BiCCL
The Biometric Client Configuration Language

Matthew Aronoff
matthew.aronoff@nist.gov

Ross J. Micheals
ross.micheals@nist.gov

National Institute of Standards and Technology
Gaithersburg MD 20899, USA

Abstract

In this paper, we introduce the Biometric Client Configuration Language, or BiCCL. BiCCL is a platform-independent, domain-specific language (DSL) that formally describes the biometric acquisition process. BiCCL uses high-level constructs—e.g., 'sensors', 'tasks', 'experimental conditions', and 'workflow' —to help users directly encode their intent. To test the expressiveness of this new DSL, we built a reference compiler that translates BiCCL directly into XML that can be consumed by the NIST Multimodal Biometric Application Resource Kit (MBARK).

Problem & Background

With the falling cost of biometric authentication hardware (fingerprint scanners, iris scanners, digital cameras, etc.) and the increased integration of biometrics within daily computer interaction, multimodal biometric authentication is being used in a broadening range of identity verification tasks. A self-serving example would be the National Institute of Standards and Technology's (NIST) Multimodal Biometric Application Resource Kit (MBARK), available at http://mbark.nist.gov.

Typically, a biometric client's *workflow*—the logic of what happens and when—is hard coded, or 'baked-in,' to a biometric application. The result is a software package that lacks flexibility if system owners wish to change sensors that are used, the order that they are presented, and the metadata that is used to dynamically affect the flow control of the running application.

MBARK enables such flexibility by not hard-coding workflow information, but by incorporating workflow into data structures used by MBARK at runtime. A custom serialization architecture allows those data structures to be saved out to disk in XML. The result is an extremely flexible configuration—but one that is tied to the particular design decisions made during the development of MBARK.

Because these XML files are more reflections of MBARK's internal *state* than they are a reflection of the indented workflow, they are difficult to generate by hand. Minor changes in workflow—such as

reordering tasks—are possible via transpositions of XML elements and may be performed in a text editor. Advanced workflow changes—such as adding a new Boolean flag that alters flow control—requires insight into the MBARK workflow data structures and XML sorcery.

Because of this, historically, the easiest way of creating a new MBARK XML configuration file was to use the MBARK source code to write a small 'bootstrapping' application that would programmatically generate the desired workflow. By serializing these data structures out to disk, the corresponding MBARK XML configuration file was generated. Although this approach is sufficient for many some users, it requires a full source-code snapshot of MBARK—publicly available, but extremely inconvenient for a large number of users. Those who desired the utmost simplicity faced an extremely steep learning curve.

BiCCL (pronounced "*bicycle"*), or the Biometric Client Configuration Language, was created to address those shortcomings. It is a higher-level, human readable, Domain-Specific Language (DSL) for biometric workflows. We have created both the BiCCL language and a converter that generates MBARK XML configuration files from BiCCL files; the two were developed simultaneously on the grounds that having a functional converter would prove that the BiCCL language was sufficiently descriptive for our needs. However, BiCCL was not designed to be MBARK specific—it is a *domain* specific, not an *application* specific, language.

BiCCL Design

A generally-applicable language for biometric workflows needed to provide three main capabilities:

- The modular use and configuration of both new and previously-defined sensors (i.e., no recompiling required for the software itself when new sensors become available).
- The ability to describe and record, at runtime, arbitrary metadata about what the sensors are capturing.
- The concept of a workflow as a sequence of tasks, with the option of conditional branching.

There were two major design decisions that were made to support these capabilities—one regarding the semantics of the DSL and one regarding the syntax.

Semantically, BiCCL is designed to directly reflect the desired workflow. As stated previously, direct storage of MBARK's internal state does not allow for a natural expression of the user's intent. A biometric workflow needs to use obvious and simply-structured logic. For example, users need to be able to easily describe logic such as "if the subject is carrying glasses, take one picture with glasses on, and one with glasses off. Otherwise, just take two pictures." The internal representation of this construct is irrelevant from the user's point-of-view.

Syntactically, we desired a language that was directly editable in a standard text editor. Therefore, the DSL was designed to take on a 'C'-like syntax. We did not choose XML because typical language constructs—declaring variables, defining control flows—do not cleanly map to readable XML. For example, consider the Microsoft Visual Studio DSL Tools (S. Cook, 2007). Although these tools produce DSLs represented in XML, they are designed to be edited and manipulated in complimentary *graphical* DSLs.

