SCML: A Structural Representation for Chinese Characters

Senior Honors Thesis

Daniel G. Peebles
Advisor: Devin Balkcom

May 29, 2007

Dartmouth College Technical Report
TR2007-592

Abstract

Chinese characters are used daily by well over a billion people. They constitute the main writing system of China and Taiwan, form a major part of written Japanese, and are also used in South Korea. Anything more than a cursory glance at these characters will reveal a high degree of structure to them, but computing systems do not currently have a means to operate on this structure. Existing character databases and dictionaries treat them as numerical code points, and associate with them additional 'hand-computed' data, such as stroke count, stroke order, and other information to aid in specific searches. Searching by a character's 'shape' is effectively impossible in these systems.

I propose a new approach to representing these characters, through an XML-based language called SCML. This language, by encoding an abstract form of a character, allows the direct retrieval of important information such as stroke count and stroke order, and permits useful but previously impossible automated analysis of characters. In addition, the system allows the design of a view that takes abstract SCML representations as character models and outputs glyphs based on an aesthetic, facilitating the creation of 'meta-fonts' for Chinese characters. Finally, through the creation of a specialized database, SCML allows for efficient structural character queries to be performed against the body of inserted characters, thus allowing people to search by the most obvious of a character's characteristics: its shape.

Contents

List of Figures

1 Introduction

1.1 Basic Terminology

An important distinction that will be used throughout the paper is that between the terms *character* and *glyph*. The former is defined as 'the smallest component of written language that has semantic value,'[1] and as such is a purely abstract concept. A glyph is instead a concrete visual representation of a character. These definitions are consistent with Unicode's terminology.

The terms *Chinese character* and *character* will be used throughout the paper to mean sinographs, or characters from the Chinese system. They do not necessarily mean the characters currently used in China or even characters originating from there, as many characters were coined elsewhere.

1.2 Background

The Chinese character system as we know it today has existed for well over 3000 years[1]. As well as being the primary writing system for Chinese, it is very prominent in written Japanese, used in South Korea, and was used historically to write Vietnamese. Because of this, it is often called the CJK or CJKV script. It has also been used as a foundation or inspiration for several other writing systems, such as the Japanese Kana syllabaries, the now-extinct scripts Khitan and Jurchen, and the Nakhi script Geba.

The system is sometimes called an ideographic script[2], but this is not technically correct as only a small percentage of the characters are actually ideographs. More accurately, the system is called a logographic one, in which each symbol represents not necessarily an idea but a morpheme. The characters are sometimes called sinographs to show the connection to China without explicitly referring to a specific country's system, as there are many variations between the national writing systems.

It is significantly more complex than most phonetic systems (as can be seen in figure 1) due to its high number of primitives and the effectively infinite possibilities for combining them. Despite this, over 1.3 billion people worldwide are literate in the characters, which means knowing at least two or three thousand characters, depending on the country. There is no definitive authority on the full body of characters, but the Japanese *Dai Kan-Wa Jiten* (大漢和辞典) dictionary has entries for over fifty thousand of them, and the Chinese *Zhonghua Zihai* (中華字海) has entries for a staggering 85,568 characters. The Unicode standard currently supports over seventy thousand of them, and is considering a proposal for an additional twenty-six thousand, while the Japanese Mojikyo Institute (文字鏡研究会) has catalogued over one-hundred thousand of them for computing. These numbers are impressive, but most of the characters beyond the first ten or so thousand are historical variant forms of other, more common characters, and are extremely rare.

Figure 1: Examples of Chinese characters.

Even ignoring the huge number of rare characters, several thousand remain relatively common.

[1] Recent studies[2] on carvings at Damaidi (大麦地) in northwestern China suggest that proto-Chinese characters may have existed as early as 8,000 years ago.

[2] The Unicode Consortium, for example, uses this nomenclature.

These can still be extremely complex, with dozens of strokes and elaborate structures. To facilitate character lookup, the radical system was created, which designated one primary part of each character as its radical, and indexed characters by this radical and the number of strokes required to write the rest of the character. In this system, to look up a character, one must follow these steps:

1. Identify the component in the character which is probably its radical.
2. Count the strokes in the chosen component, to find it in the radical index.
3. Count the residual strokes in the character to see where in the list of characters it lies.
4. Find the appropriate entry, or repeat steps 1-3 with a different component of the character.

This process is better than perusing a huge unordered table of characters, but it is cumbersome even for people who are fluent in the characters, and for learners it is often prohibitively difficult. There is a set of rules prescribing the order in which radicals are chosen, but these are sometimes ambiguous, unpredictable or inconsistent. Some electronic dictionaries improve the situation by allowing one to search by not only the official radical, but also by other recognizable parts of the character, or by allowing the restriction of the search by number of total strokes. They all have one fundamental shortcoming, though: they associate human-generated data with each code point. That is, a human will count the strokes of a character, and associate this computed value to the character's code point in the database. Similarly, a human will decide which component in the character is the radical, and put that in the database. The computer effectively knows nothing about a code point other than the information provided by the human, and nothing prevents the information from being self-contradictory. Furthermore, with a database of tens of thousands of characters, only a human reviewer can prevent duplicate characters from being added, despite the fact that computers are remarkably well-suited, in theory, for this kind of operation.

