

# A declarative approach towards ensuring auto-completion and auto-update of HTML form fields

Sunil Kothari

April 29, 2006

## Abstract

Form field relationships, when modelled appropriately at the client-side, can greatly enhance the usability of new and existing forms by providing for auto-completion and auto-update of form fields without making a round-trip to the server. Traditionally, field relationships are handled exclusively at the server-side, in which case no associated JavaScript mechanisms are required, or at client-side by means of highly customised JavaScript code. Both approaches have associated pitfalls. Server-side handling of form field relationships makes such an approach server and network dependent whereas, client side handling of relationships exposes a programmer to a full programming language. Further, JavaScript has an operational form and makes a programmer think about the order in which the relationships should be used to auto-update and auto-complete. Additionally, consistent behaviour of relationships is not guaranteed as different browsers handle JavaScript differently.

We suggest an approach that allows declarative specification of form field relationships. Our approach serves the niche where the form is independently created or perhaps automatically generated by a tool and where the field relationships are added at a later stage or in some cases, separately specified. The relationships are specified in a high-level domain-specific language that exposes a programmer to only the relevant details. This helps in modular development of forms and makes forms specifications easier to read, write and maintain. The non-uniform behaviour of JavaScript is handled by translating high-level specifications to a subset of JavaScript that all browsers implement. The auto-update and auto-completion of form fields is accomplished by incorporating a rule-based mechanism that satisfies relationships through a fixed-point process. Our language is expressive enough to model a large subset of relationships that can occur in HTML forms.

We illustrate the entire methodology by a number of examples.

## 1 Introduction and Motivation

Among the many constituents of the World Wide Web, as it exists today, a very important part is the graphical user interface, mostly in the form of HTML forms. HTML forms provide a means for handling client inputs and also for handling bidirectional flow of data between the client and the server.

HTML forms typically include a number of elements specifically for handling data. These elements may be either *named* or *unnamed*. A named element requires a `name` attribute for form processing and includes fields like `input`, `textarea`, `select` etc. whereas an unnamed element is often included with a form to fulfil a special purpose. For example, `reset` is used to set all named form fields to their default values.

The named elements may again be of different types; defined by the kind of values that they are designed to handle. A `input` field of type `text` is designed to take a single word or line of text whereas a `input` field of type `radio` is designed to reflect a single choice among a number of choices, a `input` field of type `checkbox` is designed for a situation when more than one answer is acceptable.

Further, most field values are visible to the user. But, a `input` field of type `hidden` is never visible to the user or a field of type `password` where the user input is visible in a form that reveals nothing about the field value. The number of fields, type of a field, the nature of values a field can be assigned, are all design-time decisions taken when the form is first conceptualised.

Also, known at design-time, though seldom modelled, are the underlying *relationships* amongst form fields. These relationships are implied at all times; from the moment the HTML page (containing the form) is loaded by the browser to the moment the form is submitted to the server. Traditionally, these relationships are handled at the server-side which we will argue is neither good for the server traffic nor for the network bandwidth. We take a simple example to make our point. Consider an online purchase order application where a user is encouraged to browse and choose items he or she would like to purchase. Once the user has

### Purchase Order Form

Item No.	Article Description	Quantity	Price per Item	Cost
1	Microsoft Office Suite	1 <input type="button" value="Update"/>	6	
2	Microsoft SQL Server	1 <input type="button" value="Update"/>	2	
3	Corel Draw Version 6.0	2 <input type="button" value="Update"/>	7	
Total Quantity		4	Total Price	15

Figure 1: A typical purchase order form.

chosen the items to be purchased, the total price and the total quantity are computed by the server and a new page with price and quantity details (along with the users chosen item) is shown to the user as shown in Figure 1. A user still has the option to change his order details by changing the quantity of the chosen items.

If a user now tries to update one of the quantities then he is explicitly asked to request the server to update the corresponding details about the total price, discount and so forth. The server essentially *re-computes* a new total cost and a new total quantity on the basis of changed field quantity and sends a new page to the client with the updated details. Such an approach is highly server-centric and relies heavily on the assumptions:

- that the round-trip to server is relatively fast; and
- that server is available (at all times) to process the client requests.

Moreover, such an approach has other side-effects.

1. The client is annoyed by the delay in response caused due to the round-trip to server which is essentially re-computation of the existing details.
2. The re-computation process leads to less server availability for other incoming requests.

Moreover, the service programmer has to do extra programming to take care of this re-computation and re-show of pages.

These round-trips to the server and the notion of all-time server availability could have been avoided if the the form field relationships were handled at the client itself. But, frequently that is not the case. This could be attributed to the fact that firstly, there is no standard for specifying such relationships and secondly, the handling of relationships on the client side is fraught with inconsistent behaviour due to the non-uniform handling of JavaScript (see Appendix A.1). The advantage of client-side handling of relationships is that it opens up the possibility of auto-completion of unfilled form fields and auto-update of filled form fields without explicitly contacting the server. We also conjecture that such an approach can also be used to give meaningful and precise error messages *incrementally*, and not when the form is submitted to the server. We have not investigated that in our report but we believe such a thing is feasible.

The simplicity of relationships justifies a purely *declarative* approach. This is in complete contrast to programming in JavaScript where a programmer is exposed to low-level details and has to think about the order in which these relationships are handled. A high level description of such relationships in a domain-specific language will expose a programmer to only the relevant details. This results in specifications which are easier to read, write and maintain. Further, a high-level approach encourages modular development of the form where a form is designed independently or perhaps generated automatically by a tool and the relationship specifications are later added or even specified separately.

Furthermore, since web browsers do not support JavaScript in a uniform manner (Appendix A.1), the generated code from such relationships must be targeted to a subset of JavaScript that all browsers implement. For many cases this subset is not easy to find even by experienced JavaScript programmers. A tool that automatically translates these high-level descriptions to this common subset is therefore much desired; this means that only the compiler writer is concerned with such an issue.

Additionally, these specifications should be expressible in a XML-like language so as to easily integrate with other XML-tag based form technologies such as *Scalable Vector Graphics* (SVG) [13] and *Synchronised Multimedia Integration Language* (SMIL) [14].

We summarise our requirements as:

- the relationship specifications should be declarative;
- the specifications should be expressed in a high-level domain-specific language and should integrate well with XML; and
- the implementation should not be browser dependent.

The rest of this report is organised as described below. Section 2 gives a background on the techniques and concepts involved. Section 3 describes the related work. Section 4 describes the auto-completion mechanism. Section 5 describes some applications of the above formalism. Section 6 describes the various properties of a well-behaved mechanism. Section 7 describes the implementation details. Finally, Section 8 gives an overview of the future work.

We digress here to make explicit the distinction between relationships and constraints in context of a HTML form. A field may be *related* to other fields by an algebraic binary function as ‘+’, or it may take the form of an arbitrary n-ary function. On the other hand, a field value can be *constrained* to take only integer values for example, between 1 and 1000. Thus, constraints are defined at a *micro* level (on a field value) whereas relationships are defined at a *macro* level (amongst form field values). In this report, we will intermittently use the term constraints to mean relationships. When we want to imply constraints on field value we will refer constraints as field validation formats or simply field formats.

## 2 Background

We highlight some of the underlying concepts and technologies on which our implementation depends.

### 2.1 JavaScript Programming

JavaScript has been widely used for adding dynamism to static HTML pages. These range from changing the font of a given link to managing HTML page content in different frames. The power of JavaScript lies in the fact that its a general purpose programming language that can simultaneously handle among others: HTML document content, user interaction with the browser.

#### 2.1.1 Controlling HTML Document

JavaScript has a set of objects that allow it to interact with the HTML document elements and form contents in particular, through an interface known as DOM (Document Object Model) [8]. This interface provides

methods and properties to retrieve, and modify HTML elements including HTML form fields. Specifically, JavaScript objects can read from one form field and write its value to another form field.

### 2.1.2 Handling User Interaction

For graphical user interfaces like HTML forms, a user action is handled by invocation of *event-handlers*. Each event-handler is a call to a chunk of JavaScript code that is either in-lined with the HTML elements or defined separately between `<script>` `</script>` tag. For example, when a user points to an image of a world map in a HTML document a pop up suggests the country name at which the user is currently pointing to.

## 2.2 The Need for Domain-Specific Languages

But, being a general purpose language, many users find JavaScript too intimidating even for trivial tasks such as validating a field. Besides, heavy use of scripting language makes HTML pages less maintainable and relatively trivial changes require considerable human effort. A simple change like considering discount field in a purchase order form, as mentioned earlier, involves a clear understanding of the scripts that handle code for form field validation and scripts that handle code for interaction among the various form widgets. Many specialised libraries exist [27] but they still expose the user to a full programming language. Thus, high-level domain-specific languages, that can handle each of these tasks separately, are very much desirable. W3C working draft [3] on extending forms has stated this objective as:

”It should be possible to define rich forms, including validations, dependencies, and basic calculations without the use of a scripting language.”

## 2.3 PowerForms Language

PowerForms [7] is one such language for declarative specification of field formats and interdependencies in a XML-like language that has been designed for incremental input validation. The format specification for a named textual field is expressed as a standard regular expression. More information about PowerForms can be found in [10] [26]. This expression is then converted to a minimised deterministic finite automata and the states of the automaton are used to show the validity status of a field. The field with the associated automata is annotated with a traffic light icon displaying red, yellow or green light corresponding to whether the automaton is in crash, reject or accept state when run on the field value as input. Formally, let  $L$  denote the set of strings accepted by the finite automata associated with a particular field format and  $v$  be the current field value the three possible states are described as:

- *Valid* :  $v \in L$  (shown by a green light)
- *Valid* :  $\exists w: vw \in L \wedge |w| \neq 0$  (shown by a yellow light)
- *Invalid* : otherwise (shown by a red light)

A format for a field that can only accept 3-digit positive integers can be specified as:

```
< constraint field="B1">  
  < repeat count="3">  
    < charrange id="number" low="0" high="9"/>  
  </ repeat>  
</ constraint>
```

Figure 2 shows a form field for which valid values are field values that conform to the above format specification. This field when given a decimal value is shown *invalid* by means of a red light.