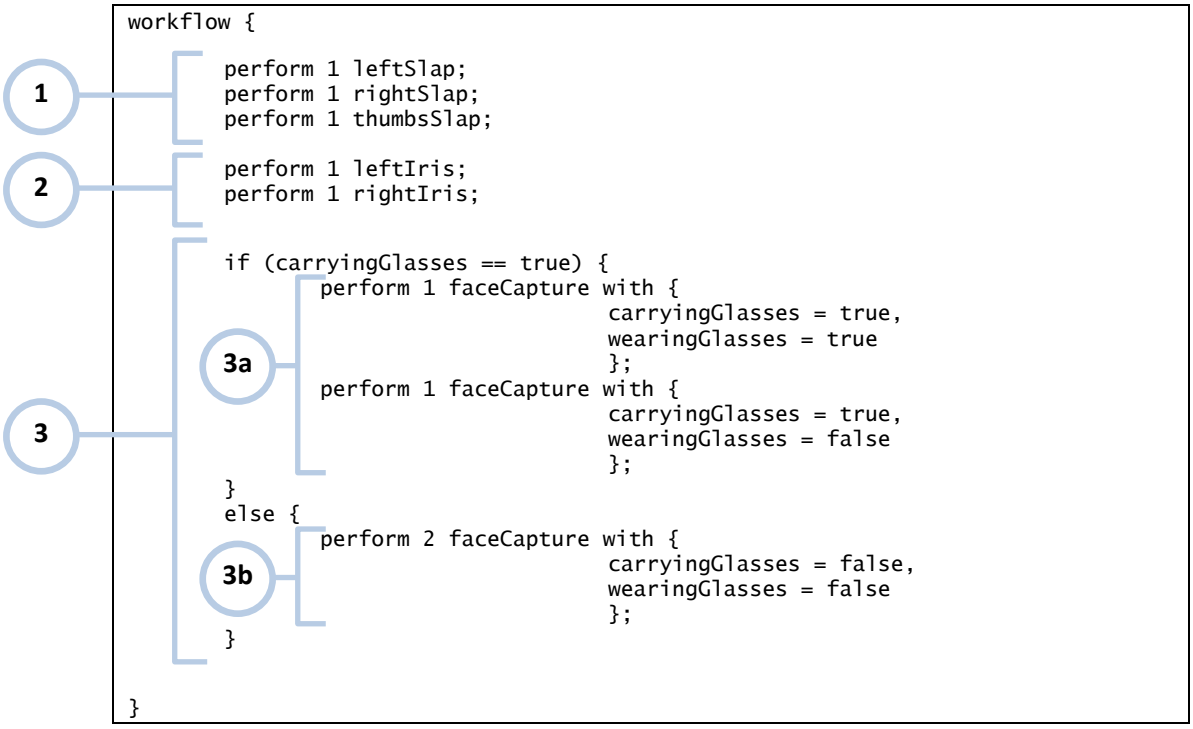
A disadvantage of this approach is that standard XML tools cannot be used to parse or transform BiCCL files. However, since our reference BiCCL complier transforms configuration files *into* MBARK XML, using XML tools is still an option, although the additional complexities of understanding how MBARK stores system state accompanies such a choice. In the future, should such a need arise, an XML-specific version of BiCCL could be crafted. However, given our experiences developing BiCCL, and given the graphical nature in which current XML-based DSLs are typically used, the authors do not anticipate this need.

In summary, to satisfy these semantic and syntactic constraints, we determined that we needed a domain-specific language (DSL), rather than a data serialization markup such as the existing XML structure used by MBARK.

BiCCL Capabilities

To illustrate some of the advantages of using BiCCL, consider a BiCCL code snippet with its XML counterpart. This comparison is not meant to be an evaluation of the capabilities of XML. It is a direct, high-level comparison of how the two formats (BiCCL versus MBARK XML) describe a sequential, task-based, workflow.

Here is an illustrated BiCCL snippet that defines the core of a desired workflow:



The section labeled '1' describes a slap fingerprint sequence—a left slap, followed by a right slap, followed by a two-thumb slap. Likewise, the section marked '2' indicates a left iris capture followed by a right iris capture. Section '3' illustrates conditional (if-else) tasking based on user-definable metadata (`carryingGlasses` in this case). Section '3a' describes what should happen when the subject is carrying glasses—one image should be taken with their glasses on, and then one with their glasses off. Section '3b' describes the tasks to be performed if the subject is *not* carrying glasses—two face images with glasses off.

Each task (`faceCapture`, `leftSlap`, `rightSlap`, etc.) references a definition (not shown) that specifies which sensor it uses, desired parameters for that sensor, and other information (described in detail later). The result is that end users can very easily copy and paste a workflow into another BiCCL file that

uses the same sensors. Furthermore, with this syntax it is immediately clear how to reorder tasks—just reorder the lines of text.

