xAcme: CMU Acme Extensions to xArch.

Bradley Schmerl.

23 March, 2001.

Introduction

XML is emerging as an industry-standard for the representation of data, a standard that is much more flexible than HTML. XML allows one to define a document's structure using user-defined tags. This allows tags to be defined that capture structure and meaning for specific domains, allowing for exchange of structured data between participants in that domain. One specific domain to which XML is being applied is that of software architectures. A standard for architectural structure, called *xArch*, has been developed by the ABLE group at CMU and the ISR group at UCI. Since xArch merely represents the structure of an architecture, extensions are required to include higher-level concepts such as types, styles, constraints or variation in an architecture.

This document describes the CMU extensions to xArch that allow all of Acme to be described. It is assumed that the reader is familiar with the Acme ADL, as described in http://www.cs.cmu.edu/~able/publications/acme-fcbs/. The xAcme extensions add facilities for representing properties, types and families in XML.

A brief introduction to XML Schema

In order to produce a standard set of tags that can be used in XML documents to represent some data, the tags and the rules for composing documents need to be defined. Traditionally, this has been accomplished using Document Type Definition (DTD) files, that specify the grammar for an XML document. XML documents may be validated against this definition, which can tell whether an XML document uses tags in a way consistent with the standard expressed by the DTD.

Recently, the W3C has developed a more flexible mechanism for specifying the grammar of an XML document. This mechanism is called XML Schema, and has the following benefits over DTDs:

1. Stronger data typing. The structure of the document can be expressed in both DTDs and XML Schema, but in XML Schema there are more constraints that can be specified to restrict the types that can included in "nodes" in the structure.
2. Inheritance. XML Schema allow element definitions to extend or restrict elements, which can then be used in XML documents.

Extensions to xArch rely heavily on the use of inheritance to extend the basic notions in xArch. The Acme extensions use inheritance to provide five additional schema to include the architectural notions of properties, constraints, types, and families. Before discussing these extensions, a brief example of inheritance and its use in XML documents is provided.

```
1 <schema …>
2   <element name=”purchaseOrder” type=”PO”/>

3   <complexType name=”PO”>
4     <sequence>
5       <element name=”orderDate” type=”date”/>
6       <element name=”shippingDate” type=”date”/>
7       <element name=”billingAddress” type=”Address”/>
8       <element name=”shippingAddress” type=”Address”/>
9     </sequence>
10  </complexType>

11  <complexType name=”Address”>
12    <sequence>
13      <element name=”name” type=”string”/>
14      <element name=”street” type=”string”
15               minOccurs=”1” maxOccurs=”2”/>
16      <element name=”city” type=”string”/>
17    </sequence>
18  </complexType>
  …
```

Figure 1. Schema Definition of an invoice.

Figure 1 provides a schema definition for purchase orders. The schema defines the tags that can be used in an XML document and the order and contents of those tags. In the schema defined in Figure 1, the first tag that can be used is `<purchaseOrder>`, which is the top element in the schema (defined on Line 2). The structure of this element is defined by the schema type `PO`, which is defined in lines 3-10. The type `PO` is in turn defined as a sequence of elements whose tag names and types are defined in lines 5-8. This definition specifies that a `purchaseOrder` must contain (in sequence) one each of `orderDate`, `shippingDate`, `billingAddress`, and `shippingAddress`.

Lines 11-18 of Figure 1 define what an address looks like. Note that in this address, there is no state or zip defined. The reason for this is that we wish to define `purchaseOrders` that can be shipped or billed to international addresses, and the format of these is differs from country to country. We can consider `Address` to be the base type of all addresses in our `purchaseOrders`, and for this we assume that all addresses have in common a `name`, one or two `street` elements and a `city`.

