Streaming-Archival InkML Conversion

Birendra Keshari and Stephen M. Watt
Dept. of Computer Science
University of Western Ontario
London, Ontario, Canada N6A 5B7
{bkeshari,watt}@csd.uwo.ca

Abstract

Ink Markup Language (InkML) provides a platform–neutral data format that can be used to represent, store and transmit digital ink data. Both streaming and archival applications are supported through different uses of InkML's primitives. While streaming ink and archival ink data can represent the same information, each supports certain operations more directly. Indeed, certain applications can benefit from access to both representations of the same digital ink data. In this paper we present an efficient method to convert archival style InkML to streaming style and vice–versa.

1. Introduction

Multimodal interaction becomes richer and more powerful when it incorporates a pen input modality. With the easy availability of pen–based devices such as tablet PCs, graphics tablets and PDAs, interest in pen–based applications such as online handwriting recognition, mathematical symbol recognition, electronic form filling, ink messaging and authentication, has grown rapidly in the last few years.

For easy and flexible interchange of ink data between packages, it is important to store digital ink in a standard format. The data format should also consider and fulfill the needs of diverse pen–based applications dealing with storage, manipulation and transmissions of digital ink. Jot [3] and UNIPEN [2] were popular formats to represent digital ink before *Ink Markup Language (InkML)* [1]. UNIPEN is very focused on handwriting recognition requirements and is not optimized for data storage or real time data transmission. Jot is a proprietary format and is also not optimized for these purposes. InkML, developed under the ægis of the W3C Multimodal Activity Working Group, is a non–proprietary format whose specification takes into account a wide range of pen–based applications.

InkML is an XML–based language and therefore extensions such as adding application–specific information to ink files are easy. It can provide an accurate representation of digital ink by recording information such as device characteristics, pen tilt, pen pressure and so on. This information can be useful for applications such as handwriting recognition and authentication.

The primitives of InkML allow ink data to be organized in a variety of ways. There are two that are of particular interest: One form, known as *Archival InkML*, allows all declarative information to be given in one place and for ink strokes to be organized and annotated hierarchically. Another form, known as *Streaming InkML*, presents ink strokes in time order and declarative information (brush choice, *etc*) is given in–line, relative to an ambient current state. Streaming style markups are more suitable for applications that deal with transmission of ink. On the other hand, archival style markups are optimized for applications that store digital ink for future purposes because they can recorde more structure. Conversion between these two styles is required by applications that operate in both modes. Here, we present algorithms that provide efficient conversion between these two styles of digital ink.

We discuss streaming and archival ink in Section 2. The translation algorithm is presented in Section 3. Section 4 presents optimization issues to be considered and we Section 5 concludes the paper.

2. Archival versus Streaming

As mentioned earlier, pen–based applications can be broadly categorized into archival and streaming types. Archival applications capture the digital ink from pen devices and store it for future retrieval and processing. In Archival InkML, all the contextual elements, such as brush, trace format and so on, are defined within `<definitions>` elements. Traces and trace groups make direct references to these contextual elements. Such a structure makes it possible to determine the context infor-

