# From XML Specifications to Parallel Programs *

Ignacio Peláez, Francisco Almeida, and Daniel González

Departamento de Estadística, I. O. y Computación
Universidad de La Laguna, c/ Astrofísico F. Sánchez s/n
38271 La Laguna, Spain
ignacio.pelaez@gmail.com, falmeida@ull.es, dgonmor@ull.es

**Abstract.** Skeleton-based libraries are considered as one of the alternatives for reducing the distance between end users and parallel architectures. We propose a general development methodology that allows for the automatic derivation of parallel programs assuming the existence of general structures as the skeletons. We propose the introduction of a new, high level abstraction layer that allows the user to extract problem specifications from particular skeleton languages or libraries. The result is a tool that allows for the generation of parallel codes from successive transformations to this high level specification without any loss of efficiency. We apply the technique to the automatic generation of parallel programs for Dynamic Programming Problems.

## 1 Introduction

A widespread research methodology among scientists is based on developing a mathematical formulation that conceptually provides a solution to a problem. However, it is also common for the problem to remain partially unsolved if the scientist is not able to transform this conceptual solution into a computer program. Researchers usually bridge this gap by using software tools like mathematical solvers, or even by writing their own code. This gap is even more evident in the case of parallel computers, where much more effort is required by the user.

A substantial effort has been made in recent decades to reduce the distance between parallel architectures and end users. As stated in [2], skeletal programming has emerged as an alternative and has helped simplify programming, enhance portability and improve performance. Such systems conceal the parallelism from the programmer and are characterized by being embedded entirely into a functional programming language, or for integrating imperative code within a skeletal framework in a language or library. Some of these approaches can be seen in [4], [8], [3], [1], [6], [11], [5], [2]. The underlying idea of separating the specification of a problem, or an algorithm, from implementation details that are hidden to the user is present in all the proposals.

As also stated in [2], skeletal programming has yet to make a substantial impact on mainstream methods in parallel applications programming. It is safe to say that in many cases, the end user of many algorithmic skeletons, like those

* Supported by MCyT projects TIN2005-09037-C02-01

derived from algorithmic techniques (Divide and Conquer, Simulated Annealing, etc.), is not a programmer but a scientist, say a biologist or a physicist, or a mathematician. The potential use of this kind of tool is strongly conditioned by its ease of use. Those who lack expertise in object or functional oriented programming, or who have no programming knowledge at all, should be able to apply them.

Based on XML specifications, we propose to introduce a new abstraction level that contributes to a higher separation between the specification of a problem and the implementation details. Skeletons act as an intermediate level between the specification made by the user and the parallel implementation, so that the user describes the problem through a general model and successive transformations convert the model into the instance code for a skeleton. The distance between the scientific knowledge and the specification is bridged by the scientist (not a programmer), while the distance between the specification and the executable code is bridged by the machine. Since the code generated is based on skeleton tools, the approach still maintains the advantages of the skeletal development methodology while providing significant benefits for the scientist:

- No need for codification. The user specifies the problem and does not codify the algorithm.
- Independence from specific programming languages or skeleton libraries. Once the problem has been specified, it can be transformed into several implementation proposals.
- Delivery of new applications due to the rapid development time.
- Improved application quality.
- Increased use of parallel architectures by non-expert users.
- Rapid inclusion of emerging technology into their systems. New transformers can be delivered when needed.

This paper makes two primary contributions: a general development methodology for deriving parallel programs automatically, starting from a very high level problem specification, and an application example in generating parallel programs for Dynamic Programming (DP from now on) problems.

Although we focus on a specific scientific domain, the parallelization of DP Problems, from a general point of view the technique presented here can be considered as a particular instance of the MDA (Model Driven Architecture) [9] general software engineering development methodology, and can be applied to many other scientific domains. We will show how this methodology can be applied, without any loss of efficiency, to parallel architectures. We also require the project to adhere to general standards as much as possible; for that reason, we follow the W3C [20] recommendations along every transformation step.

This paper is structured as follows: in Section 2 we introduce the software architecture of the proposed methodology and show how the technique is applied for the particular case of Dynamic Programming problems, and in Section 3 an XML specification for Dynamic Programming is presented. The paper finalizes with some concluding remarks and future lines of work.