Figure 2 illustrates the extension of the base `Address` type defined in Figure 1 to two international addresses: United States addresses and Australian addresses. The `USAddress` type extends `Address` by adding two elements at the end of the sequence – a `state` element and a `zip` element. The `AusAddress` type extends `Address` by adding two different elements, a `postcode` and a `state`.[1]

These extensions can be used in place of an `Address`, by indicating their type (the indication of their type could be omitted if there was no need to validate an XML

[1] The state elements of `USAddress` and `AusAddress` are different types, but are not shown in the example. These types could be enumerations containing the abbreviations of all 50 US states for `USState`, and all 8 Australian states and territories for `AusState`.

document). Figure 3 gives an example of an XML document that uses the schema defined in Figure 1 and Figure 2. Line 4 of Figure 3 states that the `billingAddress` tag must satisfy the type definition for `USAddress`; Line 11 states that the `shippingAddress` is an `AusAddress`.

```
1   <complexType name="USAddress">
2     <complexContent>
3       <extension base="Address">
4         <sequence>
5           <element name="state" type="USState"/>
6           <element name="zip" type="positiveInteger"/>
7         </sequence>
8       </extension>
9     </complexContent>
10  </complexType>

11  <complexType name="AusAddress">
12    <complexContent>
13      <extension base="Address">
14        <sequence>
15          <element name="postcode" type="string"/>
16          <element name="state" type="AusState"/>
17        </sequence>
18      </extension>
19    </complexContent>
20  <complexType>
```

Figure 2. Extensions for US and Australian addresses.

```
1  <purchaseOrder>
2    <orderDate>2001-02-27</orderDate>
3    <shipDate>2001-03-01</shipDate>
4    <billingAddress xsi:type="USAddress">
5      <name>Bradley Schmerl</name>
6      <street>5000 Forbes Ave</street>
7      <city>Pittsburgh</city>
8      <state>PA</state>
9      <zip>15213</zip>
10   </billingAddress>
11   <shippingAddress xsi:type="AusAddress">
12     <name>Bradley Schmerl</name>
13     <street>48 Main North Road</street>
14     <city>Willaston</city>
15     <postcode>5118</postcode>
16     <state>SA</state>
17   </shippingAddress>
```

Figure 3. An example of inheritance in an XML document.

As stated previously, extension to xArch rely on the inheritance described in this section to add structure to the basic elements defined in the xArch schema. More information about XML Schema can be found at http://www.w3.org. A note of caution: many resources on the web refer to the XML Schema candidate recommendation of April 2000. However, the current XML Schema, and the one followed by xArch and its extensions, is the candidate recommendation of October 2000.

A brief introduction to xArch

The xArch schema was defined jointly by CMU and UCI to provide a common definition for the structural elements of an architectural instance. An architectural instance is an instance of a structure which may occur at design time or runtime.

The types defined in xArch are:

> **ComponentInstance**: an instance of a component. A component contains a description and a set of interfaces, with a possible subArchitecture to represent hierarchy.
>
> **ConnectorInstance**: an instance of a connector, containing a description and a set of interfaces. A connector may also contain a subArchitecture to represent hierarchical decomposition.
>
> **LinkInstance**: links define attachments between interfaces of components and interfaces of connectors.
>
> **GroupInstance**: a group is a collection of references to components and connectors that form some logical grouping. For example, the group may represent all components running on a particular machine or written by a particular developer.
>
> **ArchInstance**: an instance of an architecture, containing components, connectors, links and groups.

More information on xArch can be found at http://www.isr.uci.edu/projects/xarchuci/.

There are a number of differences between instances as described by xArch and instances as described by Acme. These differences are:

- roles and ports are not distinguished in xArch – both are refered to as InterfaceInstances;
- there are no properties attached to any of the elements. It is intended that the description element be used as non-interpreted property elements;
- the hierarchical decomposition of connectors and components is restricted to one choice; in Acme, connectors and components may have multiple representations;
- interfaces (ports and roles) in Acme are allowed to have representations; in xArch, this is not possible; and
- xArch provides a grouping facility that Acme does not support.[2]

It is necessary, therefore, to extend xArch to support these additional Acme features, in addition to support for higher-level Acme concepts such as families and types.