As a comparison, here is a small piece of the original XML workflow definition. It defines only the first task within the `if` statement (3), the `leftSlap` and `rightSlap` workflow items, and would have to be copied, pasted, and potentially reordered for a new workflow definition:

```
<TaskFactory>
        <TargetConditions>
                <EquivalenceClasses Name="NotCarryingGlassesEquivalence"/>
                <FactoryProxies Name="wearingGlasses"/>
        </TargetConditions>
        <SensorType>Mbark.FaceCamera.dll+Mbark.Sensors.FaceCamera</SensorType>
        <TaskPrerequisite PredicateType="ConditionEqualsLiteralPredicate">
                <FactoryArgument xsi:type="xsd:string">carryingGlasses</FactoryArgument>
                <FactoryArgument xsi:type="xsd:boolean">true</FactoryArgument>
                <UnmetPrerequisiteMessage Name="Empty"/>
        </TaskPrerequisite>
        <Category>Face</Category>
        <TaskCount>1</TaskCount>
        <SensorConfiguration xsi:type="FaceCameraConfiguration">
                <CaptureCriticalTime>0</CaptureCriticalTime>
                <CaptureExpirationTime>200000</CaptureExpirationTime>
                <DownloadExpirationTime>150000</DownloadExpirationTime>
                <Zoom>15</Zoom>
                <Resolution>16</Resolution>
                <Flash>2</Flash>
                <LCD>2</LCD>
                <Exposure>3</Exposure>
                <WhiteBalance>4</WhiteBalance>
                <PrecaptureTimeout>12000</PrecaptureTimeout>
                <PostcaptureTimeout>12000</PostcaptureTimeout>
                <TimeoutPollingInterval>3000</TimeoutPollingInterval>
        </SensorConfiguration>
        <ReassignableCategories>
                <TaskCategories/>
        </ReassignableCategories>
</TaskFactory>
<TaskFactory>
        <SensorType>Mbark.FingerprintScanner.dll+Mbark.Sensors.FingerprintScanner</SensorType>
        <Category>LeftSlap</Category>
        <TaskCount>1</TaskCount>
        <SensorConfiguration xsi:type="FingerprintScannerConfiguration">
                <CaptureCriticalTime>3000</CaptureCriticalTime>
                <CaptureExpirationTime>12000</CaptureExpirationTime>
                <DownloadExpirationTime>0</DownloadExpirationTime>
        </SensorConfiguration>
        <ReassignableCategories>
                <TaskCategories>
                        <SensorTaskCategory>LeftSlap</SensorTaskCategory>
                        <SensorTaskCategory>RightSlap</SensorTaskCategory>
                        <SensorTaskCategory>ThumbsSlap</SensorTaskCategory>
                </TaskCategories>
        </ReassignableCategories>
</TaskFactory>
<TaskFactory>
        <SensorType>Mbark.FingerprintScanner.dll+Mbark.Sensors.FingerprintScanner</SensorType>
        <Category>RightSlap</Category>
        <TaskCount>1</TaskCount>
        <SensorConfiguration xsi:type="FingerprintScannerConfiguration">
                <CaptureCriticalTime>3000</CaptureCriticalTime>
                <CaptureExpirationTime>12000</CaptureExpirationTime>
                <DownloadExpirationTime>0</DownloadExpirationTime>
```

```
        </SensorConfiguration>
        <ReassignableCategories>
                <TaskCategories>
                        <SensorTaskCategory>LeftSlap</SensorTaskCategory>
                        <SensorTaskCategory>RightSlap</SensorTaskCategory>
                        <SensorTaskCategory>ThumbsSlap</SensorTaskCategory>
                </TaskCategories>
        </ReassignableCategories>
</TaskFactory>
```

Although we did not perform any formal usability testing, we suspect that this XML is much more difficult to interpret than the BiCCL fragment. A BiCCL workflow uses references to tasks, to localized versions of strings, and to sensors so that the user can manipulate the workflow at more natural levels of abstraction.

Finally, building BiCCL as a DSL allows us to clearly separate the domain (of biometric capture) from the implementation (the specific driving application—MBARK in our case). The reference BiCCL compiler currently has the ability to export an XML file that can be used to control MBARK; other plug-ins can be added to support different existing biometric authentication tools, but the core language can remain unchanged.

