported code should satisfy all the requirements from the W3C current HTML specification [14]. This is achieved by another feature of the language which provides a theory notion, close to the

notion of type. Setting in MHTML that a module mod is well-formed (resp. valid) is achieved by the declaration mod :: WellFormedHTML (resp. mod :: ValidHTML). At compile time, in the HTML case, this property is checked by calling the appropriate HTML tools.

The example below illustrates these issues considering the WEB site presented in Section 2:

Example 3.2 The WEB page of Figure 1 is written as the following module.

```
%module page1
<html>
  <head>
    <link href="styles.css" rel="stylesheet" type="text/css" />
    <title>Home Page</title>
  </head>
  <body>
        %import header :: WellFormedHTML
        <div class="header">
            <h1 class="title">Research Interests</h1>
        </div>
                <div  class="menu">
            %import menu :: WellFormedHTML
        </div>
        <div class="content">
            %import contents :: WellFormedHTML
        </div>
         <div class="footer">
            %import footer :: WellFormedHTML
        </div>
   </body>
</html>
```

Templates

The language also provides a template mechanism, that extends the language with parameterized modules. In this case, actual parameters are expected to be other modules, that in their turn, may be required to conform to a certain theory. In the HTML case, we currently simply consider these as either valid or simply well-formed HTML. Again, the parameters are transformed into a textual expansion when the preprocessor composes instances of such templates. The following example shows the definition of a parameterized MHTML module:

Example 3.3 For reusing the structure of the WEB page of Figure 1, the following template is provided:

```
%module template1 [
              Title
              Header   :: WellFormedHTML
```

```
              Menu     :: WellFormedHTML
              Contents :: WellFormedHTML
              Footer   :: WellFormedHTML   ]
<html>
  <head>
    <title>%use Title</title>
  </head>
   <body>
           %use Header
        <div class="header">
            <h1 class="title">Research Interests</h1>
        </div>
        <div class="menu">
           %use Menu
        </div>
         <div class="content">
           %use Contents
        </div>
          <div class="footer">
           %use Footer
        </div>
   </body>
</html>
```

This template can generate a new instance with the following instantiation:

Example 3.4 Actual module names are substituted to parameters to obtain an appropriate instance of the previous template:

```
%module publications
%import template1 [myHomePage
                   myPageHeeader
                   myBookmarks
                   myPublications
                   myPageFooter]
```

Figure 2 presents the syntax used in the examples above.

4 Semantics, anchoring and compilation

We are now ready to describe how the modular features provided in `MHTML` can be compiled into HTML. From an implementation point of view, the compiler accepts as input an `MHTML` program, consisting of native code combined with the modularity primitives of the language described in Section 3 and generates pure native code as output. For describing this compilation process, we choose to write an algebraic specification with rewrite rules, implemented in ELAN 4 [10]. In this way, we also provide the operational semantics of our `MHTML` language. Outlines of this specification is given in Figure 3. This is sim-

Module	::=	%*module* ⟨moduleName⟩ ("[" **Param**+ "]")∗ **Body**
Body	::=	([**Import**][**Use**] [**HTML**])∗
Import	::=	%*import* ⟨moduleName⟩ [**Paraminst**+] ["::" **Theory**]
Param	::=	⟨paramName⟩ ["::" **Theory**]
Paraminst	::=	⟨moduleName⟩
Use	::=	%*use* ⟨paramName⟩
Theory	::=	WellFormedHTML | ValidHTML
HTML	::=	Any HTML code with the clear exception of the primitives listed above

Fig. 2. Syntax for the generic structuring constructs

ilar to (but indeed simpler than) the rules given in [5] to describe the (more elaborated) modularity features of the Maude language.

ELAN 4 shares the syntax definition of SDF [4]. HTML code is represented by the sort `HTML`. The next rules can be read as follows:

- The first rule, establishes that modules with no further structuring constructs than `%module` should be transformed into HTML code.

```
[ ] translate ( %module m html ) => html
```

- The couple of rules below state that any importation without parameters nor theories should start a recursive call to the translate function, and as result would concatenate the produced HTML in the position the `%import` mark was found

```
[ ] translate ( %module m %import m1 html ) => translate(load(m1)) html
[ ] translate ( %module m html %import m1 ) => html translate(load(m1))
```

- The following rules have a similar effect as the previous ones, but they ensure that the modules being imported conform to a certain theory, through the `satisfies` function.

```
[ ] translate ( %module m html %import m1 :: th) =>
          html translate(load(m1))
    if satisfies( load(m1), th)
[ ] translate ( %module m  %import m1 :: th html) =>
          translate(load(m1)) html
    if satisfies( load(m1), th)
```