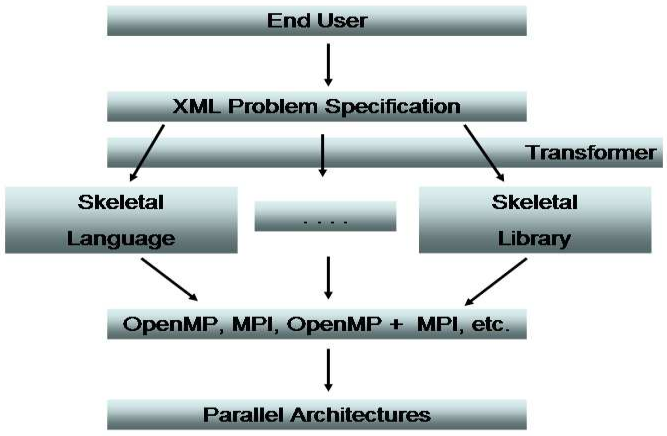


**Fig. 1.** Software Architecture for the Methodological Approach.

## 2 Software Architecture

We propose the introduction of a new abstraction layer between the user and the skeletons so as to separate the fundamental logic behind a problem specification from the specifics of the particular middleware that implements it (the skeleton instance). The advantages of this approach were listed in Section 1. This new layer should be closer to the scientific notation and should also be simple enough to generate the appropriate code. In general, the use of XML as the specification language is not mandatory; however, technologies based on XML specifications have proven to be interesting standard alternatives for transformation into different formats. Although we use this specification to generate C++ code, it could be used to generate WSDL [19] and web services applications.

The software architecture of our transformation methodology is presented in Figure 1. The layer between the end user and the skeletal libraries and languages tries to span the significant gap for those scientists who are not directly involved with programming. In order to provide access to data coming from an XML document, an analysis of the document and its decomposition into nodes and recognizable pieces through a standardized API is needed. The proper linking of this analysis with the later processing can be achieved by following the DOM (Document Object Model) [15] recommendation of the W3C. DOM is a set of specifications for developing standards that allow programming languages to interact with documents. DOM manipulates the whole document as a tree and XML data are provided as nodes.

The elements needed to apply the methodology can be summarized as:

- A skeleton that can be architecture dependent.
- A specification language describing the domain of the application, independent of the architecture.

– A transformer of data documents expressed in that language as instances for the skeleton.

The transformation from XML documents into parallel programs could be done directly without using the skeleton layer; however, we consider the skeleton layer necessary as a way of separating the parallel implementation, from the specification of a solution to a problem using a sequential programming language. The skeleton introduces several advantages, such as modularity, ease of use, portability, etc... The sequential skeleton can be ported to any parallel machine and the appropriate parallel code can be generated if necessary.

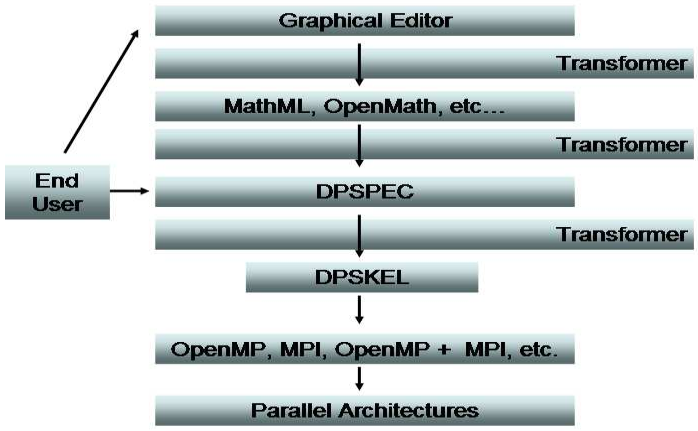


**Fig. 2.** Software Architecture for the Dynamic Programming Implementation.

Figure 2 shows how we have implemented this methodology for the particular case of Dynamic Programming problems. The automatic parallelization of DP problems is achieved by making explicit transformations on the DPSPEC data file (our specification language, described in Section 3) containing the XML description of a Dynamic Programming equation. The XML description of the formula is converted into a specific instantiation of the DPSKEL C++ skeleton (actual release of the Dynamic Programming Mallba skeleton [10]) to solve the problem in question. The C++ code generated for the Dynamic Programming skeleton is the same as would be generated by an experienced C++ programmer, so no loss of efficiency is introduced during the process. This transformation step involves an analysis of the Dynamic Programming functional equation. We have developed a DPSPEC to DPSKEL transformer. Our transformer performs a DOM parsing of the XML functional equation to produce the proper DPSKEL C++ required classes. We have not developed a general parser for mathematical equations; instead, the DOM tree is parsed so that the nodes are matched with expected values on Dynamic Programming recurrences (operators, variables, constants, etc,...). Once a value is found, the proper C++ code