---

[2] Prior to the definition of xArch, there had been some discussion about adding a grouping mechanism to Acme. Watch for a proposal to extend Acme to support groups.

CMU Acme Extensions

The Acme extensions defined by CMU extend the basic xArch structural core to include properties, multiple representations, types, and families. These extensions are expressed in five separate schema definitions that use schema inheritance as the extension mechanism.

The goal of the Acme extensions for xArch are to allow tools that describe architectures using Acme to be able to interchange architectural descriptions with tools that understand xArch. Standard XML parsers and tools can be used to build and save the XML representations, saving developers from having to rewrite these tools for particular architecture languages. The ultimate aim of xArch is to be able to use extensions created by other institutions; however, the current inheritance model of XML schema only allow this in cases where the two schema are non-interfering, i.e., both schema do not extend the same type.

The five Acme extensions to xArch are:

> **properties**: adds simple properties to xArch instances. At this level, properties have no detailed structure; they simply consist of a name, a value and a type, each represented by uninterpreted strings. In addition, support for multiple subarchitectures is included in this extension.
>
> **acmeProperties**: extends properties by adding structure to property types and values consistent with Acme property values and types. Structures for sets, sequences, records, and enumerated types are included.
>
> **families**: adds to properties support for types of components, connectors, interfaces and properties, as well as the collection of all of these types into families. Instances defined in the properties schema are extended to refer to families as supertypes; in addition to this, instances may refer to a particular family used to define the default structure of the instance.
>
> **constraints**: extends families to add support for general constraints. At this level, constraints merely have a type and a description, both of which are uninterpreted strings.
>
> **acmeConstraints**: extends constraints to add support for Armani constraints to instances.

Figure 4 illustrates the hierarchical inheritance structure of the xArch Acme extensions. As mentioned above, xArch is the base schema, representing architectural structure. The arrow in the diagram represents an inheritance and use relation. Thus, tag types defined in the properties extension extend tag types in the xArch schema, and may also compositionally use types defined in xArch. The dashed line represents compositional use only. Thus, *acmeConstraints* uses *acmeProperties* but extends (or inherits from) types in *constraints*.

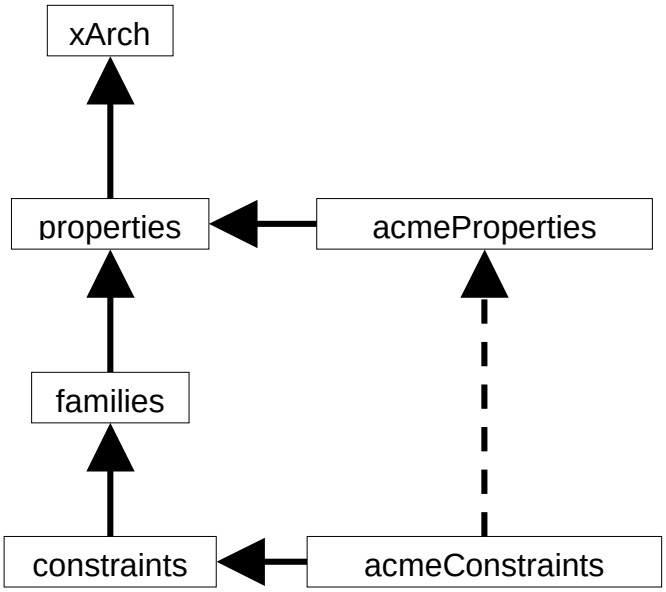

Figure 4. Organization of the xArch Acme extensions

Descriptions

This section describes the five Acme extensions, providing definitions of the added elements and types, and how they inherit from other schema.

Extension 1: properties

The *properties* extension allows design elements to be annotated with properties consisting of name, type and value. In each of these cases, the values are uninterpreted string. It is left to tools to interpret the values. The *acmeProperties* extension gives the XML Schema definition for the abstract syntax of Acme-style properties with structured values and types.