These issues are evidenced in the IRG (Ideographic Rapporteur Group) proposals for the addition of new characters to the Unicode standard, in which submissions of rare characters are often rejected or corrected with comments along the lines of 'radical correct? maybe U+2FA4?' or 'stroke count wrong,' or are simply marked as duplicates of already-present characters[3]. The submissions consist of rough, often pixelated or hand-drawn glyphs, along with the assigned radical, computed stroke count, and information on sources.

Countless other applications exist in which computers are used with Chinese characters, but in all of them, they are effectively used only to store and access human-generated information: flat (i.e., pure bitmap data) stroke-order diagrams are meticulously created by dedicated volunteers[3], structural IMEs (Input Method Editors) contain their own specific information on character structure, and CJK fonts are created by people looking at reference glyphs provided by standards organizations and using their own knowledge and sense of aesthetics to create new glyphs. What these all have in common is that people repeatedly interpret Chinese characters, extract structural data based on their knowledge and criteria, and then retain the data in a form unusable by anything but their specific application.

1.3 The Idea

It is clear that the CJK computing world would benefit from a system giving computers a complete structural knowledge of characters. Such a system would aid electronic dictionaries, standards orga-

[3]See, for example, http://www.kanjicafe.com/ or http://commons.wikimedia.org/wiki/Category:CJK_stroke_order.

nizations[4], learners, font producers, typists, and many others.

The purpose of SCML (Structural Character Modeling Language) is exactly this. Its aim is to be a language describing a data structure that uniquely represents a character, allowing algorithms to process the structure in order to provide features necessary to the applications listed above and more:

- A language that describes the pure character without aesthetic considerations. This would aid standards organizations such as Unicode by providing an abstract character structure that can be processed to find stroke count and order, thus minimizing redundant and error-prone information.
- A hierarchical structure that can be decomposed into smaller and simpler components to aid learners and linguists researching the characters.
- A database that allows efficient searches of arbitrary character structures and prevents, by its design, the insertion of duplicates. It can also allow for topological sorting of character graphs, giving an ordered derivation of characters by their components, which can be of use to learners and educators.
- A concept of views that allows designers to create CJK meta-fonts by codifying their sense of aesthetics and using it to generate glyphs from abstract character structures. These could also be integrated with the included stroke-order information to either annotate character glyphs with stroke order or create animated stroke-order diagrams.
- A system that provides consistent structural information for potential use by appropriate IMEs.

One of its main advantages is that the language aggregates previously separate information into a single definition. A character dictionary will have, for every entry, a glyph, stroke count, radical, other significant components, and possibly a stroke order. An SCML definition encompasses all of the above, providing more flexibility and eliminating issues of self-consistency between data associated with a character.

SCML was specifically designed to be a *character* description language, using the word 'character' in the same way Unicode defines it, without any glyph-like characteristics.

1.4 Approach

My approach to the problem was to take the Chinese character system and divorce it completely from any linguistic associations or background. I start with the assumption that character etymologies, national variants, pronunciations, and the like, are irrelevant to the character system itself. Structures can later be annotated with metadata that cover all of the above, without losing generality. The method is based on the observation that all characters are composed of rough hierarchies of subcomponents, which all ultimately consist of arrangements of primitive strokes. To construct characters, I provide around 40 stroke primitives which can then be composed with other basic constructs, and which together allow for the generation of arbitrarily complex characters. This approach effectively leads to a *character graph* that uniquely defines the character's abstract structure.

[4]Unicode, for example, has struggled[4] to apply the same standards to the CJK characters as it does elsewhere: other scripts, such as Hangul and all Latin-derived scripts have been successfully decomposed into combining characters, where possible.

The constructs were selected after the extensive examination of thousands of characters to see what would be needed to represent them simply and unambiguously. The objective was to be able to describe as many characters as possible, but to also stay as simple and concise as possible. Because of this, some outliers (see figure 2) that are present in the Unicode specification do not currently have a means of encoding in SCML. It would not be impossible to add one, but I was reluctant to add features that would only be used to describe one or two characters. I therefore consider these outliers to be outside the character system I define, and do not attempt to deal with them. Luckily, there appear to be barely more than a dozen of these unrepresentable characters, so their existence does not present a significant completeness issue.

(a) Cursive last stroke (b) Upside-down 予 (c) Rhombi (d) Double border

Figure 2: Examples of 'strange' characters that SCML cannot currently represent.

Also not covered by SCML are inter-character size relations. The only difference between Unicode code points U+53E3 (口) and U+56D7 (囗) is that the latter is larger, and is used as an enclosure, whereas the former is used within characters. This is a historical difference that arises from the *Kangxi* radicals[5], and the distinction in individual characters can still be determined by algorithmic means.