is generated for it. The parser has been developed using the Xerces-C++ library [7]. Figure 3 shows a section of the parser that generates the C++ code of the Evalua DPSKEL method for a <cond> DPSPEC element. Typically, a Dynamic Programming equation is expressed as a piecewise equation, where a condition must be tested before the function is evaluated. We use the label <cond> to express such a condition. Note in Figure 3 the use of Xerces-C++ library functions, such as getFirstChild() and getNextSibling(), to traverse the DOM tree. We have developed the X2C_Exp.Exp() method to evaluate expressions recursively.

```
.... omitted code
// Generate header for the method Evalua of class State
Buffer.write("void State::Evalua(int stage, int index)\n");
Buffer.write("{\n");
Buffer.write("\tint v0, v1;\n");
Buffer.write("\tState st0(pbm, sol, table);\n\n");

 // Parse a condition
 //  <cond>
 //     #Exp0
 //     #Exp1
 //  </cond>
 //  The former element is translated as
 //  if (#Exp0) {
 //      v0 = #Exp1;
 //  }
 if (!strcmp(name, "cond")) {
     do {
        strcpy(name, (const char *)XMLString::transcode(p->getNodeName()));
        if (!strcmp(name, "cond")) {
            Buffer.write("\tif ");          // Generate if code
            X2C_Exp.Exp(p->getFirstChild());
            Buffer.write(" {\n");
            Buffer.write("\t\tv0 = ");
            X2C_Exp.Exp(p->getFirstChild()->getNextSibling());
            Buffer.write(";\n\t}\n");
         } else
            break;
         p = p->getNextSibling();
      } while(p != 0);
 } else
     X2C_Exp.Exp(p);
  // Generate the assignment of the evaluated state and the subsequent insertion in table
  Buffer.write("\tst0.setvalue(v0);\n");
  Buffer.write("\ttable.PUT_STATE(st0, stage, index);\n");
  Buffer.write("}\n\n");
  // Write the generated code to the required DPSKEL C++ data file
  if ((fevalua = fopen("dp.req.cc","w")) == NULL)
      return (-1);
  fprintf(fevalua, "%s", Buffer.buf());
  fclose(fevalua);
  return (0);
```

**Fig. 3.** DOM parser for an DPSPEC XML data file (element <problem>).

The Evalua (void State::Evalua(int stage, int index)) method, required for users of DPSKEL, includes a C++ specification of the recurrence equation. The code

in Figure 4 shows the method for the optimal matrix parenthesization (Table 1 and Equation 1). Note that the DOM parser shown in Figure 3 generates C++ code since we are interested in the specific DPSKEL C++ implementation. A different parser can be developed that generates code for any other programming language if needed.

```
void State::Evalua(int stage, int index)
{
        int v0, v1;
        State st0(pbm, sol, table);

        if (stage == index) {
                v0 = 0;
        }
        if (stage != index) {
                v0 = INT_MAX;
        for( int k = stage; k <= (index - 1); k++) {
                v1 = (table.GET_STATE(stage, k).get_value() +
                     table.GET_STATE((k + 1), index).get_value() +
                     (r[(stage - 1)] * r[k] * r[index]));
                if (v1 < v0)
                        v0 = v1;
        };
        }
        st0.setvalue(v0);
        table.PUT_STATE(st0, stage, index);
}
```

**Fig. 4.** Evalua method generated for the Optimal Matrix Parenthesization (see Table 1 and Equation 1).

However, scientists typically represent the functional equations as mathematical expressions, like those shown in Table 1, by using their favorite equation editor (Latex, OpenOffice, etc.). Therefore, a new transformation is implicitly involved in the process: the conversion of the mathematical equation to the XML specification. Currently, the transformation between a mathematical formula and an XML specification is provided automatically for many popular MathML (the W3C recommendation for describing mathematics as a basis for machine to machine communication) [17], [16], software editors, for example, EdiTex, OpenOffice, Scientific Word, sMArTH, etc. (see [18] for a detailed MathML software list). All of them generate a MathML document for a given equation. DPSPEC is compatible with MathML and the conversion between a MathML document into a DPSPEC document can be achieved through a preprocessing step.