Any auto-completion mechanism should ensure that auto-completed fields have values that conform to the specified field formats. For instance in our purchase order example we would not like to deduce price if a user enters a negative item quantity. The traffic lights also makes a user aware that a field has been auto-completed by reflecting the change in states of the associated automaton as a result of auto-completion.



Figure 2: Field validation in PowerForms.

## 2.4 Local Propagation and Rule Triggers

In constraint world, relationships are modelled as constraints. Constraints allow the user to declare a set of relations on a set of objects and the task of finding the values (that satisfy these constraints) is left to a constraint solver. A constraint solver satisfies the constraints by a number of different constraint satisfaction techniques [9].

A commonly used technique for constraint satisfaction, Local Propagation [11], assumes constraints can be modelled as a graph where nodes represent operators and edges represent the possible flow of values. When a node receives sufficient value from its edges, it *triggers*, calculates one or more values for the edges that do not contain values and sends these new values out. These new values in turn may cause other nodes to trigger but the current node is unaware of such future triggers. Figure 3 describes the possible rules for one such node that models addition of two numbers.

```
plus ( a, b, c)
rule : a ← c - b ( c and b are known and a is unknown)
rule : b ← c - a ( c and a are known and b is unknown)
rule : c ← a + b ( a and b are known and c is unknown)
```

Figure 3: Rules for a plus node.

Note here the rule triggers are local to each node and only involve information that is contained on the edges that connect to this node. Inspired by this approach of deducing unknown values from known values, we will base our formalism to reflect how relationships can deduce unknown form field values from known form field values by means of rule triggers.

## 2.5 Restricting Rule-triggers

A general rule based mechanism, as illustrated above, is clearly not the right choice in HTML forms where user action can trigger event handlers. We elaborate this further with the help of an example. Consider again the online purchase order form as shown in Figure 1. The relationships amongst the form fields can be summarised as:

- total cost is the *sum* of individual costs;
- total quantity is *sum* of individual quantities; and
- cost is a *product* of price and quantity.
- total cost is always *deduced* from individual costs; and
- total quantities is always *deduced* from individual quantities;

To make the total quantity reflect the change in any of the item quantities we make the field apparent i.e. change its types from `hidden` to `text`. Notice that we have *not* introduced any new fields in this process. If our approach was to mention the above relationships as suggested in the previous section we will get aberrant behaviour especially in following scenarios:

- A user might try to change the total quantity field;

Clearly, we need to disable user action for such fields a field like total quantity is *always* deduced from other fields. So, it should be *read-only*. This scenario is interesting because even though we disable user action we can still use DOM to assign values to this field. The effect of such a restriction is that it reduces the possible number of rule triggers. We will incorporate this provision to disable user actions on certain fields and still retain the flexibility offered by the rule based mechanism.

### 3 Related Work

Since our contributions are towards modelling relationships in HTML forms we survey the existing methodologies and work in this area and avoid issues like presentation details and user interface aspects of HTML forms.

W3C [1] has proposed XForms [6] as the next generation of web forms. The central idea in XForms is to separate *purpose* from the *presentation* of the form. XForms relies on the abstraction of form to two distinct areas :

- **Form Description(Xforms Model)**: defines structure of the instance data, describes constraints on the instance data (validation specification), and to a certain extent, describes computational relationships. XForms Model consists of two components: *Instance data* and *Form Logic*.
  - **Instance Data** : describes form data as an XML tree. Among other things, a user may modify the instance data by client side interaction. Consider our purchase order example. A typical instance of an item could be described as:

```

<xforms:Instance>
  <order>
    <items>
      .....
      <item name="itm1">
        <quantity>1</quantity>
        <description>Microsoft SQL Server</description>
        <price>100</price>
        <cost>100</cost>
      </item>
      ...
    </items>
  </order>
</xforms:instance>

```

- **Form Logic** :defines event handlers, constraints on the instance data (including the computational relationships) and submission information. The model item definitions are bound to the instance data using XPath expression in the `ref` attribute. The price, quantity relationship for our example can be described in terms of XPath [5] expressions as:

```

<xforms:bind ref="order/items/item/cost"
  calculate="../quantity * ../price"
  relevant=" ../quantity > 0" />

```

The above XPath expression bind the cost node to be a product of quantity and the price. Notice here that the above binding is valid only if the quantity is greater than zero. This kind of conditions on field values, in our case, are reflected in the field format specification as shown in Section 2.3.

- **Form Presentation:** defines how the form is shown and also expresses bindings to instance items. The quantity and price fields can be specified as:

```

.....
<tr>
  <td><xforms:textbox ref="order/items/item/quantity"/></td>
  <td><xforms:output ref="order/items/item/price"/></td>
</tr>
.....

```

The `ref` attribute specifies that the input data should be bound to a node `price` in instance data. The `output` tag refers to the fact that price is *read-only*.

Various vendors have tried to make a working implementation of XForms based on W3C recommendations. A comprehensive list is available at the XForms homepage [6]. These either lack the feature of modeling relationships or they model very simple calculations. Moreover, all these vendors have *platform-specific* implementations which we believe is a severe limitation. Note that one of the key things in our implementation is that we make *minimum* assumption about the browser and the platform.

We discuss the approach taken by Honkala and Boyer [12] and discuss their implementation of XForms. Their implementation is important because it gives a comprehensive view of how computational relationships between the form fields can be used to do auto-updates especially in context of XForms. We ignore the specific technical details regarding other Xforms features but focus on the *xforms-recalculate* event [4]. Briefly, a recalculate event is triggered when a user changes an instance data. First, we define some terminologies that we will use later. The underlying notion of relationships in XForms is expressed as a graph : *Master Dependency Directed graph* and *Pertinent Dependency subgraph*.

A Master Dependency graph M is a directed graph consists of a set of vertices V and a set of edges E that represent the computational dependencies between vertices. It is constructed when the page is first loaded in the browser and remains unchanged till the browser eventually unloads the page.

A Pertinent Dependency Graph is obtained from master graph and is invoked when the recalculation algorithm detects a change in instance nodes and is obtained by exploring the paths of edges and vertices that are reachable from the changed vertex.

The implementation by Honkala and Boyer makes explicit use of topological sorting (to determine which field values will be calculated next ) in implementing the recalculation algorithm. Also, they rely heavily on the notion that it is possible to deduce Pertinent graph from a Master graph whenever a user modifies a field. Since XForms uses XPath expressions to access the instance data, their implementation suffers from the limitations of XPath itself. For example, XForms forbids use of XPath constructs that result in a change of node-set reference within a XPath expression based on any changes to the instance data. Also, for very complex forms, building of a pertinent graph, every time a user inputs data, is a time intensive operation and may not lead to instantaneous updates. Moreover, Xforms requires a Xforms processor on the client.

In contrast, our approach uses a simple rule-based formalism where the dependencies between the various relationships are settled through a fixed-point process. Such an approach obviates the need for maintaining explicit information about the dependencies and doing a topological sort every time a field is changed. Further, our formalism is more expressive as it can handle **if-then-else** kind of relationships. The explicit use of guards gives a very clean semantics to if-then-else construct in our language.

Abdualrraouf and Ridley [21] have suggested a way of disallowing certain form field values for forms which collect information that is stored in a database. They conjecture that meta information stored in a database system can be extracted in a useful manner and then used to restrict values in form fields that violate this meta information. Clearly, their work targets a very small subset of HTML forms. Moreover, their work is still at a conceptual stage and lacks a full implementation.

Another implementation of client-side handling of form field relationships in XML is XFDL, *extensible form description language* [29]. The implementation lacks declarative specifications and a programmer has

to specify exactly what should happen when some field is given a input. Moreover, their implementation is rigid and requires platform specific browsers that support Java runtime environment.

To the best of our knowledge, no one has worked before on auto-completion of and auto-update of HTML form fields using a fixed-point approach of satisfying relationships.

## 4 Auto-completion and Auto-update Mechanism

We now give precise meaning as to how relationships can be used to predict form field values.

### 4.1 Terminology

We introduce the concepts and the terminology needed for further discussion on auto-completion and auto-update mechanism.

#### 4.1.1 Known and Unknown Fields

Let  $\mathbb{W}$  denote all named field values in a form and  $\mathbb{F}$  denote a set of all form fields. Let  $\mathbb{V} \subseteq \mathbb{W}$  denote a collection of named HTML form fields that have non-blank values and are accessible from DOM. A field  $v$  is *known* iff  $v \in \mathbb{V}$ . Conversely, a field is *unknown* if  $v \notin \mathbb{V}$ . Let  $\mathbb{Z}$  be the collection of named field values and let  $val: \mathbb{V} \rightarrow \mathbb{Z}$  be a function such that it maps a named field to its value then,

$$v \equiv \begin{cases} known & \text{if } val(v) \neq null \\ unknown & \text{otherwise} \end{cases}$$

#### 4.1.2 Predicate

A set of fields  $V \subseteq \mathbb{F} = \{v_1, v_2, \dots, v_n\}$  that satisfy a certain relationship  $\mathbb{R}$  is grouped as *predicate* and is denoted by  $P(v_1, v_2, \dots, v_n)$ .

#### 4.1.3 Deduced Field

To derive any of the unknown values that constitutes a relationship the following condition should be satisfied:

$$\exists v \in V : v \notin \mathbb{V}$$

which states that there exists at least one field that is a part of the relationship but its value is still unknown. A field value that is derived on the basis of the given relationship is known as *deduced* field value and the corresponding field is known as *deduced* field.