The *properties* extension defines the following new types:

> **PropertyValue**: a type that contains no substructure, but has a string attribute (*description*) that contains the string representation of the value.
>
> **PropertyType**: a type that contains no substructure, but has a string attribute (*description*) that contains the string representation of a type.
>
> **Property**: a type that has a *name* attribute (defining the name of the property) and a sequence containing one each of **PropertyValue** and **PropertyType**.

In addition, the properties extension extends the following xArch types:

> **ComponentInstance**, **ConnectorInstance**, **InterfaceInstance**, **Group**, **ArchInstance**: Each of these is extended to include zero or more `<property>` tags, each of type `Property`.
>
> **ComponentInstance**, **ConnectorInstance**: The arity of the subArchitecture component is extended to include multiple subArchitectures, which model Acme representations.

Example of use

Figure 5 gives two examples of using the properties extension and the associated Acme equivalents. In the XML example, the convention `<extension:tag>` is used to denote the extension in which a tag is defined. Figure 5(a) defines an Acme property as a sequence of integers. The corresponding XML denotation of this Acme property using the properties extension is given in Figure 5(b). Note that the detailed Acme sequence structure is captured within the description attribute of the value and type tags.

Figure 5(c) shows an Acme component definition containing one port and one integer property. The XML in Figure 5(d) shows the use of inheritance to capture this more complex component in xArch.[3] The XML instance of `componentInstance` (whose id is c2) is actually an XML instance of the schema type `properties:componentInstance`. In this way, the properties can be added to a component instance.

Extension 2: acmeProperties

The *acmeProperties* extension adds support for structured property values and types, following the Acme language definition of properties. The `PropertyValue` and `PropertyType` tags, defined as strings in the properties extension, are extended to include this structure.

The `PropertyType` tag type is extended to be a choice of the following types:

> **PropertyPrimitiveType**: This element type is an empty tag that contains a type attribute that can have values restricted to those of primitive Acme types, i.e., `integer`, `float`, `string`, `double`, `long`, `any`, `uri`, or `char`.
>
> **PropertySetType**: This element type is used to represent a type that is an Acme set. This type consists of one element, which refers to another type that represents the type of the members of the set.
>
> **PropertySequenceType**: This element type is used to represent a type that is an Acme sequence. This type consists of one element, which refers to another type that represents the type of the members of the sequence.
>
> **PropertyEnumType**: This element type is used to represent an Acme enumerated type. The type is defined by listing the potential values of the type as strings.
>
> **PropertyRecordType**: This element type defines an Acme record. It is composed of one or more `PropertyFieldTypes`, also defined in this extension.
>
> **XMLLink**: This element type (defined in instances) is used to refer to an already defined type.

[3] Note that this XML would be embedded in a bigger XML document, as is not valid in itself.

```
property events : sequence <int> =
  <12, 1, 14, 5>;
                        (a)

<properties:property name="events">
  <properties:value
    description="<12, 1, 14, 5>"/>
  <properties:type
    description="sequence <int>"/>
<properties:property>
                        (b)

component c2 = {
  port in;
  property throughput : int = 2;
}
                        (c)

<xArch:componentInstance id = "c2"
    xsi:type="properties:ComponentInstance">
  <xArch:description/>
  <xArch:interfaceInstance id="in">
    <xArch:description/>
  </xArch:interfaceInstance>
  <properties:property name="throughput">
    <properties:value description="2"/>
    <properties:type description="int"/>
  </properties:property>
</xArch:componentInstance>
                        (d)
```

Figure 5. Examples of using elements from the properties extension.

The `PropertyValue` tag type is extended to include a choice of the following values, representing literals of the Acme property types.

PropertyPrimitiveValue: XML Schema datatypes are used to define the contents of the corresponding primitive values.

PropertySetValue: This is a list of `PropertyValues`.

PropertySequenceValue: This is a list of `PropertyValues` with an associated sequence number.

PropertyRecordValue: This is a list of `PropertyFieldValues`.