Example Document & Overview

The following is a sample BiCCL that we will examine in more detail. The non-whitespace lines have been numbered for ease of reference.

```
 1:  localizations {
 2:       SubjectCarryingGlasses {
 3:               en-US = "Subject is carrying glasses.";
 4:               en-UK = "Subject is carrying spectacles.";
 5:               de-DE = "Thema trägt Gläser.";
 6:       }
 7:       SubjectNotCarryingGlasses {
 8:               en-US = "Subject is not carrying glasses.";
 9:               en-UK = "Subject is not carrying spectacles.";
10:               de-DE = "Thema trägt nicht Gläser.";
11:       }
12:       SubjectIsWearingGlasses {
13:               en-US = "Subject is wearing glasses.";
14:               en-UK = "Subject is not wearing spectacles.";
15:               de-DE = "Thema trägt Gläser.";
16:       }
17:       SubjectIsNotWearingGlasses {
18:              en-US = "Subject is not wearing glasses.";
19:              en-UK = "Subject is not wearing spectacles.";
20:              de-DE = "Thema trägt nicht Gläser.";
21:      }
22:      GlassesOn {
23:              en-US = "Glasses on.";
24:              en-UK = "Spectacles on.";
25:              de-DE = "Gläser an.";
26:      }
```

```
27:         GlassesOff {
28:                 en-US = "Glasses off.";
29:                 en-UK = "Spectacles off.";
30:                 de-DE = "Gläser weg.";
31:         }
32:         SubjectMustBeCarryingGlasses {
33:                 en-US = "Subject must be carrying glasses.";
34:                 en-UK = "Subject must be carrying spectacles.";
35:                 de-DE = "Thema muß Gläser tragen.";
36:         }
37:     }
38:     experimentalConditions {
39:         static boolean carryingGlasses {
40:                 standaloneMessages {
41:                         trueMessage    = SubjectCarryingGlasses;
42:                         falseMessage   = SubjectNotCarryingGlasses;
43:                         invalidMessage = none;
44:                 }
45:                 compoundMessages {
46:                         trueMessage    = none;
47:                         falseMessage   = none;
48:                         invalidMessage = none;
49:                 }
50:
51:                 requires { modality == "face" || modality == "iris"; }
52:                 unmetPrerequisiteMessage = none;
53:         }
54:         boolean wearingGlasses {
55:                 initialValue = false;
56:                 standaloneMessages {
57:                         trueMessage    = SubjectIsWearingGlasses;
58:                         falseMessage   = SubjectIsNotWearingGlasses;
59:                         invalidMessage = none;
60:                 }
61:                 compoundMessages {
62:                         trueMessage    = GlassesOn;
63:                         falseMessage   = GlassesOff;
64:                         invalidMessage = none;
65:                 }
66:                 requires { carryingGlasses == true; }
67:                 unmetPrerequisiteMessage = SubjectMustBeCarryingGlasses;
68:         }
69:     }
70:     equivalenceClasses {
71:         NotCarryingGlassesEquivalence{
72:                 ~carryingGlasses || (carryingGlasses && ~wearingGlasses);
73:         }
74:     }
75:     sensors {
76:         FaceCamera {
77:                 libraryFileName        = "Mbark.FaceCamera.dll";
78:                 sensorClassName        = "Mbark.FaceCamera";
79:                 configurationClassName = "FaceCameraConfiguration";
80:         }
81:         FingerprintScanner {
82:                 libraryFileName        = "Mbark.FingerprintScanner.dll";
83:                 sensorClassName        = "Mbark.FingerprintScanner";
84:                 configurationClassName = "FingerprintScannerConfiguration";
85:         }
86:         IrisScanner {
87:                 libraryFileName        = "Mbark.IrisScanner.dll";
88:                 sensorClassName        = "Mbark.IrisScanner";
89:                 configurationClassName = "IrisScannerConfiguration";
90:         }
91:     }
```

```
configurations {
    FaceCameraConfiguration FaceCameraConf {
            CaptureCriticalTime   = 0; // 0 sec
            CaptureExpirationTime = 200000; // 200 sec
            DownloadExpirationTime = 150000; // 150 sec
            Zoom                  = 15;
            Resolution            = 16;
            Flash                 = 2;
            LCD                   = 2;
            Exposure              = 3;
            WhiteBalance          = 4;
            PrecaptureTimeout     = 12000;
            PostcaptureTimeout    = 12000;
            TimeoutPollingInterval = 3000;
    }
    FingerprintScannerConfiguration FingerprintScannerConf {
            CaptureCriticalTime    = 3000; // 3 sec
            CaptureExpirationTime  = 12000; // 12 sec
            DownloadExpirationTime = 0; // 0 sec
    }
    IrisScannerConfiguration IrisScannerConf {
            CaptureCriticalTime    = 0;
            CaptureExpirationTime  = 20000;
            DownloadExpirationTime = 0;
            FocusThreshold         = 12;
            FileSystemWatcherDir   = “tmp\images”;
            RegistryName           = "IrisScannerRegistryName";
    }
}
tasks {
    faceCapture {
            sensor                    = FaceCamera;
            configuration             = FaceCameraConf;
            intendedSubmodality       = "Face";
            reassignableSubmodalities = none;
    }
    leftSlap {
            sensor                    = FingerprintScanner;
            configuration             = FingerprintScannerConf;
            intendedSubmodality       = "LeftSlap";
            reassignableSubmodalities = {
                    "LeftSlap", "RightSlap", "ThumbsSlap"};
    }
    rightSlap {
            sensor                    = FingerprintScanner;
            configuration             = FingerprintScannerConf;
            intendedSubmodality       = "RightSlap";
            reassignableSubmodalities = {
                    "LeftSlap", "RightSlap", "ThumbsSlap"};
    }
    thumbsSlap {
            sensor                    = FingerprintScanner;
            configuration             = FingerprintScannerConf;
            intendedSubmodality       = "ThumbsSlap";
            reassignableSubmodalities = {
                    "LeftSlap", "RightSlap", "ThumbsSlap"};
    }
    leftIris {
            sensor                    = IrisScanner;
            configuration             = IrisScannerConf;
            intendedSubmodality       = "LeftIris";
            reassignableSubmodalities = {
                    "LeftIris", "RightIris"};
    }
    rightIris {
            sensor                    = IrisScanner;
```

```
158:              configuration             = IrisScannerConf;
159:              intendedSubmodality       = "RightIris";
160:              reassignableSubmodalities = {
161:                      "LeftIris", "RightIris" };
162:      }
163:   }
164:   initialState {
165:      carryingGlasses = true;
166:      wearingGlasses  = false;
167:   }
168:   workflow {
169:      if (carryingGlasses == true) {
170:              perform 1 faceCapture with {
171:                                        carryingGlasses = true,
172:                                        wearingGlasses = true
173:              };
174:              perform 1 faceCapture with {
175:                                        carryingGlasses = true,
176:                                        wearingGlasses = false
177:              };
178:      } else {
179:              perform 2 faceCapture with {
180:                                        carryingGlasses = false,
181:                                        wearingGlasses = false
182:              };
183:      }
184:      perform 1 leftSlap;
185:      perform 1 rightSlap;
186:      perform 1 thumbsSlap;
187:
188:      perform 1 leftIris;
189:      perform 1 rightIris;
190:   }
```