- The next two rules deal with the importation of parameterized modules, in this case a second version of the translation function is activated to treat the actual parameters.

```
[ ] translate ( %module m %import m1 [i] html) =>
       translate(load(m1), i) html
[ ] translate ( %module m %import m1 [i] :: th  html) =>
       translate(load(m1), i) html
    if satisfies( load(m1), th)
```

```
module  mhtmlTranslate

imports basic/ElanLayout
        basic/BuiltinBool
        mhtmlSyntax

       exports
               context-free syntax

                       translate(Module)                 -> HTML
                       translate ( Module , ParamList ) -> HTML
                       HTML HTML                         -> HTML

                       load( ModuleName )         -> Module
                       satisfies(Module, Theory)    -> BuiltinBool
        hiddens
                variables
                       "m"[0-9]*    -> ModuleName
                       "x"[0-9]*    -> ParamName
                       "p"[0-9]*    -> ParamDecList
                       "i"[0-9]*    -> ParamList
                       "html"[0-9]* -> HTML
                       "th"[0-9]*   -> Theory
rules
   [] ...(c.f. text)
```

Fig. 3. Rewrite rules for the MHTML compiler

- The following rules run through lists of formal parameters to verify them against actual parameters, and perform the corresponding substitution wherever the parameter occurrence was found

```
[ ] translate ( %module m  [x p] html %use x, m2 i ) =>
              translate ( %module m [p] html translate(load(m2)) , i)

[ ] translate ( %module m  [x :: th  p] html %use x, m2 i ) =>
              translate ( %module m [p] html translate(load(m2)) , i)
         if satisfies(load(m2), th)
```

These rewrite rules provide an outline of the compilation of MHTML to HTML. When completed with the full description of the instantiation process, this specification provides a normalization process that compiles MHTML into HTML. Compared to the implementation of *formal islands* provided for matching and rewriting in TOM, this process is simpler from several points of views. With the modularity primitives introduced here, there is no need for an anchoring dealing with the structure of terms. This could change when considering, for example, dependent types. So rather than a "deep" anchoring involving the structure of terms as used in [11], we need here only a "shallow" one dealing with the module content only. Another difference is that we

do not have to certify the compilation process in the same way: in [11], the matching compilation has to be certified, taking into account the properties of the anchoring. In our case, the verification process will concern the validation of the importations and of the parameter instantiations, leading in particular the way to another use of type systems for HTML.

5 Related Work about HTML

Restricting our attention to the specific question of anchoring modularity in HTML, the question arises to identify the improvements provided by our approach with respect to the WEB site development process. First of all, `MHTML` provides the following main advantages:

- It is independent of the WEB server hosting the WEB site.
- It is not necessary to process the language in a graphical environment, like in WYSIWYG HTML editors. This simplifies the maintenance process. It could also be used by the graphical environment to produce modular and hopefully readable code.
- It has a lightweight compilation process.
- It is easy to learn.
- It does not need to be executed every time the site is accessed.
- Finally, we should emphasize that, as for any anchorage, it does not induce any dependence on the language extension, as all the anchored language features are compiled into the target language.

The lack of HTML modularity has of course already drawn attention and we can mention the following WEB related works.

The JWIG project[3] provides an implementation of a language aimed at designing interactive web services. It is based on a session centered execution model of requests made through the WEB. As a secondary result, it provides a template language for dynamic WEB page construction. This language allows the definition of gaps, that may be filled with HTML code. Surely, the approach provides reuse of HTML code, but it is dependent on the whole environment to do simple structuring tasks.

In [7] a complete management environment with a language is developed to attack the management problems that appear in the implementation of data intensive WEB sites. The system combines a query language to specify the site's structure and content with a template language for its HTML representation. Although reusability is not the main concern of this work, the template language offers flexibility and extensibility to the creation of the site, it presents the same disadvantage as the previous one.

With respect to scripting languages like PHP, ASP, PERL, etc, this approach has the advantage of being simple and straightforward for the user. Another advantage of `MHTML` when compared to scripting languages or server

side includes, available in Apache for example, is that it does not need to be re-executed every time the root module is accessed via the WEB. Moreover we believe, although this has not yet been considered, that MHTML can be combined with other languages in the development of WEB sites.

Similar forms of template mechanisms are provided by a number of WYSIWYG HTML editors. This restricts the re-use of HTML because the user depends on a graphical environment to generate a new WEB page from existing templates, whereas the same functionality can be obtained in MTML through a simple text editor. It is also obviously possible for the user to design his page in his favorite WYSIWYG editor, and after, determine what are the parts he would like to reuse from that page in MHTML.