#### 4.1.4 Check-Code and Mode-Check

On the other hand, if all the fields are known then it can be verified whether the relationship holds for the given predicate. This verification can be done by running appropriate piece of code known as *check-code*  $C$ . A check-code  $C$  for a predicate  $P$  is evaluable if and only if its mode  $\mathcal{M}_c$ , known as *mode-check*, is satisfied. The mode  $\mathcal{M}_c$  is a collection of fields that comprise the relationship  $\mathbb{R}$ . Evaluation of check-code  $eval_c$  can be seen as a mapping

$$eval_c : C \rightarrow \mathbb{B}$$

#### 4.1.5 Auto-Completion Rules and Mode-Rule

To deduce unknown field values from known values, we associate with every predicate  $P$ , a set of rules denoted by  $\mathcal{A} = \{R_1, R_2, \dots, R_n\}$ . Each rule  $R$  has a set of fields which must be known if the rule is to *trigger* and deduce unknown field value. This set of known field values is collectively termed as *mode-rule* and is denoted by  $\mathcal{M}_r$ .

### 4.1.6 Guard and Mode-guard

A rule is *guarded* against triggers by an optional guard  $G$ . A guard makes a rule trigger *conditional* to fulfilment of certain conditions. These conditions are represented as a boolean expression tree  $T$  with non-terminal nodes as boolean operators and terminal nodes as calls to JavaScript functions as shown in Figure 4. In addition, every guard has a collection of fields known as mode-guard  $\mathcal{M}_G$  such that a guard will be evaluated if and only if all the fields comprising mode are known. The evaluation of guard  $eval_g$  is defined in terms of the associated boolean expression tree  $T_G$  as:

$$eval_g \equiv eval_{T_G} : T_G \rightarrow \mathbb{B}$$

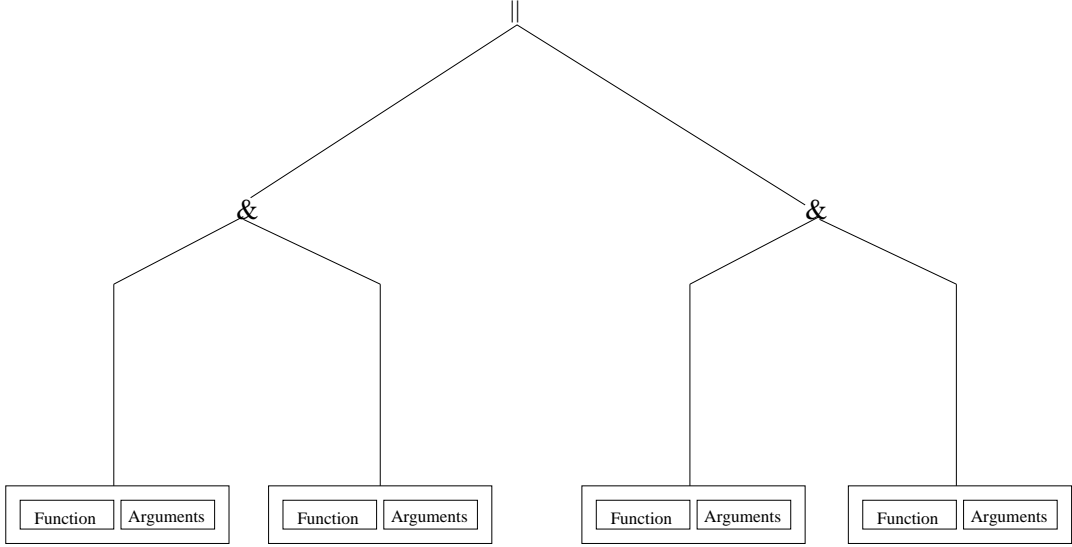


Figure 4: A guard is represented as an expression tree with non-terminal nodes as boolean operators.

### 4.1.7 Auto-completed and Auto-updated field

A deduced field  $f \in \mathbb{F}$  is said to be *auto-completed* if it is unfilled initially. Similarly, a deduced field  $f$  is said to be *auto-updated* if it already has a value and the deduced value is different from its previous value.

## 4.2 A Small Example

With these notion of check-code, auto-completion rules and associated guards we can now model a given predicate. Consider a HTML form involving three `input` fields of type `text`: `p`, `q` and `r`. These fields satisfy the relationship  $\mathbb{R}$  given by a predicate  $P(p,q,r)$ . The relationship is defined by the corresponding check-code  $C$ :

$$C \equiv val(p) == val(q) / val(r)$$

A set of rules can now be given to model how a unknown field value can be deduced from a set of known field values. The complete set of rules and the corresponding modes and guards for the predicate  $P$  is shown in Figure 5.

```

        divide ( p, q, r)
        mode-check(p, q, r) check-code( p == q / r )
        Autocompletion Rules
mode-guard( r )   guard( r != 0 )   mode-rule( q, r ) rule : p ← q / r
mode-guard( p )   guard( p != 0 )   mode-rule( r, p ) rule : q ← r * p
mode-guard( p )   guard( p != 0 )   mode-rule( q, p ) rule : r ← q / p

```

Figure 5: A divide predicate with check-code and auto-completion rules and associated guards

### 4.3 Modeling Language: Syntax and Semantics

We now describe the various constructs in our language and give a brief description of its intended semantics. We have used the following convention for our grammar namely, ‘\*’ indicates zero or more repetitions, ‘+’ indicates one or more repetitions and ‘—’ indicates a choice and attributes enclosed in ‘[’ and ‘]’ are optional. The complete grammar specification is also available in Appendix A.2 for reference.

```

predicatedef      → ipredicate id = stringconsti
                    predicate-body
                    i/predicatei
predicate-body   → ibodyi
                    argumentlist
                    checkcode
                    autocompletionrules
                    i/bodyi
                    | itemplatei
                    argumentlist
                    checkcode
                    autocompletionrules
                    i/templatei
                    | ibind [idref=stringconst] [url=stringconst]i
                    binding-list*
                    i/bindi
                    | iifi
                    predicate-expression
                    ithenipredicate-refi/theni
                    [ielseipredicate-refi/elsei]
                    i/ifi
predicate-expression → predicate-ref
                    | iandi
                    predicate-expression
                    predicate-expression
                    i/andi
                    | iori
                    predicate-expression
                    predicate-expression
                    i/ori
                    | inoti
                    predicate-expression
                    i/noti
predicate-ref      → ipredicate idref=stringconst/i

```

Any auto-completion specification consists of a number of predicate definitions. A predicate can be defined from scratch by defining each of its individual components like argument-list, check-code and auto-completion rules as shown by **body** element or it be a template for other predicates denoted by **template** element. The argument-list for a template serves as a placeholder for the template variables. It is possible

to deduce an argument-list at compile-time but it is prone to errors. By making the argument-list explicit we can ensure that there are no template variables that remain uninstantiated. The **bind** element denotes a predicate which is defined by instantiating the referred template (given by **idref** attribute) and binding the arguments specified in the template's argument-list to concrete HTML form field values . A template can also be stored at a different place and can be accessed over the Internet by specifying **url** attribute of the **bind** tag. The predicate can also be formed by supplying appropriate predicate references to the IF-THEN-ELSE construct. We will describe the intended semantics of this construct later in Section 4.4. It suffices to say for the moment that the non-terminal *predicate-expression* is used to specify boolean expression trees which acts as guards to the auto-completion rules of predicate references given by **then** and **else** tags. The **predicate-ref** element denotes a reference to a predicate by means of **idref** attribute. Any predicate is a 3-tuple:

$$P = ( A, C, \mathcal{A} )$$

where,

- A is a set of *fields* that form the predicate,
- C is the *check-code*, and
- $\mathcal{A}$  is a finite set of *rules*.

$$\begin{array}{ll}
 \textit{binding-list} & \rightarrow \textit{;to}_i \\
 & \textit{argument} \\
 & \textit{assignment} \\
 & \textit{i/to}_i \\
 \textit{assignment} & \rightarrow \textit{;field name=stringconst/i} \\
 & | \textit{;const value= intconst/i} \\
 & | \textit{;interface id=stringconst name=stringconst /i} \\
 \textit{argument} & \rightarrow \textit{;argument name=stringconst/i}
 \end{array}$$

The **to** element denotes a list of bindings. The **field** element denotes a field name. The **const** element denotes a constant value. The **interface** element denotes a link to another predicate given by **id** attribute and the **name** attribute. The **argument** element denotes a field or a function name. The template is instantiated by binding the concrete form field values with the arguments in the template's argument-list in a copy of the referred template. Each template binding is a list  $B = B_1, B_2, \dots, B_n$  of pairs:

$$B = (A_1, A_2)$$

where,

- $A_1$  is a *name* of the argument of the template to which  $A_2$  will bind to, and
- $A_2$  is a *link* to another predicate (given by **interface** element ),
- or a constant (given by **const** element) or a named form field (given by **field**).

$$\begin{array}{ll}
 \textit{checkcode} & \rightarrow \textit{;checkcode}_i \\
 & \textit{operator} \\
 & \textit{result} \\
 & \textit{argumentlist} \\
 & \textit{i/checkcode}_i \\
 \textit{operator} & \rightarrow \textit{;operator name =stringconst /i} \\
 \textit{result} & \rightarrow \textit{;result name =stringconst/i} \\
 \textit{argumentlist} & \rightarrow \textit{;argumentlist}_i \\
 & \textit{argument}^+ \\
 & \textit{i/argumentlist}_i
 \end{array}$$

The **checkcode** element denotes the check-code  $C$  for a given predicate. The **operator** element denotes the function name. The **result** element denotes the field name and the element **argumentlist** denotes one or more arguments. The argument nonterminal has already been explained above. A check-code  $C$  is a four-tuple:

$$C = ( \mathcal{M}_c, A_c, f_c, r_c )$$

where,

$\mathcal{M}_c$  is a set of *known fields* required for check-code evaluation,  
 $A_c$  is a list of *arguments* to the function,  
 $f_c$  is a *name* of the function, and  
 $r_c$  is a *name* of the result field.