Lines 1–37 describe messages that the system may display for the user. Each message has a set of localizations (translated versions of the message) for at least one language.

Lines 38–69 describe two experimental conditions that apply to the capture process. (This is where the user defines their custom metadata.) On line 39, we define a condition called `carryingGlasses`; it is a Boolean, and since it's unlikely to change during the capture process (the subject either brought glasses with them or they didn't), we declare it as `static`. Lines 40–49 define the messages that the system will use when this condition is in various states —when this condition is true, and is displayed in conjunction with other the value defined on line 41 is what the user sees. When the condition is false, and displayed in conjunction with at least one other condition, the system uses the value set on line 47 (in this case, nothing). On line 51, we specify what is required for `carryingGlasses` to be considered *valid*. This experimental condition only makes sense if the modality is face or iris. The next line, 52, describes the string to display when the condition is not valid—i.e., the 'requires' clause on line 51 is not fulfilled.

Lines 54 though 69 define the `wearingGlasses` condition.

Beginning on line 75, we desribe the particular sensors to be used in this workflow. In this file, there are three, named (rather indicatively) `FaceCamera`, `FingerprintScanner`, and `IrisScanner`. In the first sensor

definition, lines 77–79 tell the system where to find the plug-in (and what the proper sensor class name is within the plug-in) used to control that sensor. The remaining sensors are defined in lines 81–91.

Lines 92–120 are sets of parameters that to be applied, at runtime, to the sensors defined in lines 75-91. The allowable parameters are defined by the sensor plug-ins, so the `FaceCameraConfiguration` has parameters specific to cameras (zoom, flash, etc.), while the `FingerprintScannerConfiguration` does not. Configurations are separate entities so that different configurations could be applied to different tasks within the same workflow. For example, one face capture task might have a close zoom, while another has a wide angle.

Lines 121–163 describe the desired tasks (finally!) that will later be assembled into a workflow. Each task references a sensor, a sensor configuration, and the type of biometric data being captured. The `intendedSubmodality` indicates what biometrics the sensor should capture. The `reassignableSubmodality` indicates how the captured data might be re-categorized given a mistake. Consider the `leftIris` task defined on line 149. Line 150 defines the sensor to be used, 151 points to a particular set of parameters to be used with that sensor, and 152 describes the type of data to be captured. The final two lines (153–154) indicate that the data could be valid as either left iris or right iris data, and can be reassigned to either of those categories later.

Lines 164–167 define the starting values for the experimental conditions from lines 38–69. Our default state is a subject that has glasses with them (line 165), but isn't wearing them (line 166).

Finally, starting on line 168, we define the workflow, which is nearly identical to our previous example. The order in which the tasks are defined in this section is the order in which they should be performed. Line 169 tells the system to check the value of the `carryingGlasses` condition. If that condition is true, the system performs the `faceCapture` task (defined on line 122) one time with the recorded conditions set to the values on lines 171 and 172, then again with the values on lines 175 and 176. Afterwards, from lines 184 through 189, we tell the system to perform each listed task a certain number of times.

BiCCL Syntax & Structure

Syntax

The basic syntax for BiCCL files is fairly simple, and is loosely based on a C-style syntax. Single statements (e.g. assignments, workflow declarations) are followed by a semicolon. Depending on context, single statements may be grouped together in blocks. Blocks are enclosed in curly braces; each block is named, and certain blocks may also have a type declaration and/or a "static" declaration, which indicates that the data described in the block is unlikely to change during workflow execution. Comments are allowed in C++/Java style. Single-line comments

begin with `//`; multi-line comments begin with `/*` and terminate with `*/`. When available, the unary `not` operator is ~.

Appendix A contains a formal description of BiCCL in Extended Backus–Naur Form (EBNF).

Structure

The BiCCL file is split into eight main blocks, which may appear in any order. Each of the following blocks is required:

`localizations`

A collection of localized string definitions used for display. Each localized string is a named block that contains a set of country codes matched with their corresponding messages.