Example of use

Figure 6 shows the definition of an Acme component using the *acmeProperties* extension. This component is the one defined in Figure 5(c). *acmeProperties* is used in the definition of the value and the type. The value is defined on lines 8-16 of Figure 6; the type is defined on lines 17-21. The tag type of the value of the property is the `PropertyValue` from *acmeProperties*. It contains a `propertyPrimitiveValue` which contains an `intValue`. The type of the property is represented by a `PropertyType` defined in the *acmeProperties* extension.

```
1  <xArch:componentInstance id = “c2”
2      xsi:type=”properties:ComponentInstance”>
3    <xArch:description/>
4    <xArch:interfaceInstance id=”in”>
5      <xArch:description/>
6    </xArch:interfaceInstance>
7    <properties:property name=”throughput”>

8      <properties:value
9          xsi:type = “acmeProperties:PropertyValue”
10         description=”2”>
11       <acmeProperties:propertyPrimitiveValue>
12         <acmeProperties:intValue>
13           2
14         </acmeProperties:intValue>
15       </acmeProperties:propertyPrimitiveValue>
16     </properties:value>

17     <properties:type
18        xsi:type=”acmeProperties:PropertyType”
19        description=”int”>
20       <acmeProperties:propertyPrimitiveType type=”int”/>
21     </properties:type>
22   </properties:property>
23 </xArch:componentInstance>
```

Figure 6. Example of using the elements from the acmeProperties extension.

Extension 3: families

The *families* extension extends tag types in the *properties* extension to add support for Acme types and families. The following new tag types are introduced:

ComponentType, ConnectorType, InterfaceType: Each of these types extends the respective instance definition in the *properties* extension. For example, the `ComponentInstance` type defined in *properties* defines the default structure for the `ComponentType` in the *families* extension. In addition, each of these tag types has zero or more links to supertypes.

> **Family**: A family is a collection of element types and an optional default structure. The family may also refer to zero or more super families.
>
> **TypedComponentInstance, TypedConnectorInstance, TypedInterfaceInstance**: These instances extend the element instances defined in the *properties* extension to refer to zero or more types. Additionally, the instances can refer to a particular type to instantiate their default structure, which corresponds to the “`new`” phrase in Acme.
>
> **TypedDesign**: A typed design is a collection of family definitions and system definitions.

```
family pipeFilter = {
  port type Readport;
  port type WritePort;
  component type filter = {
    port stdin : ReadPort;
    port stdout : WritePort;
  }
}
```

(a)

```
1  <families:typedDesign>
2    <families:family id="pipeFilter">
3      <families:interfaceType id="ReadPort"/>
4      <families:interfaceType id="WritePort"/>

5      <families:componentType id="Filter">
6        <families:structure>
7          <xArch:interfaceInstance
8              id="stdin"
9              xsi:type="families:typedInterfaceInstance">
10           <families:type href="#ReadPort"/>
11         </xArch:interfaceInstance>
12         <xArch:interfaceInstance
13             id="stdout"
14             xsi:type="families:typedInterfaceInstance">
15           <families:type href="#WritePort"/>
16         </xArch:interfaceInstance>
17       </families:structure>
18     </families:componentType>

19   </families:family>
20 </families:typedDesign>
```

(b)

Figure 7. The definition of an Acme family and the associated definition using the families extension.

Example of use

Figure 7 illustrates the definition of an Acme family, containing two port types and one component type, in both Acme and xArch:Acme. We will assume that the xArch definition is contained in a file called pipeFilter.xml.