A fundamental aspect of my approach was to omit any kind of numerical measurements in the character definition, to avoid associating the Form (as Plato might have called it) with any particular visual representation thereof (see figure 3 for an example.) On the other hand, the other primary objective was to maintain enough information to allow the reconstruction of a meaningful glyph from the abstract representation, to create the basis for a Chinese character meta-font.

Figure 3: Different glyph styles for the same abstract character.

I thus define a character to be a unique sequence of strokes and compositions thereof using the constructs provided. This definition takes into account the sequence of strokes, so two characters that 'look the same' but have different underlying stroke orders are considered distinct characters in the system. The association of these alternate stroke orders as being semantically equivalent is left to higher-level metadata.

Unlike other character description languages, SCML creates a means to fully describe, analyze, and search characters. The search requirement was a significant guideline in the design of the representation, as for obvious reasons I wanted to avoid linear searches over all known characters unless strictly necessary. The language and the database were therefore designed to take advantage of the high degree of component reuse in chinese characters.

1.5 Thesis Overview

I have already demonstrated the applications for SCML and the approach to its design. In section 2, I describe related work in the field of Chinese character description languages, and in the associated character processing fields. In section 3, I cover the ideas behind the language, the structure it represents, and an alternate compact language. In section 4, I present some practical approaches to gathering useful information from abstract character structures. Section 5 discusses the design of a structural database of characters. Finally, in section 6, I examine the possibility of automating the creation of a body of SCML graphs using existing outline fonts.

2 Related Work

The idea that Chinese characters can be decomposed into smaller units is by no means a novel one. In everyday spoken Chinese and Japanese, it is as common to verbally describe characters by their components as it is in western societies to spell out unfamiliar words. The idea is at the basis of the *Kangxi* radical system, the more recent Spahn & Hadamitzky simplified radical system[6], Heisig's Kanji learning system[7], and countless others. In computing, there are also several prior examples of character description languages. These, however, are geared toward specific purposes and don't represent true abstract characters. This section will cover the principal character description languages and briefly examine previous work in the field of character segmentation related to section 6.1.

2.1 The Wenlin Institute's CDL

The Wenlin Institute's Character Description Language project is an ambitious effort to define all the characters currently defined in Unicode (they currently have over 56,000[8]) in terms of an XML language that places 39 types of strokes[9] individually. In describing the concept, Wenlin says that 'CDL descriptions lie somewhere between abstract character and concrete glyph.' This is because their language, while breaking a character up into its constituent strokes and maintaining their order, still describes these strokes in terms of an absolute coordinate system. The implementation currently defines a 128×128 canvas[10] upon which strokes are placed, with explicitly defined control points and bounding boxes. The data structure this language represents is essentially a list, but the system also allows one to include the contents of previously defined characters, allowing for optional hierarchy.

CDL therefore contains enough information to reconstruct glyphs fully. It also contains the information to generate self-consistent stroke counts, and permits the creation stroke-order diagrams. It is, however, not well suited to character analysis and search due to its reliance on explicit coordinate positioning. It is non-trivial to determine if two lists of the same strokes with different control points are 'the same character,' or to even come up with a moderately flexible definition of character identity. Because of this, it would be difficult to use a CDL description as a search key in a database, without a specific and practical definition of character equivalence (one more practical than exact coordinate equality.)

Also, because the data structure it describes is essentially flat, it is extremely difficult to extract structure from a CDL description. This would make it extremely difficult to perform, for example, a useful substructure search (i.e., 'find all characters that contain 木',) or character analysis operations such as stroke order inference, which rely on rules defined in terms of high-level concepts such as 'side-by-side' or 'inside/outside.'

2.2 HanGlyph and the Unicode IDS

HanGlyph, despite its name, specifies *characters*, by the definition used in this paper. It is similar to SCML in that it constructs a character structure by composing stroke primitives. These primitives are abstract entities and do not have prescribed visual representations. Characters are specified with postfix expressions that apply composition operators (of which there are five) to strokes and other composed entities.

This language, although not directly related to SCML, covers much of the same domain as SCML, but there are some important differences:

- From the documents available[11, 12] (there is no publicly available implementation) on HanGlyph, the language appears to lack some specificity. The authors provide[12] as an example the character 土, whose HanGlyph expression is `h h=< s+_`. This expression reads, 'combine two `h` strokes with a top-bottom operator such that the upper stroke is shorter than the lower stroke (`h h=<`,) and then cross with an `s` stroke and align crossing to bottom (`s+_`.)' The first thing to note here is that no information on stroke order is maintained, and none can be maintained without adding explicit order indicators. Another issue is that the mechanism for intersecting operands isn't as flexible as some characters and stroke types might require. For example, it is unclear how one would unambiguously construct characters with complex intersection behavior, such as 丐 or 為 using the provided mechanisms.
- Another issue is that of too much specificity. Numerical measurements are still present and possible with HanGlyph. These are used in the scale operator, which allows a character designer to explicitly state the ratio of the sizes of its two operands, and in the intersect operator, in which one can specify the distance the operands are shifted in order to achieve the desired number of intersections[12]. The presence of these measurements brings the language closer to glyph (visual) descriptions, as they allow for more specificity in the reconstruction of glyphs, but make the comparison of similar character expressions more difficult.