`experimentalConditions`

The metadata that describes variables involved in the capture process. Each experimental condition contains

- an optional static declaration that indicates whether or not the condition is expected to change over the course of a single workflow
- a type declaration. Currently, BiCCL only supports Boolean experimental conditions. (Certainly other data types could easily be added.)
- a named block containing display messages and the condition's prerequisites.

`equivalenceClasses`

A logical expression that describes equivalence between two sets of conditions that might otherwise not be identical.

`sensors`

A collection of biometric sensors. Each sensor is a named block named that contains the plug-in filename, the sensor's classname, and what type of configuration object the sensor can accept. The reference BiCCL parser uses this information to perform strong type checking.

`configurations`

A collection of sensor-specific runtime parameters. Each configuration is a typed and named block. The configuration block's data-type should match a configuration classname referenced in the `sensors` section. Likewise, the contents of the block should also be type-appropriate.

`tasks`

A collection of named, single-sensor operations. Each task is a named block that contains a specific sensor to be used, a configuration to be used with that sensor, the sub-modality to be captured, and a list of sub-modalities to which the data could be re-categorized.

`initialState`

A collection of key-value pairs to be used as the default state of a workflow execution. The variable names correspond to `experimentalCondition` names. Values must conform to the condition's datatype. (Currently, only Booleans.)

`workflow`

An ordered list of tasks. This is where the biometric client's behavior is actually defined. Each line specifies a number of repetitions, and a particular task to perform. Conditional `if-else` blocks are supported; they can be used to determine whether a particular experimental condition has been satisfied prior to performing one (or several) tasks.

Predicates

A predicate is a logical expression that evaluates to either true or false. Predicates appear in

1. within the `if` of an `if-else` conditional (`workflow` section)
2. the `requires` block of an experimental condition definition (`experimentalConditions` section)
3. the body of each equivalence class (`equivalenceClasses` section)

These predicates are expected to be evaluated at runtime to determine (correspondingly)

1. if the system should branch to a different task
2. if the prerequisite of an experimental condition has been met
3. if two sets of conditions should be considered equal

Predicates are nestable and may take any of the following forms:

Predicate Type	Examples
literal	`true` `false`
standard evaluation	`carryingGlasses == true` `wearingGlasses == false`

short-form evaluation (for Boolean conditions only)	`carryingGlasses`
	`~wearingGlasses`
	
	Note: Invalid conditions evaluate to *true*

compound && for AND, || for OR	`carryingGlasses && ~wearingGlasses` `carryingGlasses == false || carryingGlasses && ~wearingGlasses`
parenthesized	`(carryingGlasses || wearingGlasses) && subjectHasEyes` `~(carryingGlasses && wearingGlasses)`

modality or submodality	`submodality == "Face"`
	`submodality in { "rightIris", "leftIris" }`
	`modality == "Face"`
	
	Notes
	• Both `modality` and `submodality` are reserved words.
	• The set-style notation is only valid for the `submodality` keyword.

Note that the && operator has a higher order of precedence than the || operator. For example,

```
A || B && C
```

is evaluated as

```
A || (B && C)
```

and *not* as

```
(A || B) && C
```

Equivalence Classes

An equivalence class is a predicate that may be used to indicate that two different sets of conditions are equivalent that might otherwise not be. Using compounds predicates allow the expression of arbitrarily complex logic. Equivalence classes facilitate the sharing of data among tasks with different experimental conditions that are *semantically* equivalent.

Formally, given two condition sets, if the equivalence class predicate evaluates to *true* for both condition sets, then those conditions (that participate in the equivalence class predicate) are considered to be equivalent.

Here is an example equivalence class with two Boolean experimental conditions. The first, `carryingGlasses` represents the condition where the participant has their glasses (whether they are wearing them or not). The second, `wearingGlasses` (which, as implemented in our example on line 66, depends on `carryingGlasses` being true) is self-explanatory.

```
NotCarryingGlassesEquivalence {

        ~carryingGlasses || (carryingGlasses && ~wearingGlasses);

}
```

This statement implies that "not carrying glasses is the same as carrying glasses and not wearing glasses." If, for each of two different condition sets,

1. `carryingGlasses` is false, or
2. `carryingGlasses` is true and `wearingGlasses` is false,

then the conditions `carryingGlasses` and `wearingGlasses` will be treated as equivalent for the sake of the comparison. In a sense, the *or* operator manifests as an *equals*.

Known Issues

There are four major known issues and problems with the current BiCCL design as described in this paper. These issues are not due to any fundamental limitation on BiCCL, but have evolved out of our own internal use of BiCCL and MBARK.

First, there is a placeholder in the EBNF grammar to allow for the inclusion of raw XML sensor configuration data instead of '='-delimited name/value pairs. Second, the grammar does not yet support nested if statements. Third, the experimental conditions do not support any datatype other than

Boolean conditions. Finally, the EBNF was designed to best facilitate the ANTLR-Java bindings, and does not necessarily represent the most compact or elegant grammar.