Figure 8 shows the definition of a system that uses the family defined in Figure 7. Figure 8(a) shows an Acme definition, while Figure 8(b) shows the corresponding XML encoding. Note the use of references to the types defined in pipeFilter.xml on lines 9, 10, 18, and 19 of Figure 8(b). These refer to the definitions in Figure 7(b).

```
system simple-demo : pipeFilter = {

  component Capitalize : Filter = new Filter extended with {
    port stdin : ReadPort = new ReadPort;
    port stdout : WritePort = new WritePort;
  }
}
                                (a)


1  <xArch:archInstance id="simple-demo"
2                       xsi:type="families:ArchInstance">
3    <xArch:component id="Capitalize"
4                     xsi:type="families:ComponentInstance">
5      <xArch:description/>
6      <xArch:interfaceInstance id="stdin"
7                               xsi:type="families:InterfaceInstance">
8        <xArch:description/>
9        <families:type href="pipeFilter.xml#ReadPort"/>
10       <families:instanceOf href="pipeFilter.xml#ReadPort"/>
11     </xArch:interfaceInstance>
12     <xArch:interfaceInstance id="stdout"
13                              xsi:type="families:InterfaceInstance">
14       <xArch:description/>
15       <families:type href="pipeFilter.xml#WritePort"/>
16       <families:instanceOf href="pipeFilter.xml#WritePort"/>
17     </xArch:interfaceInstance>
18     <families:type href="pipeFilter.xml#Filter"/>
19     <families:instanceOf href="pipeFilter.xml#Filter"/>
20   </xArch:component>
21 </xArch:archInstance>
                                (b)
```

Figure 8. An example of a component instance using the families extension.

Extension 4: constraints

The *constraints* extension adds unstructured constraints to the elements defined in the *families* extension. In this extension, a constraint is composed of two attributes: a type attribute and a description attribute. The description is the definition of the constraint in plain text. The type, also plain text, is meant to indicate the use of the constraint. In the Acme extensions, this is restricted to heuristic and invariant, but in other extensions could indicate whether a constraint is dynamically evaluated, or even the constraint engine used for evaluation. The *constraints* extension defines the following new types, which add the general constraint's respective definitions in the *families* extension:

- **ConstrainedComponentInstance**,
- **ConstrainedConnectorInstance**,
- **ConstrainedInterfaceInstance**, and
- **ConstrainedArchInstance**.

Example of use

Figure 9 shows the encoding of an Acme constraint using the *constraints* extension. Note that in this figure, the *acmeProperties* extension is not used to encode the properties – only

the *properties* extension is. The constraint is captured in lines 11-13 as two string attributes.

```
component c1 = {
   property latency : int;
   property maxLatency : int = 100;
   invariant latency < maxLatency;
}
                              (a)

1  <xArch:component
2      id="c1"
3      xsi:type="constraints:ConstrainedComponentInstance">

4    <properties:property id="latency">
5      <properties:type description="int"/>
6    </properties:property>

7    <properties:property id="maxLatency">
8      <properties:type description="int"/>
9      <properties:value description="100"/>
10   </properties:property>

11   <constraints:constraint
12       type="invariant"
13       description="latency < maxLatency;"/>
14  </xArch:component>
                              (b)
```

Figure 9. An example of using the constraints extension to model an Acme component with constraints.

Extension 5: acmeConstraints

The *acmeConstraints* extension extends the *constraints* extension to support structure corresponding to Acme constraints.[4] This extension essentially defines an abstract syntax tree for the constraint expression. In addition, it adds facilities for defining design analyses – functions that can be defined and used in constraints. Because this extension essentially follows the grammar of the Armani predicate language, it is quite large, and so all of the new types defined in this extension are not enumerated. At the top level, the new tag types introduced at this level are:

> **DesignAnalysisDeclaration**: This tag type begins the declaration of a design analysis function, which has a name, some parameters, a return type and a definition.
>
> **DesignRule**: A `DesignRule` extends the `Constraint` tag type from *Constraints*. It makes two changes: the first is to add a tag that contains the XML encoding of the constraint structure, a choice of a quantified expression or a boolean expression; the second is that it restricts the type attribute of the tag type to contain one of two possible values: heuristic or invariant.

[4] In this document we equate Acme constraints with Armani constraints.