The semantics of check-code  $C$  is that the function  $f_c$  is evaluated with  $A_c$  as the list of arguments. The evaluated value is checked with the value of the result field  $r_c$ . If they are the same then check-code returns true and false otherwise. The evaluation  $eval_c$  of check-code  $C$  can be formalised as:

$$eval_c(C) \equiv \begin{cases} true & \text{if } \exists v \in \mathcal{M}_c : v \notin \mathbb{V} \\ apply_c(f_c, A_c, r_c) & \text{iff } \forall v \in \mathcal{M}_c : v \in \mathbb{V} \end{cases}$$

$$apply_c(f_c, A_c, r_c) \equiv \begin{cases} true & \text{if } eval(f_c)(A_c) = val(r_c) \\ false & otherwise \end{cases}$$

**eval** is defined later in section 7. The mode  $\mathcal{M}_c$  is not a part of the language syntax as it is automatically deduced <sup>1</sup> by the compiler from the argument-list  $A_c$  and the result field  $r_c$ .

For primitive predicates, it might seem like redundant information to be carried around but in case of composite predicate such information is necessary as the argument-list  $A_c$  might consist of functions that the compiler generated itself.

$$\begin{array}{ll} autocompletionrules & \rightarrow \mathbf{j}autocompletionrules_i \\ & \quad autocompletionrules-body \\ & \quad \mathbf{i}/autocompletionrules_i \\ autocompletionrules-body & \rightarrow \mathbf{j}empty/i \\ & \quad | \quad rule^+ \\ rule & \rightarrow \mathbf{j}rule_i \\ & \quad argumentlist \\ & \quad guard \\ & \quad result \\ & \quad operator \\ & \quad \mathbf{i}/rule_i \end{array}$$

The **autocompletionrules** element denotes a set of rules. The **rule** element denotes an auto-completion rule. In some cases, there may be no rules and is denoted by **empty** tag. A **rule** is given by the *argumentlist*, *guard*, *result* and *operator* nonterminals. Formally, auto-completion rules  $\mathcal{A}$  is a set of rules:  $\mathcal{A} = \{ R_1, R_2, \dots, R_n \}$ . Each rule  $R$  is a five-tuple:

$$R = ( G, \mathcal{M}_r, f_r, A_r, r_r )$$

where,

$G$  is a *guard* for the rule;  
 $\mathcal{M}_r$  is a set of *known fields* needed for rule trigger,  
 $f_r$  is a *name* of the function,  
 $A_r$  is a list of *arguments* to the function, and  
 $r_r$  is a name of the *field* that receives the value from rule trigger.

---

<sup>1</sup>only for primitive predicates

The semantics of rule is that if the guard  $G$  is satisfied and mode-rule  $\mathcal{M}_r$  consists of known fields and the field  $r_r$  is unknown, then a rule trigger can be defined as evaluating function  $f_r$  with  $A_r$  as the argument-list that returns a value to be assigned to the field represented by  $r_r$ . A rule trigger  $trigger_R$  for a rule  $R$  can then be defined as:

$$trigger_R \equiv \mathbf{eval}(f_r)(A_r) \text{ iff } \{ \forall v \in \mathcal{M}_r : v \in \mathbb{V} \} \cap r_r \notin \mathbb{V}$$

$$\begin{array}{lcl} guard & \rightarrow & \mathbf{jguard}_i \\ & & \quad guard\text{-body} \\ guard\text{-body} & \rightarrow & \mathbf{i/guard}_i \\ & | & \mathbf{i/empty}_i \\ & & \quad argumentlist \\ & & \quad operator \end{array}$$

A **guard** element denotes a guard. The nonterminal *operator* and *argumentlist* have already been explained above. A guard  $G$  is a three-tuple:

$$G = (\mathcal{M}_g, f_g, A_g)$$

where,

$\mathcal{M}_g$  is a set of *known fields* needed for guard evaluation,  
 $f_g$  is the *name* of the function, and  
 $A_g$  is a list of *arguments* to the function.

Before a guard  $G$  is evaluated, the mode  $\mathcal{M}_g$  is checked. If the fields are known then the guard is evaluated by evaluating the associated expression tree  $T_G$  and is defined as:

$$eval_g(G) \equiv \begin{cases} true & \text{if } T_G = null \\ false & \text{if } \exists v \in \mathcal{M}_g : v \notin \mathbb{V} \\ eval_{tree}(T_G) & \text{iff } \forall v \in \mathcal{M}_g : v \in \mathbb{V} \end{cases}$$

Additionally, fields that are read-only are specified by the following syntax:

$$autoupdate\text{-def} \rightarrow \mathbf{jautoupdate name=stringconst status=stringconst/}_i$$

Here, the **autoupdate** element denotes information about a field whose name is given by **name** attribute and its status is given by a **status** attribute. There are instances, as described in Section 4.6 when some additional information also needs to be included for auto-update of fields. That information can be specified as:

$$metainfo \rightarrow \mathbf{jmetainfo update=stringconst ignore=stringconst/}_i$$

The **metainfo** element denotes a relationship between two fields given by **update** and **ignore** attributes. If a form has very complicated relationships additional JavaScript functions can be included by specifying a file name in which the functions have been coded. The predefined functions are included in the specifications by the following syntax:

$$include\text{-file} \rightarrow \mathbf{jinclude name =stringconst/}_i$$

where, the **include** element denotes a file whose name is given by **name** attribute.

## 4.4 Composite Predicates

Sometimes, it is possible to define predicates from other primitive predicates by using a *IF-THEN-ELSE* construct. Note that each predicate is defined as a three-tuple  $\langle A, C, \mathcal{A} \rangle$ . An abstract syntax of if-then-else construct can be described as:

$$P_r \equiv \text{if } P_f \text{ then } P_t \text{ else } P_e$$

where,

$$\begin{aligned} P_r &= ( A_r, C_r, \mathcal{A}_r ), \\ P_f &= ( A_f, C_f, \mathcal{A}_f ), \\ P_t &= ( A_t, C_t, \mathcal{A}_t ), \text{ and} \\ P_e &= ( A_e, C_e, \mathcal{A}_e ). \end{aligned}$$

The argument-list  $A_r$  of the composed predicate  $P_r$  is given by:

$$A_r \equiv A_f \cup A_t \cup A_e$$

Any evaluation of check-code  $C$  is a mapping  $eval_c: C \rightarrow \mathbb{B}$ .

Therefore, the evaluation of check-code  $C_r$  of the composed predicate  $P_r$  is defined as :

$$eval_c(C_r) \equiv \begin{cases} eval_c(C_t) & \text{if } eval_c(C_f) = true \\ eval_c(C_e) & \text{if } eval_c(C_f) = false \end{cases}$$

The auto-completion rules  $\mathcal{A}_r$  of the resultant predicate  $P_r$  can be defined as:

$$\mathcal{A}_r \equiv \mathcal{A}_t \uplus \mathcal{A}_e$$

where,

$$\begin{aligned} \mathcal{A}_t &= \{ R_{1t}, R_{2t}, \dots, R_{nt} \}, \\ \mathcal{A}_e &= \{ R_{1e}, R_{2e}, \dots, R_{me} \}, \text{ and} \\ \mathcal{A}_r &= \{ R'_{1t}, R'_{2t}, \dots, R'_{nt}, R'_{(n+1)e}, \dots, R'_{(n+m)e} \} \end{aligned}$$

The operation  $\uplus$  is a union of auto-completion rules that are transformed as described below.

For  $\mathcal{A}_t$  the transformation is defined as :

$$\mathbb{T}_t : R_t \Longrightarrow R'_t$$

Consider a rule  $R_t = ( G_t, \mathcal{M}_t, f_t, A_t, r_t )$  and the transformed rule  $R'_t = ( G'_t, \mathcal{M}_t, f_t, A_t, r_t )$ . It is clear from above that only the guards are affected by the transformation  $\mathbb{T}_t$ . Let  $T_{G_t}$  and  $T_{G'_t}$  denote the expression tree associated with guard  $G_t$  and  $G'_t$  respectively, then the transformation on guards is given as:

$$T_{G'_t} \equiv \begin{cases} maketree(C_f) & \text{if } T_{G_t} = null \\ maketree(C_f) \circ T_{G_t} & \text{otherwise} \end{cases}$$

where ,

*maketree*:  $C \rightarrow T$  creates a new expression tree. The binary operator  $\circ$  returns a new expression tree with a boolean operator **and** as a non-terminal node and operands(to the function) as children to this node. This transformation ensures that all calls to JavaScript functions are still in the terminal nodes.

Similarly, consider a rule  $R_e = ( G_e, \mathcal{M}_e, f_e, A_e, r_e )$  and the transformed rule  $R'_e = ( G'_e, \mathcal{M}_e, f_e, A_e, r_e )$ . Here again only the guards are affected by the transformation  $\mathbb{T}_e$ .

For  $\mathcal{A}_e$  the transformation is  $\mathbb{T}_e$  defined as :

$$\mathbb{T}_e : R_e \Longrightarrow R'_e$$

Let  $T_{G_e}$  and  $T_{G'_e}$  denote the expression tree associated with guard  $G_e$  and  $G'_e$  respectively, then the transformation on guards is given as:

$$T_{G'_e} \equiv \begin{cases} maketree(C_{f'}) & \text{if } T_{G_e} = null \\ maketree(C_{f'}) \circ T_{G_e} & \text{otherwise} \end{cases}$$

where,

the function *maketree* and the operator  $\circ$  have already been described above.