The Unicode Ideographic Description Sequences[13] follow the same idea used in HanGlyph, except using prefix notation instead of postfix. They rely on twelve Ideographic Description Characters that correspond roughly to the HanGlyph operators. These, however, are used to combine higher-level primitives than strokes, as they lack the flexibility to describe anything more than simple stroke intersections.

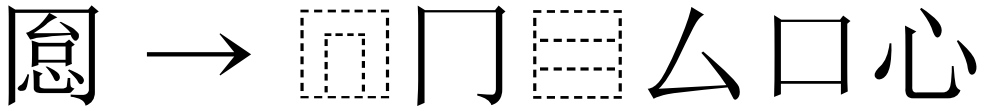

Figure 4: A Unicode Ideographic Description Sequence.

The purpose of these IDS is different from any of the other character description languages, in that Unicode explicitly explains that they are only to be used to describe characters not already present in the standard. Due to ambiguity that can arise from an IDS, Unicode does not maintain a database of these for the characters it encodes. Furthermore, there is no requirement to interpret character expressions to generate glyphs, so their purpose is ultimately to allow a reader to "create a mental picture of the ideographs from the description"[13, p. 307], and the idea has seemingly not been developed far beyond that.

2.3 Stroke Decomposition

Much research[14, 15, 16, 17] exists in the field of Chinese character stroke decomposition, but none of it covers the rather specific problem of decomposing vector sources, instead dealing with much more common (and more useful, outside of this domain) raster sources. It might be possible to apply some of these techniques to outline sources by rasterizing them first, but this would discard useful data that could otherwise be used to segment the glyph more accurately.

3 Structural Character Modeling

There are many approaches to representing what might be seen as a Chinese character:

- **A bitmap:** simple two-dimensional array of pixels that contains an image.
- **A vector drawing:** a set of linear, quadratic, and cubic segments that define a region.
- **A CDL character:** a list of specific stroke types with bounding boxes and control points to place them on the drawing canvas.
- **A HanGlyph character:** an expression using basic operators to combine different stroke types into a larger character.

Using the Unicode character/glyph paradigm, of the above, the bitmap and vector methods describe pure glyphs, whereas CDL sits somewhere between a character and a glyph, prescribing exact coordinate locations for its strokes, but maintaining the stroke order and decomposition of the character. Finally, HanGlyph appears to describe almost 'pure' characters, but maintains some glyph-like operations that prescribe relative dimensions.

SCML instead describes pure characters. It attempts to store a character in much the same way as the average human mind might remember and describe it[5]. Most people would not remember a character in terms of pixels, bézier curves, or even stroke bounding boxes or control points. Nor would most people think of composing two strokes in such a manner that one is 1.1 times longer than the other. People tend to remember characters in a more abstract manner, and then apply their sense of aesthetics to the abstract form in order to write out the glyph.

3.1 Structural Modeling Concepts

This section covers the basic concepts related to SCML's approach to structural character modeling, as well as discussing aspects of the XML-based language.

3.1.1 Definition of Structure

SCML uses the word 'structure' in the context of Chinese characters in a manner consistent with the general-purpose definition of the word, but with some specific additional points:

- Relationships between strokes in characters are considered in a relative sense, to allow for high-level concepts such as one part of a character being emphinside another, or beside it.

[5]Note that these statements on human memory are not based on academic studies, but rather on informal discussions with users of Chinese characters and my personal experience with them.

- These relationships form rough hierarchies of character parts.
- Stroke order is a considered an essential part of a character's structure.

3.1.2 SCML Constructs

SCML provides four basic constructs to allow the construction of character structures:

- **Strokes**, the most basic primitives.
- **Anchors**, denoting intersections between strokes (and in some cases, between strokes and locations locations.)
- **Layouts**, to constrain the positioning of strokes and other layouts within the character.
- **Locations**, to allow placement of strokes and layouts in regions defined by stroke intersections.

In addition to the above, it provides named positions on strokes upon which to place anchors[6], and named regions around strokes to define locations. Finally, for the sake of convention, those constructs that can be placed in a layout are called entities, and maximal sets of intersecting strokes are called components (see section 3.1.3 below.)

Layouts are of three basic types, *row*, *column*, and *free*, and arrange their children into logical structures according to those descriptions: row and column layouts specify ordering constraints that link the order of their children with their logical placement, whereas free layouts express no constraints of their own, leaving all constraints coming from children within. In the rare cases where it is needed, relative size constraints can be placed on the children of the row and column layouts (e.g., 'row with larger upper children' to allow structural distinctions between the distinct characters in figure 5, for example.)

(a) Shorter upper stroke. (b) Shorter lower stroke.

Figure 5: Two distinct characters that need explicit size relations in order to be distinguished.