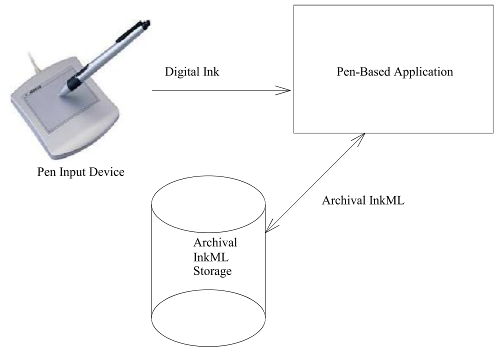


Figure 1. Application using archival ink

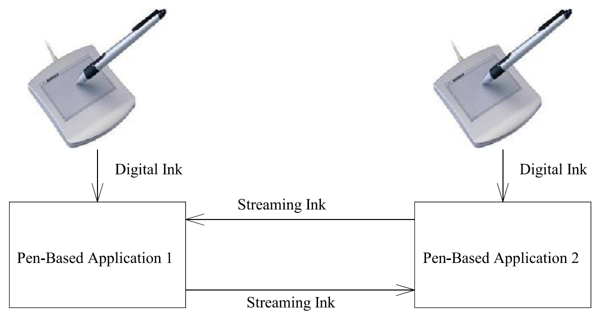


Figure 2. Applications using streaming ink

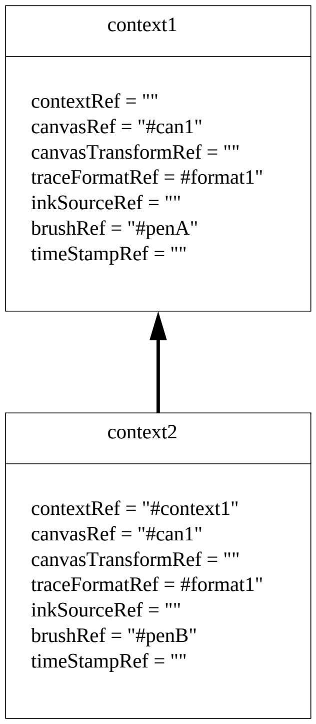


Figure 3. Inheritance and overriding of context through reference

mation directly from the `<definitions>` alone. This directly supports search and retrieval operations. Thus, archival style markups are optimized for retrieval and analysis. An example of an archival application is shown in Figure 1.

Streaming applications capture digital ink and transmit the ink data between application components as it is gathered. Similarly, they can also receive digital ink from other streaming applications. Figure 2 shows an example of a streaming application. In such applications, ink is generated in an incremental order. Changes in the current context are triggered by events such as change of brush. The structure of InkML primitives in the streaming style allows one to capture such events directly. Thus, streaming style markup provides support for the incremental transmission of an ink stream by capturing events.

In streaming style markup, the current context of each trace or trace group depends on the previous context changes and, unlike archival style markup, it can't be determined directly from the `<definitions>` alone. The brush, trace format, ink source, canvas, canvas transform and time are aspects of the current context. In InkML, the current context is updated directly by embedding the contextual elements (*e.g.* `<brush>`, `<traceFormat>` *etc*) within `<context>` or by making references to already defined contextual elements through attributes of `<context>` element (*e.g.* `traceFormatRef`, `inkSourceRef`, `brushRef`) or both. However, a contextual element that has been already referenced by referencing attribute can't appear as a child of `<context>` and vice–versa. Each `<context>` element updates the current context and this can be viewed as overriding contextual elements. For example, in the InkML fragment below,

```
<context xml:id="context1"
         brushRef="#penA"
         traceFormatRef="#format1"
         canvasRef="#can1"/>
...
<context xml:id="context2"
         contextRef="#context1"
         brushRef="#penB"/>
```

since `context2` makes reference to `context1`, all the values of attributes are inherited from `context1`. However, `brushRef="#penB"` overrides the value of the brush attribute (`#penA`). This is similar to inheritance and overriding in object-oriented languages. This is illustrated in Figure 3.