The relation between  $C_f$  and  $C_{f'}$  is given by:

$$eval_c(C_f) \equiv \begin{cases} true & \text{if } eval_c(C_{f'}) = false \\ false & \text{otherwise} \end{cases}$$

Based on the above transformations, Figure 6 shows a concrete example of a predicate being composed from a number of primitive predicates. We can generalise the condition part of IF-THEN-ELSE construct from a

If lessThan(c , g ) then divide(p,q,r) else divide(a,b,c)

lessThan (c, g)			
mode-check(c , g)	check-code(c , g)		
divide ( p, q, r)		divide ( a, b, c)	
mode-check( p, q, r)	check-code ( p == q / r )	mode-check(b, a, c)	check-code ( b == a / c )
mode-guard(r)	guard( r != 0 )	mode-guard(b)	guard( b != 0 )
	mode-rule( r , q )	mode-rule( a , b )	rule : c ← a / b
	mode-rule( r , p )	mode-rule( b , c )	rule : a ← b * c
mode-guard(p)	guard( p != 0 )	mode-guard(c)	guard( c != 0 )
	mode-rule( p , q )	mode-rule( a , c )	rule : b ← a / c
	rule : P ← q / r		
	rule : Q ← r * p		
	rule : r ← q / p		
If-else-then ( a, b, c, g, p, q, r)			
mode-check( r, c, g, a, b, c, p, q)	check-code ( (c < g) !=> (p == q / r) ^ (b == a / c))		
mode-guard( r, c, g)	guard( r != 0 && c < g )	mode-rule( r , q )	rule : p ← q / r
mode-guard( c, g)	guard( c < g )	mode-rule( r , p )	rule : q ← r * p
mode-guard( p, c, g)	guard( p != 0 && c < g )	mode-rule( p , q )	rule : r ← q / p
mode-guard( b, c, g)	guard( b != 0 && !(c < g) )	mode-rule( a , b )	rule : c ← a / b
mode-guard( c, g)	guard(!(c < g))	mode-rule( b , c )	rule : a ← b * c
mode-guard( c, g)	guard( c != 0 && !(c < g))	mode-rule( a , c )	rule : b ← a / c

Note:  $a !=> b \wedge c$  denotes evaluate a if a is true then evaluate b otherwise evaluate c

Figure 6: Composing predicates from primitive predicates.

single predicate reference to a full predicate expression by specifying a boolean expression tree as described in Section 4.3. This gives greater expressive power to our formalism but in most cases a single predicate reference will suffice.

## 4.5 Algorithm for Auto-completion and Auto-update

The compiler uses composite predicates to flag the primitive predicates which are constituents of this composite predicate. These flagged predicates are ignored in the while-loop given below. The algorithm outlined below iterates over a list of predicates P as shown below:

- ```

AUTO-COMPLETE(P)
1. let repository ← COLLECT-KNOWN-VALUES
2. if VERIFY-CHECK-CODES(P) ≠ FALSE
3.   while CHANGE(repository) = TRUE do
4.     foreach  $p_i$  do
5.       if CHECK-MODE-CHECK-CODE( $p_i$ ) ≠ FALSE

```

```

6.         foreach  $rule_j$  of  $p_i$  do
7.             if CHECK-MODE-GUARD( $guard_j$ )  $\neq$  FALSE
8.                 if EVALUATE-GUARD( $guard_j$ )  $\neq$  FALSE
9.                     if CHECK-MODE-RULE( $rule_j$ )  $\neq$  FALSE
10.                        TRIGGER-RULE( $rule_j$ )
11.                    if VERIFY-CHECK-CODES(P)  $\neq$  FALSE
12.                        then UPDATE(repository) (with rule trigger field)
13.                            break(from loop in line 6)
14.                        else RETRACT(repository)
15.                return
16.        return
17.    end-while

```

The algorithm keeps track of known form fields by means of a *repository*. The repository is initialised (line 1) and all check-codes are verified before the start of the while-loop (line 2). The loop starts with a list of predicates P(line 4). For any predicate  $p_i$ , if any of the check-code mode fields are not known, then it becomes a candidate for rule triggers (line 5). All rules of that predicate are then checked for rule triggers(line 6). If the guard for a rule ( $rule_j$ ) is not empty<sup>2</sup>, then the guard mode is checked for the existence of field values that are required to evaluate the guard (line 7). If the mode is satisfied then the guard is evaluated (line 8). If a guard is satisfied (evaluates to true) then a check is made on the rule mode (line 9). If all the fields of a rule mode are known, a rule is triggered (line 10). All the check-codes are run to ensure that check-codes are satisfied (line 11). If the check-codes are not satisfied the value is retracted (line 14) otherwise it is added to the repository (line 12). The next predicate is then chosen for rule trigger (line 13). The while-loop (line 3-17) terminates if in the last iteration there was no rule trigger (line 3). If field formats are specified, then they are taken into account (line 2,10). In line 2, all field values are checked against their specified field formats whereas in line 10, the deduced field value is checked for its conformance to the specified field format.

The algorithm for auto-update is similar except for the fact that when a user updates an existing value all the deduced values are removed from the repository and new values are then computed by the algorithm given above. If metainfo is specified then it is taken into account at this stage.

## 4.6 Aggressive Auto-completion

As described above, the auto-update algorithm *per se* will be correct only when the updated field is not the deduced value. If a deduced value is updated, we require additional information about the fields as shown by the celsius fahrenheit example later in the Section 5. An aggressive auto-completion ignores a given field value based on whether a deduced value is updated or not. The effect is that the old value is *overwritten* by a new value. This involves a run-time manipulation of repository by the extra-information about field updates. The new values are still computed on the basis of auto-completion rules specified for a given relationship.

## 4.7 Semantics of the Underlying Mechanism

Given a collection of form fields  $F_1, F_2, \dots, F_n$  and associated relationships  $\mathbb{R}_1, \mathbb{R}_2, \dots, \mathbb{R}_m$  in the form of predicates. An *iteration* can be defined that does the following for each form field  $F_i$ :

- evaluate the current relationship based on all form field values; and
- update the field value based on new current relationship.

The update of the field varies; for an unknown field the update takes the form of filling the field with the deduced value whereas, for a known field the update takes the form of overwriting the current field value.

---

<sup>2</sup>An empty guard always return true

It is possible to give an upper bound to the number of iterations that are performed before a fixed-point is reached. For simple forms it is either 1 or 2 iterations.

## 5 Applications and Availability

We illustrate the entire mechanism by a number of examples.

### 5.1 Examples

---

#### Example 1 Purchase Order

---

We consider again the purchase order form.

```
<table BORDER COLS=5 WIDTH="80%" >
<tr>
  <td WIDTH="10%">Item No.</td>
  <td WIDTH="30%">Article Description</td>
  <td WIDTH="15%">Quantity</td>
  <td WIDTH="10%">Price per Item</td>
  <td WIDTH="15%">Cost</td>
</tr>

<tr>
  <td>1</td>
  <td>Microsoft Office Suite</td>
  <td><input type="text" name="Q1" value="" size="3"/></td>
  <td>6<input type="hidden" name="P1" value="6"/></td>
  <td><input type="hidden" name="C1" value="" size="5"/></td>
</tr>

<tr>
  <td>2</td>
  <td>Microsoft SQL Server</td>
  <td><input type="text" name="Q2" value="" size="3"/></td>
  <td>2<input type="hidden" name="P2" value="2"/></td>
  <td><input type="hidden" name="C2" value="" size="5"/></td>
</tr>

<tr>
  <td>3</td>
  <td>Corel Draw Version 6.0</td>
  <td><input type="text" name="Q3" value="" size="3"/></td>
  <td>7<input type="hidden" name="P3" value="7"/></td>
  <td><input type="hidden" name="C3" value="" size="5"/></td>
</tr>

<tr>
  <td></td>
  <td></td>
  <td>Total Quantity<input type="text" name="TQ" value=""
size="4"/></td>
  <td></td>
  <td>Total Price<input type="text" name="TC" value="" size="7"/></td>
</tr>
</table>
```

```
<input type="reset" name="resetbtn" value="Reset"/></form>
```

We add format specification to the fields namely, all quantities are positive integers and total cost can be positive real number. When the HTML page containing the form is loaded by the browser the form looks as shown in Figure 7.

## Purchase Order Form






Item No.	Article Description	Quantity	Price per Item	Cost
1	Microsoft Office Suite	2 	6	
2	Microsoft SQL Server	3 	2	
3	Corel Draw Version 6.0	2 	7.85	
		Total Quantity		Total Price
		7 		33.7 

Figure 7: Purchase order form with validation and auto-completion.

The auto-complete specification is shown below. The following code fragment shows how a template for a primitive predicate that models the relationship  $C = A + B$  is defined:

```
< predicate id="predicateplus">
< template>
  < argumentlist>
    < argument name ="A" />
    < argument name ="B" />
    < argument name ="C" />
  </ argumentlist>
  < checkcode>
    < operator name="plus"/>
    < result name="C"/>
    < argumentlist>
      < argument name="A"/>
      < argument name="B"/>
    </ argumentlist>
  </ checkcode>
  < autocompletionrules>
    < rule>
      < argumentlist>
        < argument name="A" />
        < argument name="B" />
      </ argumentlist>
      < guard>
        < empty/>
      </ guard>
      < result name="C" />
      < operator name="plus"/>
```

```

</ rule>
< rule>
  < argumentlist>
    < argument name="C" />
    < argument name="B" />
  </ argumentlist>
  < guard>
    < empty/>
  </ guard>
  < result name="A"/>
  < operator name="minus"/>
</ rule>
< rule>
  < argumentlist>
    < argument name="C" />
    < argument name="A" />
  </ argumentlist>
  < guard>
    < empty/>
  </ guard>
  < result name="B"/>
  < operator name="minus"/>
</ rule>
</ autocomplete>
</ template>
</ predicate>

```

The "plus" and "minus" are calls to predefined JavaScript functions. We defer the details about how these functions are called till Section 7. A template is instantiated by binding concrete form fields to the arguments of the argument-list as shown below:

```

< predicate id="predicate5">
< bind idref="predicateplus">
  < to>
    < argument name="A"/>
    < field name="Q1"/>
  </ to>
  < to>
    < argument name ="B"/>
    < field name ="Q2"/>
  </ to>
  < to>
    < argument name ="C"/>
    < interface id="predicate6" name="A"/>
  </ to>
</ bind>
</ predicate>

< predicate id="predicate6">
< bind idref="predicateplus">
  < to>
    < argument name="A"/>
    < interface id="predicate5" name="C"/>
  </ to>
  < to>
    < argument name="B"/>
    < field name ="Q3"/>
  </ to>
</ bind>
</ predicate>

```

```

</ to>
< to>
  < argument name ="C"/>
  < field name = "TQ"/>
</ to>
</ bind>
</ predicate>

```

The code above models a relationship of the form  $Q1 + Q2 + Q3 = TQ$  where  $Q1$ ,  $Q2$ ,  $Q3$  are the three quantities above and  $TQ$  denote the total quantity as shown in Figure 7. Note the use of interfaces to capture intermediate values. A form designer can also make certain fields as read-only; specifically, fields which derive their values from other field values. In this example the total quantity and the total cost are deduced quantities and these fields are specified as read-only as shown below:

```

< autoupdate name="TC" status="forbidden"/>
< autoupdate name="TQ" status="forbidden"/>

```

The field formats for field  $Q1$  can be specified as:

```

< regexp id="digit">
  < charrange low ="0" high="9"/>
</ regexp>

< regexp id ="intnumber">
  < plus> < regexp idref="digit"/></ plus>
</ regexp>

< constraint field="Q1">
  < regexp idref="intnumber"/>
</ constraint>

```

This example also highlights the seamless integration of field formats and auto-completion in such a manner that an auto-completed value is also a valid field value.