| Problem | Recurrence | DP Category |
|---|---|---|
| 0/1 Knapsack | $f_{kc} = max\{f_{k-1c}, f_{k-1c-w_k} + p_k\}$ | Serial Monadic |
| Longest Common Subsequence | $f_{ij} = f_{i-1j-1} + 1 \;\; if \;\; x_i = y_j$; $or$ $f_{ij} = max\{f_{ij-1}, f_{i-1j}\} \;\; if \;\; x_i \neq y_j$ | Nonserial Monadic |
| Floyd's All-Pairs Shortest-Paths | $d_{ij}^k = min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$ | Serial Polyadic |
| Optimal Matrix Parenthesization | $c_{ij} = min\{c_{ik} + c_{k+1j} + r_{i-1} r_k r_j\}$ | Nonserial Polyadic |

**Table 1.** Examples of Dynamic Programming recurrences

## 3 DPSPEC: an XML specification for Dynamic Programming Problems

In this section we show the XML specification that we have developed for DP problems. Several XML specifications have been proposed to describe the semantics of mathematical expressions: MathML [17], OpenMath [13], OMDoc [12], XDF [14]. Any of them could be used to specify the DP functional equation; however, we decided to develop our own specification adapted to DP problems. The reason behind our decision was two-fold: to reduce the elements to a minimum, and to introduce some changes in the structure for specific elements appearing in classical specifications. These design issues make the subsequent parsing easier and allow for a better semantic analysis to detect data dependencies, all while adhering as closely as possible to the user defined equation. The semantic analysis determines the traversing mode of the DP table. DPSPEC brings together the elements to describe piecewise defined functions, simple variables and vectors, arithmetic, logical, relational and max-min operators, and iterators. Table 2 summarizes the elements available in the DPSPEC language. The proper syntaxis and semantics of DPSPEC have been defined as an XSD schema (omitted in the paper due to space constraints).

| Element | Operation |
|---|---|
| Main element | `<problem>` |
| Logical functions | `<and>, <or>, <not>` |
| Conditional | `<cond>` |
| Relational operators | `<lt>, <le>, <gt>, <ge>, <eq>, <ne>` |
| Binary mathematical operators | `<minus>, <divide>, <power>` |
| N-ary mathematical operators | `<plus>, <times>, <max>, <min>` |
| N-ary iterative operators | `<imax>, <imin>` |
| States | `<state>` |
| Variables | `<ci>` |
| Constants | `<cn>` |
| User defined functions | `<functiondef>, <function>` |
| User defined vectors | `<vectordef>, <vector>` |

**Table 2.** Current elements available in DPSPEC

DPSPEC has been used to specify DP problems appearing in Table 1. As an example to illustrate the use of DPSPEC, we will consider the functional equation for the multiplication parenthesization problem (Equation 1). This is a paradigmatic problem and its recurrence equation represents a wide range of Dynamic Programming applications. Note that the equation appears represented as a piecewise function. The variables $i$ and $j$ will be mapped as the variables *stage* and *index* of the method Evalua of the class State in DPSKEL. Figure 6 shows the XML specification using DPSPEC for Equation 1. The `<cond>` element allows us to describe each one of the conditions of the equation. Thanks to the availability of assistant XML editors, non-expert programmers can generate DPSPEC XML specifications more readily than sequential C programs can. Moreover, as stated in Section 2, the DPSPEC code can also be directly generated from equation editors.

$$c(i,j) = \begin{cases} 0 & \text{if j = i, } 0 < \text{i} \leq \text{n} \\ min_{i \leq k < j}\{c(i,k) + c(k+1,j) + r_{i-1} \cdot r_k \cdot r_j\} & \text{if } 1 \leq i < j \leq n \end{cases} \quad (1)$$