Anchors exist primarily to allow points on multiple strokes to correspond. The same logical anchor on two separate strokes specifies that those strokes must intersect. The anchor lies on specific named positions on the individual strokes to allow for flexible definitions. For example, in figure 6(a), the vertical hooked stroke has an anchor defined on its `begin` position, and the horizontal stroke has the same anchor defined on its `inside` position. Anchor semantics then require that those two points coincide, and thus that the strokes intersect. Named positions can be of two types, *singular* and *plural*. The former typically lie at the extremities and corners of stroke segments, where having multiple anchors wouldn't have a purpose, whereas the latter fall on the inner lengths of stroke segments, and tend to be approximately evenly spaced across the segment's length (although this will depend on the renderer's layout strategy, see 4.)

[6]See A for a complete list of strokes and data defined for them.

Locations are defined in a manner similar to anchors. A location is a region of the plane defined by the intersection of regions around strokes. Simple one-segment strokes will usually define only two regions, but more complex strokes will usually have more. In figure 6(b), the horizontal stroke's `below` region is being intersected with the vertical hooked stroke's `left` region, to define the location for the tick stroke. Note also that the intersection of the two strokes was defined as above with an anchor, but using the `inside` positions on both strokes.

(a) An anchor used to intersect two strokes (b) A location defined by two stroke regions

Figure 6: Examples of anchors and locations.

It is also possible to define an anchor inside a location, to allow strokes and components to be only partially within the bounds of a location, as is necessary for characters like 贞.

The four basic constructs, when used to define a character, nearly form a tree. That is, they form a tree if one ignores the presence of anchors and locations. Thus, with the latter two constructs taken into account, a character is a directed acyclic graph. Edges represent parenthood, so a layout that contains two strokes will have two edges leading into the strokes. They also represent ownership of anchors, so two strokes that are connected by an anchor will have edges (labeled with the anchor position in the stroke) going into the anchor. Locations are represented in a similar manner to anchors, but with an added outgoing edge to the entity contained inside the location.

A Note on Notation: Throughout the rest of the paper, the term *character tree* is used when ignoring anchors and locations, and *character graph* is used when these are relevant to discussion.

A detail to note is that the children of a layout form a list, and not a set. The ordering of entities within this list is defined to be the stroke order of the character. That is, a depth-first traversal of the character tree should produce the strokes in their natural stroke order. This rests on the assumption that components are well-ordered when compared by the order in which their constituent strokes are written: if A and B are components such that $\text{order}(A) < \text{order}(B)$, then $\forall\, a \in A, \forall\, b \in B, \text{order}(a) < \text{order}(b)$. This restriction appears to fit with traditional stroke ordering rules and with the official stroke orders, and no characters seem to violate it.

3.1.3 Components

An important concept in SCML that does not have a language construct associated with it is that of components. These are, like the connected components of graph theory, defined as maximal sets of intersecting strokes. Components are useful in that they allow characters to be decomposed into simpler structures using only layouts (see figure 7 for examples) and locations, avoiding the complications of anchors.

Components can contain instances of *Fine-grained substructure*, which is defined as normal structural elements such as layouts and locations occurring within a component as a result of stroke intersections.

(a) 2 Components: 宀, 子 (b) 2 Components: 几, 丶 (c) 4 Components: 口, 乂, 丶, 丶

Figure 7: Examples of components in characters.

The algorithm to find components in a character graph is equivalent to that of generic graph theory. One must simply select all the stroke and anchor nodes from a character graph, maintaining undirected edges between anchors and strokes that were connected in the original graph. It is then sufficient to run one of the many available connected component algorithms on the resulting undirected subgraph.

The concept of components is used in section 5.1 to construct a database for SCML characters, and can be used to map SCML to other character description languages, such as Unicode's IDS.

3.1.4 Ambiguity

One issue that needs to be examined is that of ambiguity of definitions. Due to the specificity of an SCML definition, it is impossible for one definition to represent multiple characters, but it may be possible to represent a character with more than one definition. For example, in the absence of other factors, the character shown in figure 8 can be represented as a row layout containing two column layouts, or as a column layout containing two row layouts.

Figure 8: An ambiguous character?

However, as the stroke order of the character depends on which of the above structures is chosen, there is no room for ambiguity. As stroke order rules go first left-to-right and then top-to-bottom, the only possible structure for this character is a row layout containing two column layouts.

It is thus possible in some cases to define what might be called the same character, using different structures in SCML. This, however, would lead to different stroke orders, and thus create separate characters, by the SCML definition of character (see 1.4.) Although it doesn't coincide with the common definition, given the existence of distinct official stroke orders and SCML's intent to maintain these, this definition is convenient.

3.2 XML-based Language

The actual SCML language is based on XML, due to the ubiquity of processing tools and libraries for it. It may eventually be useful to create a 'compact language' similar to that provided by RELAX NG[18] that can be mapped in a one-to-one fashion with the more verbose XML. However, as the current database implementation (section 5.1) does not store the raw XML, verbosity does not present an issue for space consumption, and a compact language would thus be only for aesthetic and practical purposes.