The authors are interested in feedback about the severity of these existing limitations.

Conclusion & Future Steps

Implementing BiCCL as a domain-specific language has allowed us to create a high-level, human-readable file format for biometric workflows. We believe that it is simpler to add new sensors to the system using BiCCL, as well as simpler and less error-prone to restructure the actual authentication workflow. In the future, it would be beneficial to add additional output plugins to the BiCCL parser (to allow, for example, WSDL output for a web-services-based authentication session). The ability to combine multiple BiCCL files into a new workflow, as well as to integrate several fragmentary BiCCL files into one complete workflow, will also be a priority.

Acknowledgements

The authors would like to acknowledge Terence Parr for his ANTLR parsing framework and Kayee Kwong of NIST for her valuable feedback.

About the Reference Compiler

The reference BiCCL compiler was written in ANTLR 3.0 (Parr, 2007) targeting the Java 5.0+ runtime. Readers can e-mail *mbark@nist.gov* to obtain a copy of the reference BiCCL compiler or check the MBARK website, http://mbark.nist.gov.

References

Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Raleigh, NC: The Pragmatic Bookshelf.

S. Cook, G. J. (2007). *Domain-Specific Development with Visual Studio DSL Tools.* Addison-Wesley.

Appendix A: Extended Backus-Naur Form (EBNF) for BiCCL

```
//---------------------------
// Initial production
//---------------------------

                      prog ::= section+ ;

//---------------------------
// Section definitions
//---------------------------

                   section ::= 'localizations'          BLOCK_OPEN localization_def+ BLOCK_CLOSE
                             | 'experimentalConditions' BLOCK_OPEN condition_def+    BLOCK_CLOSE
                             | 'equivalenceClasses'     BLOCK_OPEN equiv_def+        BLOCK_CLOSE
                             | 'sensors'                BLOCK_OPEN sensor_def+       BLOCK_CLOSE
                             | 'configurations'         BLOCK_OPEN config_def+       BLOCK_CLOSE
                             | 'tasks'                  BLOCK_OPEN task_def+         BLOCK_CLOSE
                             | 'initialState'           BLOCK_OPEN init_state_def+   BLOCK_CLOSE
                             | 'workflow'               BLOCK_OPEN workflow_def+     BLOCK_CLOSE;


//---------------------------
// Localization productions
//---------------------------
           localization_def ::= localization_name BLOCK_OPEN localization_assignment* BLOCK_CLOSE;
          localization_name ::= block_id;
     localization_assignment ::= string_literal_assignment;

//---------------------------
// Condition productions
//---------------------------
              condition_def ::= condition_signature condition_name
                                  BLOCK_OPEN (condition_initval
                                  | condition_message_assignment
                                  | condition_unmet_prereq
                                  | condition_requires )+
                                 BLOCK_CLOSE;
         condition_signature ::= 'static'? RETURN_TYPE;
             condition_name ::=  block_id;
          condition_initval ::= 'initialValue' '=' bool EOL;
condition_message_assignment ::= condition_message_type BLOCK_OPEN
                                   (condition_localization_assignment EOL)+ BLOCK_CLOSE;
      condition_unmet_prereq ::= 'unmetPrerequisiteMessage' '=' unmet_prereq_message EOL;
          condition_requires ::= 'requires' BLOCK_OPEN predicate_factory EOL? BLOCK_CLOSE;
      condition_message_type ::= 'standaloneMessages'
                               | 'compoundMessages';

        unmet_prereq_message ::= name_or_string;

                  equiv_def ::= block_id BLOCK_OPEN predicate_factory EOL? BLOCK_CLOSE EOL?;

                 sensor_def ::= sensor_name BLOCK_OPEN sensor_plugin sensor_class
                                  sensor_conf_class BLOCK_CLOSE;
                sensor_name ::= block_id;
               sensor_plugin ::= 'libraryFileName'        '=' name_or_string EOL;
                sensor_class ::= 'sensorClassName'        '=' name_or_string EOL;
           sensor_conf_class ::= 'configurationClassName' '=' name_or_string EOL;

//---------------------------
// Configuration productions
//---------------------------
                 config_def ::= config_literal EOL
                              | 'xml' config_type config_name
                                 BLOCK_OPEN
                                 /* Placeholder for raw XML (not valid yet, see Known Issues) */
                                 BLOCK_CLOSE
                              | config_type config_name BLOCK_OPEN (assignment EOL)* BLOCK_CLOSE;
```

```
              config_literal ::= string_literal_assignment;
                 config_type ::= block_id;
                 config_name ::= block_id;

//---------------------------
// Task definition productions
//---------------------------
                      task_def ::= task_name BLOCK_OPEN
                                   task_sensor task_conf task_primary_cat task_reassignable_cat
                                 BLOCK_CLOSE;

                     task_name ::= block_id;
                    task_sensor ::= 'sensor'                    '=' name_or_string EOL;