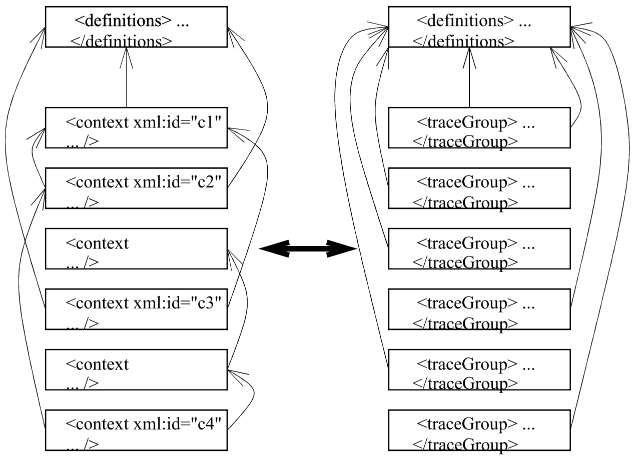
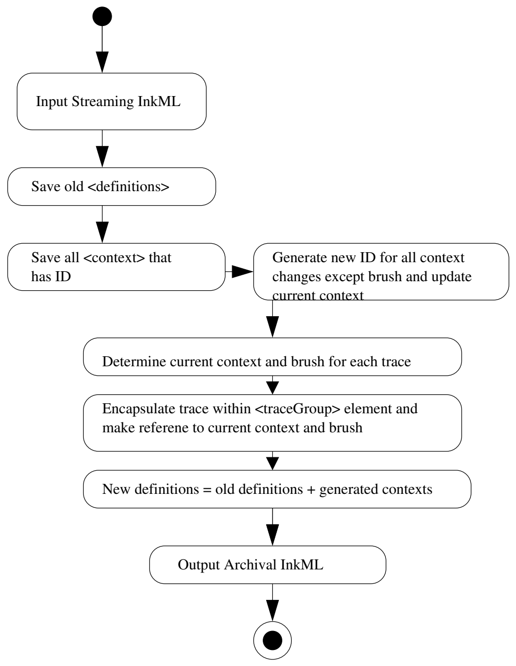


Figure 4. Streaming (left) and archival (right) ink markup structure conversion



Input Streaming InkML
Save old <definitions>
Save all <context> that has ID
Generate new ID for all context changes except brush and update current context
Determine current context and brush for each trace
Encapsulate trace within <traceGroup> element and make referene to current context and brush
New definitions = old definitions + generated contexts
Output Archival InkML

Figure 5. Streaming to archival translation

Although both styles of InkML are equivalent, they impose different requirements on the markup processor and generator. It is desirable to create archival ink from streaming if we want to store the data for future retrieval and manipulation. Streaming ink can be generated from archival if we want to know about the event changes that caused the generation of the ink. For example, such a conversion is required to animate the ink in the way it was written or to transmit the archival ink to other applications. Therefore, translation tools for the conversion between these two equivalent markup styles can be very useful.

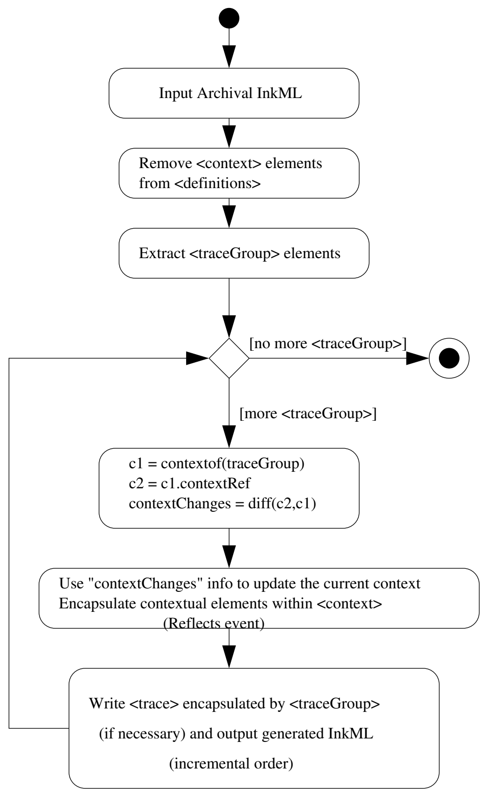
3. The InkML Translation

Translation of InkML from one markup style to another requires analysis of one structure and generation of another, equivalent structure optimized for a different use. One can view the translation problem as the re–arrangement and update of InkML primitives while preserving semantics. This is summarized in Figure 4. As we can see from the figure, in Archival InkML the current state can be known directly from the contextual elements that appear as children of the `<definition>` element. But the determination of the current state in streaming style may not be straightforward because of the possibility of chains of multiple references. The following sections discuss two translation algorithms in detail.