The tree paradigm is very easily reflected in XML syntax, so SCML's tree-like constructs (layouts) were easily expressed in XML. Anchors and locations, however, are less straightforward. They are currently implemented by giving each anchor or location a unique id, and defining multiple occurrences of the same id to be the same node. There is no restriction on the type of the id, as it is just compared for equality against other instances.

On a structural level, all SCML definitions follow the schema defined in appendix B. It is relatively compact and relies on almost no attributes or text to define character structures, but one detail to note is that there are two different kinds of location elements, called `Location` and `StrokeLocation` in the schema. The difference is that the latter defines where the location will be, and the former specifies the entity that is to be contained within the location. This is represented in the character graph by having the `StrokeLocation`s as inbound edges to the location node, and the contents of the `Location` element connected by the one outbound edge of the node (see figures 10 and 11 for an example of this.)

Also one must note that the order anchors appear within a stroke is important, *when those anchors are placed in the same named position*. This is applicable in the case of positions that are plural on a stroke (see section 3.1.2) upon which multiple anchor points can be defined. The order in which the anchors are placed within the stroke defines the order the anchors' associated points will be allocated on the stroke. For singular positions, the order does not matter.

The following example demonstrates the SCML language, and how it relates to the character and its associated graph[7].

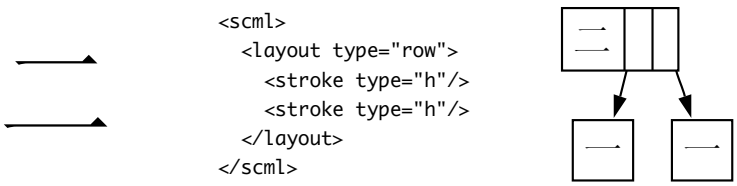

Figure 9: A character, its corresponding SCML definition, and its character graph.

It should be noted how in the above XML there is no explicit 'new row' indicator in the row layout: each sub-element gets assigned a row in the order it appears in the XML. This affects the logical placement of the stroke and the implied stroke order of the character. Also, in the sample glyph, it is obvious that the upper stroke is slightly shorter than the lower one. This is a common aesthetic choice exhibited in almost all CJK fonts, and is thus the default behavior for the row layout, although there are layout variations that specify neutral or opposite proportions in the case that these are required by the structure and not aesthetics. Because of this, there is no explicit statement in the structure that the upper stroke should be shorter.

The SCML definition in figure 10 illustrates a more complicated example for the character 存 that employs the other two constructs (anchors and locations) as well as free layouts.

Here, one can see that the first three strokes contain anchors. Their `at` attributes specify where on the parent stroke the anchor should lie, and the `id` attributes determine anchor coincidence. The strokes also contain location specifiers. In this case, these constrain location 0 to be below the first and second strokes, and to the right of the third stroke. Finally, the contents of that location are specified

[7]The graphs in figures 9 and 11 use rectangular nodes to represent layouts, square nodes for strokes, circular nodes for anchors, and triangular nodes for locations.

```
<scml>
  <layout type="free">
    <stroke type="h"> <anchor id="0" at="inside"/>
      <location id="0" at="below"/>
    </stroke>
    <stroke type="pb"> <anchor id="0" at="inside"/> <anchor id="1" at="inside"/>
      <location id="0" at="below"/>
    </stroke>
    <stroke type="s"> <anchor id="1" at="begin"/>
      <location id="0" at="right"/>
    </stroke>
    <location id="0">
      <layout type="free">
        <stroke type="hg"> <anchor id="2" at="hook-tip"/> </stroke>
        <stroke type="wg"> <anchor id="2" at="begin"/> <anchor id="3" at="inside"/> </stroke>
        <stroke type="h"> <anchor id="3" at="inside"/> </stroke>
      </layout>
    </location>
  </layout>
</scml>
```

Figure 10: A complex SCML definition that uses all the constructs.

with another free layout that describes the 子 character. This description corresponds to the graph in figure 11, which is an almost direct mapping from the XML. The one peculiarity of the graph is the order-labeled edges ('0:inside' and '1:inside') on the second stroke in the character. This is used to maintain the order of the two anchors defined on the plural named position on that stroke, as described earlier in this section.

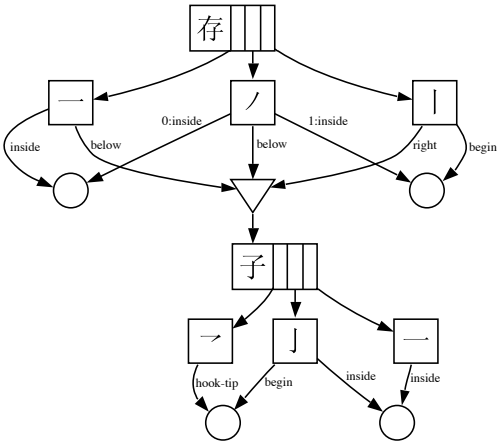

Figure 11: The character graph for figure 10.