---

### Example 2 Decision Tree

---

We take another example to show how primitive predicates can be used to compose higher-order predicates. Consider a HTML form where a user can enter three numbers and the smallest number is automatically deduced from the given information and assigned to a fourth field. A code fragment of the HTML form is shown below:

```

<form>Decision Tree
<table BORDER COLS=3 WIDTH="50%" >
<tr>
  <td>Number 1</td>
  <td>Number 2</td>
  <td>Number 3</td>
</tr>
<tr>
  <td><input type="text" name="N1" size="5"/></td>
  <td><input type="text" name="N2" size="5"/></td>
  <td><input type="text" name="N3" size="5"/></td>
</tr>

```

<i>PredicateId</i>	<i>Relationships</i>
Predicate0	N1 > N2
Predicate1	N2 > N3
Predicate2	N3 > N1
Predicate3	N1 = N4
Predicate4	N2 = N4
Predicate5	N3 = N4

Table 1: Predicates Ids and Corresponding Relationships

```

<tr>
<td></td>
<td>The smallest number</td>
<td><input type="text" name="N4" size="5"/></td>
</tr>
</table>
</form>

```

This problem can be formulated with a number of primitive and composed predicates as shown below:

```

1.< predicate id="predicate6">
2. < if>
3.   < predicate idref="predicate1"/>
4.     < then>< predicate idref="predicate5"/></ then>
5.     < else>< predicate idref="predicate4"/></ else>
6. </ if>
7.</ predicate>
8.< predicate id="predicate7">
9. < if>
10.  < predicate idref="predicate2"/>
11.    < then>< predicate idref="predicate3"/></ then>
12.    < else>< predicate idref="predicate5"/></ else>
13.  </ if>
14.</ predicate>
15.< predicate id="predicate8">
16. < if>
17.   < predicate idref="predicate0"/>
18.     < then>< predicate idref="predicate6"/></ then>
19.     < else>< predicate idref="predicate7"/></ else>
20.  </ if>
21.</ predicate>

```

The compiler flags all the predicates that are used to built composite predicates. A transitive closure of dependency of composite predicates yields that last predicate is an un-flagged predicate. Hence only the last predicate (predicate8) will have rule triggers. All the predicates and the corresponding relationships they model are shown in Table 1. The resultant predicate has 8 auto-completion rules of which 4 are never triggered because the guard-mode contains a field which is the result of rule trigger. Figure 8 shows how the smallest number field is auto-completed when the three numbers are known. Note the value deduced conforms to the field format as shown by the green light. Figure 9 shows how the smallest number field changes when the second number is updated.

---

**Example 3** 2002 FIFA World Cup

---

Decision Tree

Number 1	Number 2	Number 3
34	43	344
	The smallest number	34

Figure 8: Finding smallest of 3 numbers: Auto-completion mode.

Decision Tree

Number 1	Number 2	Number 3
34	4	344
	The smallest number	4

Figure 9: Finding smallest of 3 numbers: Auto-update mode.

To show that our approach is correct for other named form fields we have tried to model 2002 FIFA World Cup [16] where the entire process of choosing a winner from 32 teams that qualify for the final stage of the tournament is modelled as a single HTML form. This example is demonstrative of what can be done when relationships are very domain-specific.

The final stage of the tournament is played in two rounds: First round and Second round. In the first round, the teams are divided into 8 groups of 4 teams each. The teams play against each other in a round-robin fashion i.e. a team plays with 3 other teams in the same group. Any particular group has six matches in all. The winner and the runners-up from each group qualify for round two. A section of the form that models first round is shown in Figure 10. A user can specify the scores of different matches to see who wins the match. But, since there are 32 teams, it becomes very tedious to specify scores for each and every match. To this end, we have given every match a default score which also reflects the actual match results and can be verified at FIFA World Cup 2002 web site.

Group A

	France FRA Team1	Senegal SEN Team2	Uruguay URU Team3	Denmark DEN Team4
Team1 France				
Team2 Senegal	1 0			
Team3 Uruguay	0 0	3 3		
Team4 Denmark	2 1	1 1	2 1	

Figure 10: The preliminary round where teams compete in a round-robin fashion.

The second round consists of a number of phases: round of 16, quarter finals, semi finals and finals in that order. In this round, teams play in a knock-out fashion; only the winner qualifies for the next phase. A section of the HTML form that models the Quarter finals and Semifinal phase is shown in Figure 11 The

Team	Team	Scores	Winner Designated By
Team 1 England	Team2 Brazil	1   2	QFWinner1
Team3 Germany	Team4 USA	1   0	QFWinner2
Team5 Spain	Team6 Korea Republic	3   5	QFWinner3
Team7 Senegal	Team8 Turkey	0   1	QFWinner4

## Semi Finals model

1. QFWinner 1 vs. QFWinner 4
2. QFWinner 2 vs. QFWinner 3

Team	Team	Match Score	Winner Designated By
Team1 Brazil	Team2 Turkey	1   0	SFWinner1
Team3 Germany	Team4 Korea Republic	1   0	SFWinner2

Figure 11: The round two where teams compete in a knock-out fashion.

appendix A.3 describes the the criteria for winners from different rounds. After each round half of the teams are eliminated. So, at the start there are 32 teams, after this round there are 16 teams and so forth till a winner emerges. A user can enter the scores of the individual matches but he cannot choose the teams that qualify for the next stage of the tournament even though an individual team is shown by a drop-down box. as shown in figure 10.

When the form is first loaded and since the match score have been supplied as default values the winner is calculated from the match scores. It is important to note that any team could be a winner out of the 32 teams which means that winners in each stage of the tournament could be any of the 32 teams as shown in Figure 11.

A programmer can specify these fields as *read-only* as they are deduced from the score entered by the user. Due to the deficiency in the design of the singular `select` the compiler inserts an *err option* if none of the options is selected. A singular `select` by default chooses the first option as selected if none is specified. With this annotation, we can now differentiate whether the first option is selected by default or not.

This example also highlights the limitations of PowerForms existing syntax of modelling field dependencies. We wrote almost 4000 lines of code in the existing PowerForms syntax for modelling field interdependencies only to realise that existing formulation is not expressive enough to model such complex relationships. The **include-file** element described in our grammar takes care of such scenarios where the existing library is unable to support these very domain specific functions. The correctness of our approach then assumes that a programmer will code these functions in the common subset of JavaScript that we earlier mentioned.

Here again, when the form is first loaded the mechanism is run in auto-completion mode and later when fields are updated it is run in auto-update mode. In each case, a fixed-point is demonstrated by a unique winner that emerges at the end of the computation. For the default case, when no field value is entered by the user, the winner is shown in Figure 12.

---

## Finals Model

---

SFWinner1 vs SFWinner2

Team	Team	Match Score	Winner
Team1 Brazil	Team2 Germany	2 0	
Final winner Brazil			

Figure 12: A section of the form showing the final winner.

---

### Example 4 Temperature Conversion

---

A final example deals with the idea of aggressive auto-completion. Here, we model a celsius-fahrenheit conversion. Consider a form that models this conversion as shown below:

```
<form >
Celsius Fahrenheit Conversion
<table BORDER COLS=3 WIDTH="50%" />
<tr>
<td>celsius</td>
<td>fahrenheit</td>
</tr>
<tr>
<td><input type="text" name="celsius" /></td>
<td><input type="text" name="fahrenheit" /></td>
</tr>
```

```
</table>
</form>
```

This conversion can be modelled by a number of predicates which model addition and multiplication. As an aid to form designer, our tool has a mechanism that allows a form designer to store and later retrieve the stored templates by specifying the URL as shown in the following specification:

```
<predicate id="predicate0" >
  <bind
url="http://www.brics.dk/~sunil/worldcupcon/predicateplus">
  <to>
    <argument name = "C"/>
    <field name = "fahrenheit"/>
  </to>
  <to>
    <argument name = "B"/>
    <const value = "32"/>
  </to>
  <to>
    <argument name ="A"/>
    <interface id ="predicate1" name ="A"/>
  </to>
</bind>
</predicate>

<predicate id="predicate1">
  <bind url
="http://www.brics.dk/~sunil/worldcupcon/predicatemultiply">
  <to>
    <argument name ="C"/>
    <interface id = "predicate2" name="C"/>
  </to>
  <to>
    <argument name ="B"/>
    <const value ="5"/>
  </to>
  <to>
    <argument name ="A"/>
    <interface id ="predicate0" name="A"/>
  </to>
</bind>
</predicate>

<predicate id="predicate2">
  <bind
url="http://www.brics.dk/~sunil/worldcupcon/predicatemultiply">
  <to>
    <argument name ="C"/>
    <interface id ="predicate1" name="C"/>
  </to>
  <to>
```

```

    i argument name ="B"/i
    i const value ="9"/i
i/ to i
i to i
    i argument name ="A"/i
    i field name ="celsius"/i
i/ to i
i/ bind i
i/ predicate i

```

When the browser loads the page, both the fields are empty. A user then inputs 60 as the celsius field value. The fahrenheit is computed and gets the value 140 as shown in Figure 13. However, when a user modifies fahrenheit field we want that the celsius field should reflect the corresponding value. Clearly, our

celsius	fahrenheit
60	140

Figure 13: Auto-completion conversion when both fields are empty.

specification above and the algorithm for auto-completion will not produce the desired effect since the celsius field is already *known*. We require additional information as given below:

```

< metainfo update="celsius" ignore="fahrenheit"/>
< metainfo update="fahrenheit" ignore="celsius"/>

```

If a user then modifies the deduced fahrenheit field value to 14, a corresponding value now appears in celsius field as shown in Figure 14. This is an instance of aggressive auto-completion wherein, overwriting of values is allowed. When a field update is detected, the algorithm uses this additional information to decide which field values to ignore in the repository.

celsius	fahrenheit
-10	14

Figure 14: An instance of aggressive auto-completion.