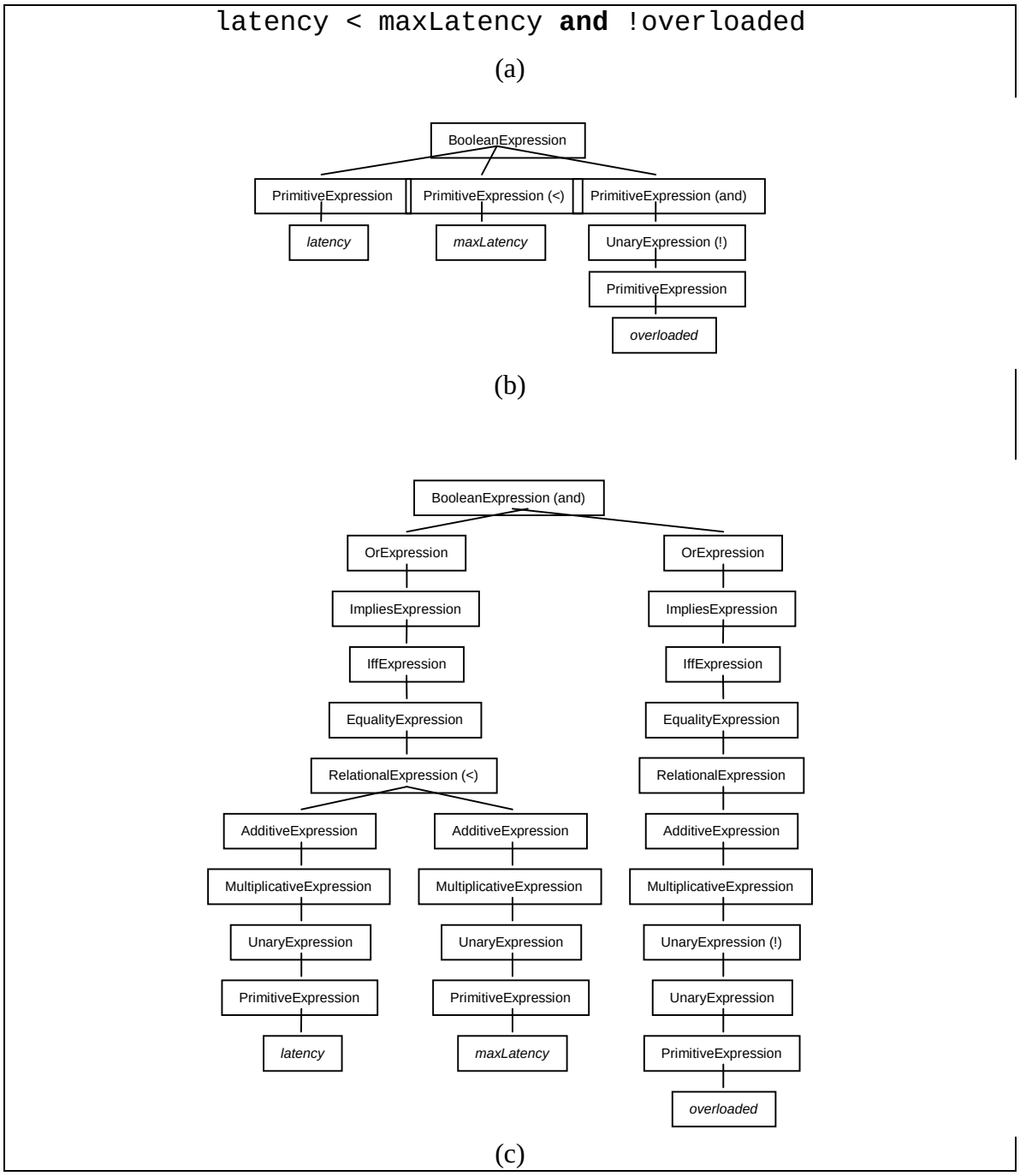

Figure 10. Example of expression flattening.