                      task_conf ::= 'configuration'             '=' name_or_string EOL;
                 task_primary_cat ::= 'intendedSubmodality'       '=' name_or_string EOL;
             task_reassignable_cat ::= 'reassignableSubmodalities' '=' set_of_strings EOL;

                init_state_def ::= boolean_assignment EOL;

                  workflow_def ::= workflow_statement
                                 | 'if' '(' predicate_factory ')'
                                    BLOCK_OPEN
                                      workflow_statement+
                                    BLOCK_CLOSE
                                      ('else'
                                      BLOCK_OPEN
                                        workflow_statement+
                                      BLOCK_CLOSE)?;
             workflow_statement ::= 'perform' INT NAME with_state? EOL;
                     with_state ::= 'with' set_of_assignments;

               predicate_factory ::= predicate_factory_compoundAnd (OR_OP predicate_factory)?;
    predicate_factory_compoundAnd ::= predicate_factory_atom        (AND_OP predicate_factory_compoundAnd)?;
           predicate_factory_atom ::= predicate_factory_atom_base
                                 | NOT_OP? '(' predicate_factory ')';
      predicate_factory_atom_base ::= pred_boolean
                                 | pred_reserved_single
                                 | pred_reserved_set
                                 | pred_numerical_other
                                 | pred_evaluation
                                 | pred_evaluation_short;

//-------------------------
// Predicates
//-------------------------
                   pred_boolean ::= bool;
              pred_reserved_single ::= reserved_word '==' name_or_string;
                 pred_reserved_set ::= reserved_word 'in' set_of_strings;
              pred_numerical_other ::= NAME MATH_OP number;
                   pred_evaluation ::= NAME '==' (bool | number | name_or_string );
             pred_evaluation_short ::= negatable_name;
                        assignment ::= condition_localization_assignment
                                 | string_literal_assignment
                                 | boolean_assignment
                                 | int_assignment
                                 | variable_assignment;

//-------------------------
// Assignments
//-------------------------
condition_localization_assignment ::= condition_localization_type '=' name_or_string;
                   int_assignment ::= NAME '=' INT;
          string_literal_assignment ::= NAME '=' name_or_string;
                variable_assignment ::= NAME '=' NAME;
                boolean_assignment ::= NAME '=' bool;

                     set_of_names ::= null_term
                                 | BLOCK_OPEN negatable_name (',' negatable_name)* BLOCK_CLOSE;
                   set_of_strings ::= null_term
                                 | BLOCK_OPEN name_or_string (',' name_or_string)* BLOCK_CLOSE;
              set_of_assignments ::= null_term
                                 | BLOCK_OPEN assignment (',' assignment)* BLOCK_CLOSE;
```

```
//-------------------------
// Primitives
//-------------------------

                    reserved_word ::= 'modality' | 'submodality';

                         block_id ::= NAME;
                             bool ::= BOOLEAN;
                           rvalue ::= null_term
                                    | bool
                                    | number
                                    | name_or_string;
                  name_or_string ::= null_term | NAME | STRING;
                           number ::= INT
                                    | FLOAT;
      condition_localization_type ::= ‘trueMessage'
                                    | 'falseMessage'
                                    | 'invalidMessage';
                  negatable_name ::= NOT_OP? NAME;


//-------------------------
// Tokens
//-------------------------

  null_term :    'none';
    BOOLEAN :    'true' | 'false';
        INT :  ('0'..'9')+;
      FLOAT :    INT '.' INT;
    MATH_OP :    '>=' | '<=' | '!=';
RETURN_TYPE :    'boolean' | 'string' | 'int';
       NAME :    ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'-'|'0'..'9')*;
     NOT_OP :    '~';
     AND_OP :    '&&';
      OR_OP :    '||';
 BLOCK_OPEN :    '{';
BLOCK_CLOSE :    '}';
        EOL :    ';';

//-------------------------
// Single-line comment
//
// Single-line comments are two forward slashes, ‘//’
//   followed by zero or more characters that are not a newline (\n) or carriage return (\r), (~('\n'|'\r'))*
//   optionally followed by either: a newline, or a carriage return, or both, ('\n'|'\r'('\n')?)?
//-------------------------
 SL_COMMENT :    '//' (~('\n'|'\r'))* ('\n'|'\r'('\n')?)?;

//-------------------------
// Multi-line comment
//
// Multi-line comments are zero or more characters encapsulated within ‘/*’ and ‘*/’
//-------------------------
 ML_COMMENT :   '/*' .* '*/';

//-------------------------
// Strings
//
// Strings are either:
//  a single quote, followed by one or more non-single-quotes, ended with a single quote, '\'' ( ~'\'' )+ '\''
// or a double quote, followed by one or more non-double-quotes, ended with a double quote, '"' ( ~'"' )+ '"'
//-------------------------
     STRING : '\'' ( ~'\'' )+ '\'' | '"' ( ~'"' )+ '"';
```