## 5.2 Availability

All the examples described in this report and several others are available on the WWW at the following URL <http://www.brics.dk/~sunil/worldcupcon/ShowCase.html>.

## 6 Properties

We identify several properties that a auto-completion mechanism must have to behave in a consistent manner. A mechanism is *well-behaved* if it satisfies the following properties : monotonicity, order independence and locality.

### 6.1 Monotonicity

The underlying semantics of the auto-completion mechanism dictate that every iteration defined in Section 4.5 on a lattice  $L$  of relationship descriptions is *monotone*. Formally, a function  $f: L \rightarrow L$  is monotonic iff  $\forall x,y \in S, x \leq y \implies f(x) \leq f(y)$ . where  $S$  denotes a set of all possible subsets of relationships and  $L=(S,\subseteq)$ . The monotonicity property guarantees an end to the fixed-point iterations.

### 6.2 Order Independence

The fixed-point reached in the computation is independent on the order in which rules are triggered.

### 6.3 Locality

Locality refers to the fact that the representation and behaviour relationships is independent of one another. Rules are triggered and values are deduced on the basis of a given relationship only.

## 7 Implementation Details

The compiler takes the optional field format specifications, auto-completion specifications and a raw HTML document and parses the document for HTML form fields and interweaves equivalent JavaScript code for the above specifications. Even though the present compiler is built on existing PowerForms compiler, yet it can be used in a stand-alone fashion (when no field formats are specified).

The whole mechanism is run by the underlying JavaScript code. The compiler converts all predicate interfaces, constants to hidden fields. These hidden fields are different from those that are part of the original HTML form design (price value in a Purchase Order form). This gives a clean and uniform way of handling various interfaces and constants described by a form designer.

All named form fields, except fields that are read-only, that form a part of the relationship are given an **onchange** event handler. The **onchange** event triggers the algorithm to compute a new fixed-point every time a field is modified. Fields that have been specified as *read-only* are given handlers as follows:

- A text field is given an **onfocus** handler.
- A select field of type is given an **onchange** handler that stores the current state of the field. A select field is made read-only by undoing any action of the user.

Also, note that these handlers have been provided in all IE3+, NN3+ versions and most other browsers and thus ensure consistent behaviour across the entire range of platforms and browsers.

If field validation formats are specified, then relevant fields are given a **onkeyup** handler and the textual fields are annotated with traffic lights to reflect the validity of the value entered so far.

A call to JavaScript function name `foo` with a list of arguments specified in the specifications is implemented as follows:

```
<rule>
...
<result name="t"/>
<operator name="foo"/>
```

```

    <argumentlist>
      <const value = "3"/>
      <const value = "2"/>
    </argumentlist>
    . . . . .
  </rule>

```

JavaScript function `eval` is called on `foo` and then arguments are passed to the evaluated function as an array. The following fragment of JavaScript code illustrates the concept further:

```

var t;
var func="foo";
var argArray = new Array(2,3);
t = eval(func)(argArray);

```

It is assumed that the function `foo` is a predefined function in the library that is included at run-time. In our case, `foo` represents addition of two numbers as shown below:

```

function foo()
{
  var arg1,arg2;
  arg1 = foo.arguments[0][0];
  arg2 = foo.arguments[0][1];
  . . . . .
  return (arg1 + arg2);
}

```

The underlying engine relies heavily on a core library of basic algebraic, array manipulation, DOM manipulation and boolean expression tree traversal<sup>3</sup> routines and consists of over 3000 lines of JavaScript code. We have provided a set of predefined functions which occur routinely in the relationship descriptions and there is a provision to support user-defined libraries.

The mechanism has two modes: *auto-completion* and *auto-update* mode. The mechanism works in auto-completion mode if the user assigns values to empty fields and does not update any of the non-empty fields whereas the mechanism works in auto-update mode when a user changes the value of a non-empty field. This causes deduced field values to be retracted and new field values are then computed on the basis of the updated field value.

We have tested our implementation on Explorer on Unix and Windows, Netscape on Unix and Windows and IE on Macintosh.

## 8 Future Work

This section focuses on ideas for the remaining period of my PhD studies.

### 8.1 Possible future extensions to auto-completion mechanism

We have found that when we compose<sup>4</sup> predicates from primitive predicates it is possible, at compile-time, to perform an analysis of the rules which will never be triggered. These rules can then be flagged and simply ignored in further compositions involving this predicate. Such an optimisation will remove a lot of redundant

<sup>3</sup>Guards are represented as expression trees with non-terminal nodes as boolean operators

<sup>4</sup>by means of **if-then-else** construct

code and make the mechanism more efficient. For example in the decision tree example in Section 5, 50% of the auto-completion rules of the un-flagged predicate are never triggered.

We hope to improve upon the algorithm mentioned in Section 4.5 by doing a dependency analysis at compile-time and use that information while iterating over the list of predicates.

There are many instances of HTML forms where a set of values can be predicted for a field. This set can be used to predict other field values (possibly, a set of values). Clearly, this requires a more complex mechanism (than one suggested in this report) and might require additional specifications but certainly it will involve extending the current work. We hope to investigate that in the future.

## 9 Conclusion

In this report, we have described and successfully implemented a declarative, high-level approach towards achieving auto-completion and auto-update of form fields in HTML forms. Such an approach has the advantage that server-side computations are now handled at the client-side in a consistent and safe manner.

## References

- [1] World Wide Web Consortium:  
URL :<http://www.w3c.org>
- [2] Core JavaScript Reference 1.5:  
URL :<http://devedge.netscape.com/library/manuals/2000/javascript/1.5/reference/>
- [3] Micah Dubinko, Dave Raggett, Sebastian Schnitzenbaumer, Malte Wedel. XForms Requirements: W3C Working Draft.  
URL:<http://www.w3.org/TR/xhtml-forms-req>.
- [4] Micah Dubinko, Leigh L. Klotz, Roland Merrick and T. V. Raman. XForms1.0. W3C Candidate Recommendation, 12 November 2002.  
URL:<http://www.w3.org/TR/2002/CR-xforms-20021112/>.
- [5] XML Path Language (XPath) Version 1.0. W3C Recommendation, 16 November 1999.  
URL:<http://www.w3.org/TR/xpath>.
- [6] XForms - The Next Generation of Web Forms, W3C.  
URL:<http://www.w3c.org/MarkUp/Forms/>.
- [7] Claus Brabrand, Anders Moller, Mikkel Ricky and Michael I. Schwartzbach. PowerForms: Declarative Client-side Form Field Validation, In *World Wide Web Journal*, vol. 3, no. 4.
- [8] Wood L. et al. *Document Object Model (DOM) Level 1 Specification* (Second Edition).W3C, 2000.  
URL:<http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/>.
- [9] Wm Leler : *Constraint Programming Languages, Their Specification and Generation*, Addison-Wesley 1988.ISBN 0-201-06243-7.
- [10] Mikkel Ricky : Automatisk validering af webbaserede formularer, Speciale DAIMI, Aarhus Universitet, Denmark, May 2002.
- [11] Guy Steele : The Definition and Implementation of a Computer Programming Language Based on Constraints, AI Technical Report No. AI-TR-595, Department of Artificial Intelligence, MIT, USA.

- [12] John Boyer and Mikko Honkala: The XForms Computation Engine : Rationale, Theory and Implementation Experience. In *Proc. of the 6th IASTED International Conference, Internet and Multimedia Systems, and Applications (IMSA 2002)*, August 12-14, 2002, Kauai, Hawaii, USA.
- [13] Scalable Vector Graphics (SVG) 1.1/1.2/2.0 Requirements, W3C Working Draft, 22 April 2002. URL:<http://www.w3.org/TR/SVG2Reqs/>.
- [14] Synchronized Multimedia Integration Language(SMIL). URL:<http://www.w3c.org/AudioVideo/>.
- [15] Jennifer Niederst. *Web Design in a NutShell, A Desktop Quick Reference*, O'Reilly, 2nd Edition, 2001.
- [16] 2002 FIFA World Cup HomePage. URL:<http://fifaworldcup.yahoo.com/>.
- [17] Claus Brabrand, Michael I. Schwartzbach and Mads Vanggaard. The METAFRONT System: Extensible Parsing Transformation, LDTA 2003, Warsaw, Poland.
- [18] Jay Earley. An Efficient Context-Free Parsing Algorithm, pg 94-102, *Communications of the ACM*, Volume 13 Number 2, 1970.
- [19] Aske Simon Christensen, Anders Moller and Michael I. Schwartzbach: Extending Java for High-Level Web Service Construction [to appear] in *ACM Transactions on Programming Languages and Systems*.
- [20] Claus Brabrand, Anders Moller and Michael I. Schwartzbach: Static Validation of Dynamically Generated HTML, PASTE, Snowbird, Utah, 2001.
- [21] Abdoulraouf A. Elbibasand and M. J. Ridley: Generation and Validation of HTML forms using Metadata, PGNet2003.
- [22] Frank van Harmelen, Jos van der Meer. WebMaster: Knowledge-based Verification of Web-Pages, *Practical Application of Knowledge Management*, PAKeM'99.
- [23] Arvind K. Joshi, Leon S. Levy and Kang Yueh: Local constraints in the syntax and semantics of programming languages, *Fifth Annual ACM Symposium on Principles of Programming Languages*.
- [24] XML Schema. W3C URL:<http://www.w3c.org/XML/Schema>.
- [25] Claus Brabrand. *Synthesizing Safety Controllers for Interactive Web Services*, Master's Thesis, URL:<http://compose.labri.u-bordeaux.fr/people/brabrand/>.
- [26] Claus Brabrand. Domain Specific Languages for Interactive Web Services, PhD Dissertation, November 2002, University of Aarhus, Denmark.
- [27] Netscape Corp. Javascript form validation sample code. URL:<http://developer.netscape.com/docs/examples/javascript/formval/overview.html>.
- [28] M.C. Rousset. Verifying the World Wide Web: a position statement. In F. van Harmelen and J. van Thienen, editors, *Proceedings of the Fourth European Symposium on the Validation and Verification of Knowledge Based Systems*(EUROVAV'97).
- [29] XFDL - *Extensible Form Description Language* available at URL : <http://www.pureedge.com/xfdl/>.