4 Views

As an abstract model, an SCML structure describes a character, but how that character is presented is left entirely up to views. There are various types of view one could create on top of the model. One

could, for example, simply display the character structure as a directed graph, or one could extract useful information from the model such as stroke count or order, create character graph definition files for Graphviz such as those used to create figures 9 and 11, or maybe one could infer which radical the character would be classified under in the traditional system. Then, of course, there are the visual views, that display character structures in a visual manner, but even these can take different forms: they could be used to statically render glyphs to screen, might direct a robotic arm in robotic calligraphy, or could be rendered into an animation illustrating stroke order.

The following sections will illustrate possible implementations of some typical views on SCML structures, but these are by no means the only ones. The simple algorithms in sections 4.1 and 4.2 have been fully implemented, but due to the many different types of strokes and rules associated with them, not all stroke types are yet supported by the implementation of the system discussed in sections 4.3.1 and 4.3.2. There is, however, no fundamental difference between different stroke type rules, so adding all the stroke types to the current implementation is just an issue of time.

4.1 Stroke Count

Determining stroke count for a given SCML structure is almost trivial. It is simply the number of stroke nodes in the character graph. This can be determined with a simple traversal of the character tree, in the following manner:

Algorithm 1: `stroke_count()`

Input: A character structure, C
Output: The number of strokes in the character

$n \leftarrow 0$;
$S \leftarrow$ new Stack;
S.push(C.root);
while *not* S*.empty* **do**
  $s \leftarrow S$.pop;
  foreach $c \in s$*.children* **do**
    if c *is a stroke* **then** `/* we have a stroke, so increase the count */`
      $n \leftarrow n + 1$;
    else `/* we have a more complex child */`
      S.push(m);
    end
  end
end
return n;

4.2 Abstract Stroke Ordering

Determining the sequence of stroke types in a character, without regard to their positioning, can be useful in some situations. For example, the Wubihua (五笔画) input method for Chinese uses this sequence to allow typists to input characters. Strokes are divided up into broad classes, which are then mapped to keys on the numerical keypad. These are then input in the order they'd be written by hand, which then leads to a list of characters that match the partial stroke order. Extracting such

a list is simple in SCML:

Algorithm 2: `ordered_stroke_list()`

Input: A character structure, C
Output: A list containing the strokes in C, in their written order

```
L ← new List;
S ← new Stack;
S.push(C.root);
while not S.empty do
    s ← S.pop;
    foreach c ∈ s.children do
        if c is a stroke then /* we have a stroke, so add its type to the end of our list */
            L.append(c.type)
        else /* we have a more complex child */
            S.push(m);
        end
    end
end
return L;
```

4.3 Visual

A visual view on SCML is one that generates a visual representation from an abstract structure. This might be an on-screen glyph, a stroke-order animation, or a set of instructions to guide a brush-wielding robotic arm.

To create such a view, an *aesthetic* must be codified to transform between character graphs and a visual representation. This is a set of principles guiding both the spatial layout of strokes, and their visual appearance. For convenience, the system can be split up into a layout engine and a rendering engine. A given layout engine might employ a variety of mechanisms to achieve the desired layout, from adapted graph drawing algorithms (such as force-based layouts) to applications of linear programming. The renderer can then process these layouts in a manner similar to CDL (see section 2.1) and generate various kinds of glyphs, stroke-order animations, and the like.

It is likely that these kinds of views will generate highly regular character glyphs, unless significant work goes into fine-tuning the algorithms used. Another possibility for designers is to break the layout/render separation and take advantage of the high degree of component reuse in characters. This would be made possible by the mechanisms behind the SCML Database outlined in section 5.1, which allow for efficient lookup by character substructures. With such a system, a designer would be able to use components as primitives rather than strokes, and thus give their glyphs a less regular feel (a desirable feature in a print-quality font.)

This section will outline the implementation of a linear-programming based approach to layout, and explain issues involved in the creation of renderers.

4.3.1 Layout

The layout of character components, due to all the constraints an SCML model describes, is performed by constructing a linear program, which is then solved using the simplex method. Each stroke, when being prepared for layout, provides a set of linear inequalities and equations to describe its layout constraints. These include requirements for the positioning of anchors, the size of stroke segments, their relative positioning, and requirements for the positioning of locations created by strokes. The objective function of the linear program tries to minimize linear distances between anchors, but can be adapted to include various other heuristics.