3.1. Streaming to Archival Translation

The central idea in Streaming to Archival InkML conversion is to determine the context for each trace, put it inside a `<definitions>` element and encapsulate traces with a `<traceGroup>` element that makes reference to the appropriate `<context>` element.

Figure 5 shows an activity diagram for streaming to archival translation. The Streaming InkML is parsed using an XML parser. The existing elements inside `<definitions>` are saved for future use. New IDs of the form "contextg$N$", where N = 1, 2,..., are generated for the contexts that don't have IDs. The state of current context is maintained in a data structure with fields for the id, brush, trace format, canvas, canvas transform, inkSource and timestamp. Whenever a change in context occurs (through the `<context>` element), the state variables of the current context are updated. Traces that share the same context are put within a shared `<traceGroup>` element which makes reference to an explicit context with the current state. Brushes are directly referenced by the `<traceGroup>` through the `brushRef` attribute. All traces are treated uniformly. The new `<definitions>` to be put in the Archival InkML are given by the union of the old `<definitions>` and the newly generated `<context>` elements.

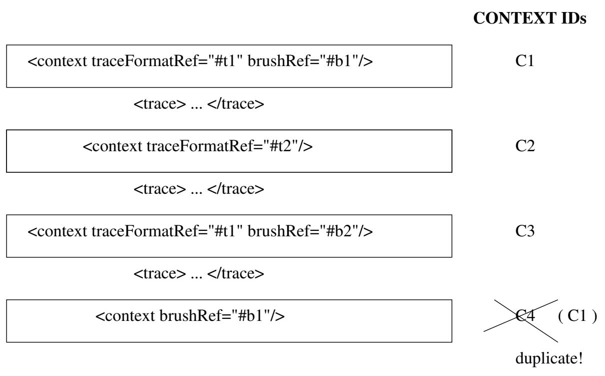


Input Archival InkML
Remove <context> elements from <definitions>
Extract <traceGroup> elements
[no more <traceGroup>]
[more <traceGroup>]
c1 = contextof(traceGroup)
c2 = c1.contextRef
contextChanges = diff(c2,c1)
Use "contextChanges" info to update the current context
Encapsulate contextual elements within <context> (Reflects event)
Write <trace> encapsulated by <traceGroup> (if necessary) and output generated InkML (incremental order)

Figure 6. Archival to streaming translation

3.2. Archival to Streaming Translation

The main idea of archival to streaming translation is to make explicit incremental context changes as an archival ink collection is traversed. This translation captures the changes in context and presents them inline. To do this, all the `<context>` definitions are removed from `<definitions>` and saved temporarily. Each `<traceGroup>` element is then processed in sequence. Let $c1$ be the context referenced by `<traceGroup>` and $c2$ be the context referenced by $c1$. Now, the change in context δc is $\mathit{diff}(c1, c2)$. Examples of such changes are a change in brush or a change in trace format. These changes are reflected in the InkML by contextual elements (*e.g.* brush, trace format) encapulated in `<context>` elements appearing in–line with the `<trace>` and `<traceGroup>` elements. Traces with annotations (`<annotation>` or `<annotationXML>`) are encapsulated within `<traceGroup>`s to contain the annotatiosn in the output streaming ink. In the absence of annotations, `<traceGroup>`s are not used. A stream of digital ink is output after each `<traceGroup>` is processed. Thus, Streaming InkML is generated iteratively and the digital ink generated in each iteration shares the same context. Figure 6 illustrates this process.



CONTEXT IDs
<context traceFormatRef="#t1" brushRef="#b1"/>
C1
<trace> ... </trace>
<context traceFormatRef="#t2"/>
C2
<trace> ... </trace>
<context traceFormatRef="#t1" brushRef="#b2"/>
C3
<trace> ... </trace>
<context brushRef="#b1"/>
C4 (C1)
duplicate!

Figure 7. An example of duplicate context

4. Optimization Issues

In Streaming InkML, the context at one point in the stream might reproduce the context of a point in the stream. Naïvely, this would cause the generation of redundant contexts in Streaming to Archival InkML translation. To reduce the size of the generated Archival InkML, it is necessary to avoid such redundant contexts.