A complete encoding of the syntax of a constraint would be too unwieldy in practice because the grammar definition has a deep nesting structure that encodes operator precedence. To avoid the deep structure in the XML, a Boolean expression is a sequence of primitive expressions, each of which has an attribute that defines the relation (or connector) between the primitive expression containing the connector and the previous

```
component c1 = {
  port stdin {property latency : int = 50;};
  port stdout … {property latency : int = 75;};
  invariant forall p : port in self.ports |
    p.latency < 100;
```

(a)

```
1  <xArch:componentInstance id=”c1”
2      xsi:type=”constraints:ConstrainedComponentInstance”>
3    <xArch:interfaceInstance id=”stdin”
4        xsi:type=”properties:InterfaceInstance”>
5      <properties:property id=”stdin.latency”>
6         <properties:value desription=”50”/>
7         <properties:type description=”int”/>
8      </properties:property>
9    </xArch:interfaceInstance>
10   <xArch:interfaceInstance id=”stdout”
11       xsi:type=”properties:InterfaceInstance”>
12     <properties:property id=”stdout.latency”>
13        <properties:value desription=”75”/>
14        <properties:type description=”int”/>
15     </properties:property>
16   </xArch:interfaceInstance>
17   <constraints:constraint xsi:type=”acmeConstraints:DesignRule”
18       type=”invariant”>
19     <acmeConstraints:expression>
20       <acmeConstraints:quantifiedExpression>
21         <acmeConstraints:quantifier id=”p”>
22           <acmeConstraints:predefinedType>
23             Port
24           </acmeConstraints:predefinedType>
25         </acmeConstraints:quantifier>
26         <acmeConstraints:setExpression set=”interfaces”>
27           <acmeConstraints:element href=”#c1”/>
28         </acmeConstraints:setExpression>
29         <acmeConstraints:expression>
30           <acmeConstraints:booleanExpression>
31             <acmeConstraints:primitiveExpression>
32               <acmeConstraints:quantifierAccess access=”latency”>
33                 <acmeConstraints:quantifier href=”#p”/>
34               </acmeConstraints:quantifierAccess>
35             </acmeConstraints:primitiveExpression>
36             <acmeConstraints:primitiveExpression connector=”lt”>
37               <acmeConstraints:literalValue>
38                 <acmeConstraints:intValue>100</intValue>
39               </acmeConstraints:literalValue>
40             </acmeConstraints:primitiveExpression>
41           </acmeConstraints:booleanExpression>
42         </acmeConstraints:expression>
43       </acmeConstraints:quantifiedExpression>
44     </acmeConstraints:expression>
45   </constraints:constraint>
46 </xArch:component>
```

(b)

Figure 11. An example of using acmeConstraints

primitive expression. Thus, the expression tree is flattened in the XML definition, and it is left to tools to handle the operator precedence.

Figure 10 provides an example of how the expression flattening works. The expression in Figure 10(a) is flattened in the XML to look like the expression tree in Figure 10(b), rather than the deep expression tree represented in Figure 10(c), and corresponding to the Acme syntax.

Example of use

Figure 11 gives an example of defining an Acme component containing constraints using the *acmeConstraints* extension. Note that the extension point for *acmeConstraints* is at the level of constraints, rather than extending the component. This means that other extensions can be written that extend components but need not use the *acmeConstraints*.

Comparison to ADML

ADML is an XML encoding of Acme that was developed by MCC and adopted by the Open Group as a standard for architectural representation. There are a number of differences between ADML and xAcme:

- ADML encodes Acme 1.0, whereas xAcme embodies the extensions to the Acme included in Acme 3.0. Primarily, these include facilities for encoding architectural design constraints and design analyses.
- ADML is defined using a DTD, whereas xAcme is defined using XML Schema. Using XML Schema provides for a layered approach to the schema which affords easier extension in the future.
- ADML uses unparsed strings to represent references to other Acme entities, whereas xAcme uses Xlinks. For example, if a component needs to refer to a type, the name is encoded inside an additional XML tag in ADML, but is referred to using Xlinks in xAcme.

Despite these differences, we believe that xAcme is an evolution of ADML, and where feasible, we have attempted to use the same tag names in xAcme as appear in ADML.