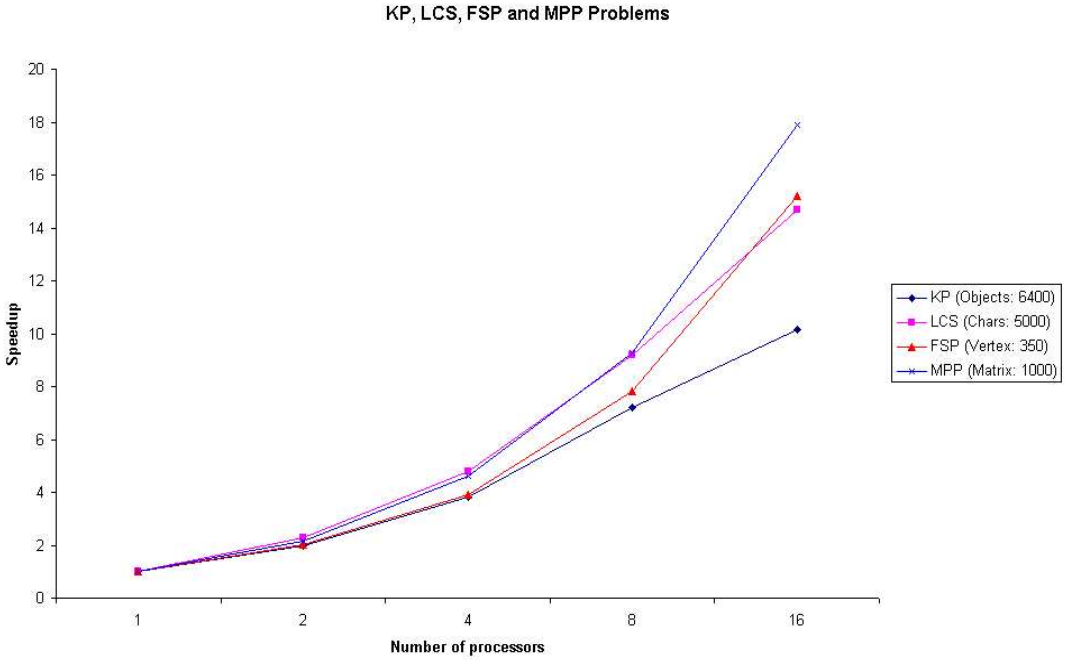


**Fig. 5.** Speedups obtained for Knapsack Problem (KP), Longest Common Subsequence (LCS), Floyd All-Pairs Shortest-Paths algorithm (FSP) and Optimal Matrix Parenthesization Problem (MPP).

The generation of the parallel code is quite straightforward: a DPSPEC document is transformed into a DPSKEL C++ instance code just by running the parser, and the DPSKEL code is compiled using the proper parallel library. Since the time invested by the parser to generate the DPSKEL code is negligible, the time invested in generating a parallel code from the XML specification is comparable to the time invested in compiling any other parallel program. No extra overhead is introduced.

In terms of the efficiency of the generated code, Figure 5 shows the performance obtained with the DPSKEL-generated parallel code. Series of instances for the problems in Table 1 were randomly generated for each problem. The instances of the skeleton generated use the OpenMP library [10] on an RS-6000 SP IBM platform. The tool's performance was satisfactory in all the cases. We observed increasing speedups in all the cases when the number of processors was increased. Superlinear speedup was observed in some cases, likely due to improved cache use on the parallel versions.

## 4 Conclusions and future work

In summary, we proposed a general methodology that allows for the automatic generation of parallel programs. The methodology is based on the existence of general parallel programs, like skeletons, that can be generated from higher level specification languages such as, for example, XML-based specifications. The code generated is efficient since no overhead is introduced during the transformation steps. The technique was validated by applying it to the generation of parallel Dynamic Programming algorithms. Once an XML specification has been stated, transformations from/to many other languages can be developed at a reasonable cost. We are also interested in the development of new transformation tools from other languages to DPSPEC, as is the case with OpenMath, for example, and from DPSPEC to other languages such as WSDL to provide the interface for a web service application.

## References


1. M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards parallel programming by transformation: The FAN skeleton framework. *Parallel Algorithms and Applications*, 16(2–3):87–122, 2001.
2. Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.
3. Marco Danelutto and Massimiliano Stigliani. Skelib: Parallel programming with skeletons in c. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1175–1184, London, UK, 2000. Springer-Verlag.
4. John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 146–160, London, UK, 1993. Springer-Verlag.