An example of duplicate contexts being generated is shown in Figure 7. The context of each trace in Streaming InkML is assigned an ID during the conversion process (section 3.1). The contexts *C1* and *C4* happen to be the same (both have ‘1’ as traceformat and ‘b1’ as brush).

One situation where contexts would normally be often repeated is in collaborative inking. If digital ink is collected from different ink sources in a single InkML stream, then the ink source might send contextual elements, even if its current state is unchanged, in order to refresh the current state at the receiver side.

Duplicate contexts can be removed from Archival InkML by the process shown in Figure 8. All contexts appearing in the input archival ink are analyzed. Each context is modeled by an object that has all of the attributes of a `<context>` element as well as a pointer to an “original” context, to be used in case it is a duplicate. As the contexts are processed, the actual values of the attributes are determined by resolving the references and inheritence. The set of contexts is then partitioned into equivalence classes using the relation that contexts are equivalent if correspoinding attributes, excluing “id” and “original”, have equal values. Duplicate contexts are removed by choosing one representative from each equivalence class as the original and replacing all references to members of an equivalence class with references to the original.

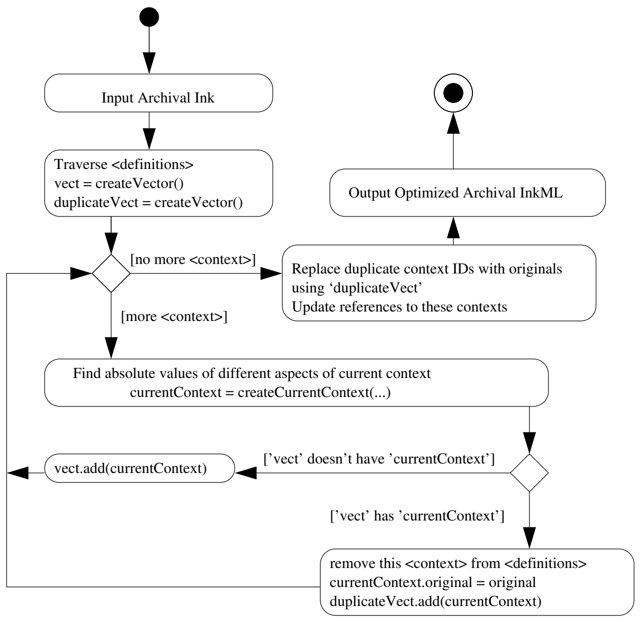

Input Archival Ink
Traverse <definitions>
vect = createVector()
duplicateVect = createVector()
[no more <context>]
[more <context>]
Replace duplicate context IDs with originals
using 'duplicateVect'
Update references to these contexts
Output Optimized Archival InkML
Find absolute values of different aspects of current context
currentContext = createCurrentContext(...)
['vect' doesn't have 'currentContext']
vect.add(currentContext)
['vect' has 'currentContext']
remove this <context> from <definitions>
currentContext.original = original
duplicateVect.add(currentContext)

Figure 8. Optimization of Archival InkML

5. Conclusions and Future Work

We have presented methods to translate streaming style InkML to archival style and *vice versa*. We have also shown an optimization to reduce the size of the Archival InkML. We believe that such translators will be useful components for pen–based applications that operate in both archival and streaming mode. Our future work includes using these translators in a collaborative inking environment to store ink conversation for further processing and to recover serialized ink from processed data.

References

[1] Y.-M. Chee, M. Froumentin, and S. M. Watt (editors). *Ink Markup Language (InkML)*, October 2006. `http://www.w3.org/TR/InkML/`.
[2] I. Guyon. Unipen 1.0 Format Definition. The Unipen Consortium, 1994.
[3] Slate corporation. JOT — A Specification for an Ink Storage and Interchange Format, 1993.
[4] X. Wu. Achieving interoperability of pen computing with heterogeneous devices and digital ink formats. Master's thesis, The University of Western Ontario, London, ON, Canada, December 2004.