6 Conclusion

The approach described in this paper is part of a much larger enterprise towards the non-invasive diffusion of formal methods and algebraic methodology through the concept of Formal Islands. For example, on one hand matching and strategic rewriting may help to model large pieces of code. On the other hand, modularity is of fundamental use in the structuration of large software.

We have developed the idea of *modular anchoring* on the example, simple but useful, of the HTML language. Introducing quite simple primitives for importation and parameterization, we have shown how this can define a modular extension of HTML. The compilation process has been outlined using ELAN 4.

This is of course a preliminary work and we are actively working on deepening several points. First an implementation of MHTML in on its way and we naturally chose as implementation framework TOM itself. This will allow us to play with the concept and to validate our initial ideas. Second and quite importantly, verification tools specific to HTML should be used or developed. One can think of course, as we mentioned before, to the (X)HTML code validation as provided by the W3C tools. Specific type systems will also be developed to ensure desirable properties. For example and as for algebraic languages, structured data types could be statically typed using prescriptive type systems. Also, the introduction of views for parameter validation will require for their verification to perform non-trivial proofs and the experience gained again from the algebraic specification community will be of great interest here. We may also think to specific verification tools for HTML, like checking the reachability of linked objects. Of course HTML is a very specific and elementary language. A natural extension will concern XML, in particular for the logic and semantic WEB, and modularity features will be of great use in projects like Rewerse [4].

Acknowledgments This paper benefits of the many fruiful discussions we had

[4] rewerse.net

with Pierre-Etienne Moreau, Horatiu Cirstea and Antoine Reilles, in particular on formal islands. We also thanks the anonymous referees for their constructive comments on the first version of this paper.

References

[1] Bergstra, J. A., J. Heering and P. Klint, *Module algebra*, Journal of the ACM **37** (1990), pp. 335–372.

[2] Bidoit, M., D. Sannella and A. Tarlecki, *Architectural specifications in* CASL, Formal Aspects of Computing **13** (2002), pp. 252–273.

[3] Christensen, A. S., A. Moller and M. I. Schwartzbach, *Extending Java for high-level Web service construction*, ACM Trans. Program. Lang. Syst. **25** (2003), pp. 814–875.

[4] Deursen, A., J. Heering and P. Klint, "Language Prototyping," World Scientific, 1996, iSBN 981-02-2732-9.

[5] Durán, F., "A Reflective Module Algebra with Applications to the Maude Language," Ph.D. thesis, Universidad de Málaga, Spain (1999), http://maude.csl.sri.com/papers.

[6] Durán, F. and J. Meseguer, *Structured theories and institutions*, in: M. Hofmann, G. Rosolini and D. Pavlović, editors, *Proceedings of 8th Conference on Category Theory and Computer Science, Edinburgh, Scotland, September 1999*, Electronic Notes in Theoretical Computer Science **29** (1999), pp. 71–90, http://www.elsevier.nl/locate/entcs/volume29.html.

[7] Fernandez, M., D. Florescu, A. Levy and D. Suciu, *Declarative specification of web sites with strudel*, The VLDB Journal **9** (2000), pp. 38–55.

[8] Guyon, J., P.-E. Moreau and A. Reilles, *An integrated development environment for pattern matching programming*, in: *2nd eclipse Technology eXchange workshop - eTX'2004, Barcelona, Spain*, Electronic Notes in Theoretical Computer Science, Brian Barry and Oege de Moor, 2004.

[9] Hrvatin, S., *Structuration pour les spécifications à base de règles : Etude et mise en œuvre pour TOM*, Rapport de DEA, Université Henri Poincaré - Nancy 1 (2004).

[10] Kirchner, C. and H. Kirchner, *Rule-based programming and proving : the ELAN experience outcomes*, in: *Ninth Asian Computing Science Conference - ASIAN'04, Chiang Mai, Thailand*, 2004.

[11] Kirchner, C., P.-E. Moreau and A. Reilles, *Formal validation of pattern matching code*, Submitted (2005).

[12] Leroy, X., *A modular module system*, Journal of Functional Programming **10** (2000), pp. 269–303.

[13] Moreau, P.-E., C. Ringeissen and M. Vittek, *A Pattern Matching Compiler for Multiple Target Languages*, in: G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, LNCS **2622** (2003), pp. 61–76.

[14] Raggett, D., A. L. Hors and I. Jacobs, *Html 4.01 specification* (1999), http://www.w3.org/TR/REC-html40/.