5. A. J. Dorta, J. A. González, C. Rodríguez, and F. de Sande. llc: A parallel skeletal language. *Parallel Processing Letters*, 13(3):437–448, 2003.
6. E. Alba et al. MALLBA: A library of skeletons for combinatorial optimisation (research note). In *Proceedings of the 8th International Euro-Par Conference*, volume 2400 of *LNCS*, pages 927–932, 2002.
7. The Apache Software Foundation. Xerces-c++. `http://xml.apache.org/xerces-c/`.
8. Daniel González-Morales, Francisco Almeida, F. Garcia, J. Gonzalez, Jose; L. Roda, and Casiano Rodríguez. A skeleton for parallel dynamic programming. In *Euro-Par '99: Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 877–887, London, UK, 1999. Springer-Verlag.
9. Object Management Group. Omg model driven architecture. `http://www.omg.org/mda/`.
10. Daniel González Ignacio Peláez, Francisco Almeida. High level parallel skeletons for dynamic programming. *Parallel Processing Letters*, To appear, 2006.
11. Herbert Kuchen. A skeleton library. In *Euro-Par'02: Proceedings of the 8th Euro-Par Conference on Parallel Processing*, pages 620–629, London, UK, 2002. Springer-Verlag.
12. MathWeb.org. Omdoc: Open mathematical documents. `http://www.mathweb.org/omdoc/`.
13. The OpenMath Society. Openmath. `http://www.openmath.org/`.
14. Dr. Brian Thomas and Dr. Ed Shaya. extensible data format (xdf). `http://xml.gsfc.nasa.gov/XDF/XDF_home.html/`.
15. W3C. Document object model (dom). `http://www.w3.org/DOM/`.
16. W3C. Mathematical markup language (mathml) version 2.0 (second edition). `http://www.w3.org/TR/2003/REC-MathML2-20031021/`.
17. W3C. W3c math home. `http://www.w3.org/Math/`.
18. W3C. The w3c mathml software list. `http://www.w3.org/Math/Software/`.
19. W3C. Web services description language (wsdl) 1.1. `http://www.w3.org/TR/wsdl`.
20. W3C. World wide web consortium. `http://www.w3.org/`.

```
<?xml version = '1.0' encoding = 'utf-8'?>
<problem>
  ....

  <!-- Declare a vector -->
  <vectordef load="xmlfile" type="int" name="r" vars="1" />
  <!-- c(stage, index) = {
                         0                                  if (stage = index),
              min{c(stage, k) + c(k + 1, index) + r(stage-1) * r(k) * r(index)}
                                                            if (stage <> index)
                        } -->

  <cond> <!-- 0 if (stage = index) -->
    <eq>
      <ci>stage</ci>
      <ci>index</ci>
    </eq>
    <cn>0</cn>
  </cond>

  <!-- Evalua c(stage, index) -->
  <cond> <!-- min{c(stage, k) + c(k + 1, index) + r(stage-1) * r(k) * r(index)}  if (stage <> index)} -->

    <ne>                    <-- if (stage <> index)} -->
      <ci>stage</ci>
      <ci>index</ci>
    </ne>
    <imin>                  <-- iterative min from k = stage to index - 1 -->
      <var>k</var>
      <startinterval>
        <ci>stage</ci>
      </startinterval>
      <endinterval>
        <minus>
          <ci>index</ci>
          <cn>1</cn>
        </minus>
      </endinterval>
      <plus>                <-- c(stage, k) + c(k + 1, index) + ... -->
        <state>
          <ci>stage</ci>
          <ci>k</ci>
        </state>
        <state>
          <plus>
            <ci>k</ci>
            <cn>1</cn>
          </plus>
          <ci>index</ci>
        </state>
        <times>             <-- ... + r(stage - 1) * r(k) * r(index) -->
          <vector name="r" >
            <minus>
              <ci>stage</ci>
              <cn>1</cn>
            </minus>
          </vector>
          <vector name="r" >
            <ci>k</ci>
          </vector>
          <vector name="r" >
            <ci>index</ci>
          </vector>
        </times>
      </plus>
    </imin>
  </cond>
</problem>
```



**Fig. 6.** XML description of the Multiplication Parenthesization problem using DP-SPEC