## A Appendix

### A.1 JavaScript Compatibility

Table 2 shows the JavaScript compatibility of different browsers on several commonly used platforms.

<i>Platform</i>	<i>Browser</i>	<i>JavaScript version</i>
Windows	MS IE 5.5	1.5 (ECMA 3)
Windows	MS IE 5.0	1.3 (ECMA 2)
Windows	MS IE 4.0	1.2 (ECMA 1)
Windows	MS IE 3.0	1.0
Windows	MS IE 2.0	Not supported
Windows	NN 6	1.5 (ECMA 3)
Windows	NN 4.7 / 4.5	1.3 (ECMA 2)
Windows	NN 4.0	1.2
Windows	NN 3.0	1.1
Windows	NN 2.0	1.0
Mac	MS IE 5.0	1.3 (ECMA 2)
Mac	MS IE 4.0	1.2 (ECMA 1)
Mac	MS IE 3.0	1.0
Mac	NN 6	(ECMA 3)
Mac	NN 4.7 / 4.5	(ECMA 2)
Mac	NN 4.0	1.2
Mac	NN 3.0	1.1
Mac	NN 2.0	1.0
Unix	NN 6	1.5 (ECMA 3)
Unix	NN 4.7 / 4.5	1.3 (ECMA 2)
Unix	NN 4.0	1.2
Unix	NN 3.0	1.1
Unix	NN 2.0	1.0

Table 2: JavaScript compatibility for different browsers on different platforms

In the table 2 following abbreviations have been used:

- MS IE - Microsoft Internet Explorer
- NN - Netscape Navigator
- ECMA - Prior 1994, known as European Computer Manufacturers Association. Now known as ECMA International - European Association for Standardizing Information and Communication Systems.

## A.2 BNF Specification of the full Grammar

<i>powerforms</i>	→	<b>&lt; powerforms &gt;</b> ( <i>predicate-def</i>   <i>constraint</i>   <i>autoupdate-def</i>   <i>metainfo</i>   <i>include-file</i> ) <sup>*</sup> <b>&lt;/powerforms &gt;</b>
<i>predicate-def</i>	→	<b>jpredicate id=stringconst</b> <i>i</i> <i>predicate-body</i>
<i>predicate-body</i>	→	<b>jbody</b> <i>i</i> <i>argumentlist</i> <i>checkcode</i> <i>autocompletionrules</i> <b>i/body</b> <i>i</i>   <b>jtemplate</b> <i>i</i> <i>argumentlist</i> <i>checkcode</i> <i>autocompletionrules</i> <b>i/template</b> <i>i</i>   <b>jbind [idref=stringconst] [url=stringconst]</b> <i>i</i> <i>binding-list</i> <sup>*</sup> <b>i/bind</b> <i>i</i>   <b>jif</b> <i>i</i> <i>predicate-expression</i> <b>i then</b> <i>i</i> <i>predicate-ref</i> <b>/then</b> <i>i</i> <b>i else</b> <i>i</i> <i>predicate-ref</i> <b>/else</b> <i>i</i> <b>i/if</b> <i>i</i>
<i>predicate-expression</i>	→	<i>predicate-ref</i>   <b>jand</b> <i>i</i> <i>predicate-expression</i> <i>predicate-expression</i> <b>i/and</b> <i>i</i>   <b>jor</b> <i>i</i> <i>predicate-expression</i> <i>predicate-expression</i> <b>i/or</b> <i>i</i>   <b>jnot</b> <i>i</i> <i>predicate-expression</i> <b>i/not</b> <i>i</i>
<i>predicate-ref</i>	→	<b>jpredicate idref=stringconst</b> / <i>i</i>
<i>binding-list</i>	→	<b>jto</b> <i>i</i> <i>argument</i> <i>assignment</i> <b>i/to</b> <i>i</i>
<i>assignment</i>	→	<b>jfield name =stringconst</b> / <i>i</i>   <b>jconst value= intconst</b> / <i>i</i>   <b>jinterface id=stringconst name=stringconst</b> / <i>i</i>
<i>checkcode</i>	→	<b>jcheckcode</b> <i>i</i> <i>operator</i> <i>result</i> <i>argumentlist</i> <b>i/checkcode</b> <i>i</i>
<i>operator</i>	→	<b>joperator name = stringconst</b> / <i>i</i>
<i>result</i>	→	<b>jresult name = stringconst</b> / <i>i</i>
<i>argumentlist</i>	→	<b>jargumentlist</b> <i>i</i> <i>argument</i> <sup>*</sup> <b>i/argumentlist</b> <i>i</i>
<i>argument</i>	→	<b>jargument name = stringconst</b> / <i>i</i>

<i>autocompletionrules</i>	→	<b> autocompletionrules</b> <sub>i</sub> <i>autocompletionrules-body</i> <b> /autocompletionrules</b> <sub>i</sub>
<i>autocompletionrules-body</i>	→	<b> empty</b> / <sub>i</sub> <i>rule</i> <sup>+</sup>
<i>rule</i>	→	<b> rule</b> <sub>i</sub> <i>argumentlist</i> <i>guard</i> <i>result</i> <i>operator</i> <b> /rule</b> <sub>i</sub>
<i>guard</i>	→	<b> guard</b> <sub>i</sub> <i>guard-body</i> <b> /guard</b> <sub>i</sub>
<i>guard-body</i>	→	<b> /empty</b> <sub>i</sub> <i>argumentlist</i> <i>operator</i>
<i>autoupdate-def</i>	→	<b> autoupdate name=stringconst status=stringconst</b> / <sub>i</sub>
<i>include-file</i>	→	<b> include name =stringconst</b> / <sub>i</sub>
<i>metainfo</i>	→	<b> metainfo update=stringconst ignore=stringconst</b> / <sub>i</sub>
<i>constraint</i>	→	<b> constraint [form=stringconst] [field=stringconst] [id=stringconst]</b> <sub>i</sub> <i>regexp-def</i> <b> /constraint</b> <sub>i</sub>
<i>regexp-def</i>	→	<b> regexp id=stringconst</b> <sub>i</sub> <i>regexp</i> <b> /regexp</b> <sub>i</sub>
<i>regexp</i>	→	<b> empty</b> / <sub>i</sub> <b> anychar</b> / <sub>i</sub> <b> anything</b> / <sub>i</sub> <b> const value=stringconst</b> / <sub>i</sub> <b> charset value=stringconst</b> / <sub>i</sub> <b> charrange low=charconst high=charconst</b> / <sub>i</sub> <b> interval low=intconst high=intconst [width=intconst] [radix=intconst]</b> / <sub>i</sub> <b> repeat count=intconst</b> <sub>i</sub> <i>regexp</i> <b> /repeat</b> <sub>i</sub> <b> repeat [min=intconst] [max=intconst]</b> <sub>i</sub> <i>regexp</i> <b> /repeat</b> <sub>i</sub> <b> complement</b> <sub>i</sub> <i>regexp</i> <b> /complement</b> <sub>i</sub> <b> optional</b> <sub>i</sub> <i>regexp</i> <b> /optional</b> <sub>i</sub> <b> plus</b> <sub>i</sub> <i>regexp</i> <b> /plus</b> <sub>i</sub> <b> intersection</b> <sub>i</sub> <i>regexp</i> <sup>+</sup> <b> /intersection</b> <sub>i</sub> <b> union</b> <sub>i</sub> <i>regexp</i> <sup>+</sup> <b> /union</b> <sub>i</sub> <b> concat</b> <sub>i</sub> <i>regexp</i> <sup>+</sup> <b> /concat</b> <sub>i</sub> <b> regexp pattern=stringconst</b> / <sub>i</sub> <b> regexp url=stringconst</b> / <sub>i</sub> <b> regexp idref=stringconst</b> / <sub>i</sub>

### A.3 2002 FIFA World Cup

For the round of 32 , Rankings in each group is determined as follows:

1. greater number of points obtained in all the group matches;
2. goal difference in all the group matches;
3. greater number of goals scored in all the group matches;  
If two or more teams are equal on the basis of the above three criteria, their place shall be determined as follows:
  4. greater number of points obtained in the group matches between the teams concerned;
  5. goal difference resulting from the group matches between the teams concerned;
  6. greater number of goals scored in all the group matches between the teams concerned;
  7. drawing lots by the Organising Committee for the FIFA World Cup

For other rounds, namely the round of sixteen, quarterfinals, semifinals, finals teams will play in a knock-out fashion. For the round of sixteen, the teams play in the following order with the winners and runners-up from the Round 1 : Winner E vs Runner-up B = 1

Winner B vs Runner-up E = 2

Winner G vs Runner-up D = 3

Winner D vs Runner-up G = 4

Winner A vs Runner-up F = 5

Winner F vs Runner-up A = 6

Winner C vs Runner-up H = 7

Winner H vs Runner-up C = 8

For quarter finals,the above 8 teams qualify and they play against each other in a knock-out fashion as described below:

Winner 1 vs Winner 3 = A

Winner 2 vs Winner 4 = B

Winner 5 vs Winner 7 = C

Winner 6 vs Winner 8 = D

The winners from the above round then play against each other in the semifinals as follows:

Winner A vs Winner B = 1

Winner C vs Winner D = 2

The finals are then played between 1 and 2 shown above.