Using the simple character ＋ as an example, one can see that it contains two strokes and an anchor joining them. The layout engine starts by enumerating the strokes present in the character and then requests from each stroke instance a set of constraints describing its layout. In this case, it will start with the `h` stroke, which will instantiate seven non-negative variables for itself: $\texttt{origin}_x$, $\texttt{origin}_y$, $\texttt{inside\_anchor}_x$, $\texttt{inside\_anchor}_y$, $\texttt{distance\_from\_ideal}_x$, $\texttt{distance\_from\_ideal}_y$, and `length`, and provide the following constraints on the values of those variables:

$$\begin{aligned}
\texttt{origin}_x + \texttt{length} &< 1 \\
\texttt{inside_anchor}_x &> \texttt{origin}_x \\
\texttt{inside_anchor}_x &< \texttt{origin}_x + \texttt{length} \\
\texttt{distance_from_ideal}_x &= |\texttt{inside_anchor}_x - \texttt{origin}_x - \texttt{length}/2| \\
\texttt{distance_from_ideal}_y &= |\texttt{inside_anchor}_y - \texttt{origin}_y|
\end{aligned}$$

Likewise, the `s` stroke will also create different instances of analogous variables, with very similar constraints:

$$\begin{aligned}
\texttt{origin}_y + \texttt{length} &< 1 \\
\texttt{inside_anchor}_y &> \texttt{origin}_y \\
\texttt{inside_anchor}_y &< \texttt{origin}_y + \texttt{length} \\
\texttt{distance_from_ideal}_x &= |\texttt{inside_anchor}_x - \texttt{origin}_x| \\
\texttt{distance_from_ideal}_y &= |\texttt{inside_anchor}_y - \texttt{origin}_y - \texttt{length}/2|
\end{aligned}$$

Additionally, the layout engine will know that because they represent the same anchor, the two strokes' $\texttt{inside\_anchor}_x$ and $\texttt{inside\_anchor}_y$ must coincide, so it will add two equations to ensure that. Then, the two strokes will provide their objective functions (in this case simply to minimize the sum of the `distance_from_ideal` variables) allowing the layout engine to solve the linear program, and finally to apply the variable bindings to lay out the strokes.

The reader may have noticed that the `distance_from_ideal` equations are not technically linear, since they use the absolute value function. It is however possible to convert these to purely linear equations[8]. Fortunately, minimizing absolute values does not require the use of expensive integer variables, so the performance of the solver remains high even with the distance-minimization objective functions.

[8]See [19] for more information on the use of absolute values in (in some cases, mixed-integer) linear programs

This procedure is generalized by providing each stroke type with two methods, `get_anchor_variables()` and `get_constraints()`. The former simply creates an `x` and `y` variable for each anchor used in the stroke. The latter retrieves a list of the anchors associated with it and their positions, instantiates `x` and `y` variables for each point of intersection, and provides constraints on those variables based on its own specifications.

4.3.2 Rendering

Rendering is mostly straightforward, but there are some small issues to note. The first is that visual stroke models need to be 'flexible,' as different segments of a complex stroke might have different lengths depending on decisions made by the layout engine. Secondly, it is possible for some renderers to fold multiple stroke types into one type. This is seen in some typefaces, in which for stylistic reasons designers choose to make two technically distinct stroke types into one. In almost all cases, this creates no ambiguity, but it is important to be aware of the practice.

4.4 Further Work

In this section, I have discussed the creation of basic views and provided some examples. The following section discusses the possibility for further work on SCML views.

4.4.1 Cursive Glyph Synthesis

Chinese character calligraphy is a fascinating field in itself. Several styles of calligraphy exist, but the three main styles that are used internationally are *Standard Script* (楷書,) *Running Script* (行書,) and *Grass Script* (草書.) Standard Script resembles print characters and is easily legible, but Running Script is more cursive and more difficult to read, and Grass Script is illegible to the untrained eye[9].

(a) Standard Script (b) Running Script (c) Grass Script

Figure 12: Different calligraphy styles for the character 疲.

As can be seen in figure 12, the script styles get progressively more curved, connected, and difficult. They are not, however, impossible to read, implying an underlying system of stylistic rules, so it should be possible to create a rendering engine that generates cursive glyphs (these would probably not be aesthetically pleasing to an expert, but should at least be recognizable) from SCML graphs.

5 Databases

An SCML database, to be useful, needs to be able to perform meaningful searches on structures as well as simple lookups. A typical relational database cannot do this directly and is therefore not suitable for the purpose. Section 5.1 covers the concepts behind the current implementation of the SCML structural database. Section 5.1.2 then discusses strategies for augmenting the basic structural

[9] An even more cursive style of Grass Script exists, called 狂草, or Wild Script[20], but this style is relatively uncommon.

database with linguistic information. Section 5.2 then discusses possible directions for further work in the field of SCML databases.

5.1 SCML DB

Simple flat storage of SCML definitions would be hard to search quickly and flexibly. All queries would need to iterate over each character in the database and check to see if it matched the criteria in the query. Unlike a typical sorted array, there is no obvious sort key for character structures, so binary searches would be impossible. While flat storage is one approach to the problem and is certainly the simplest to implement, it is inefficient in terms of both time and space, as it ignores all the instances of shared substructure in the characters.

Desirable characteristics and possible applications of an SCML database are:

1. Fast (sub-linear) character lookup from external indexing system (e.g., Unicode code point) to SCML definition. This would be used any time an SCML description was needed from an external source.

2. Fast character lookup from SCML definition. This could be used by standards organizations to ensure duplicate characters are not added, and to look up characters based on a graphical SCML input method.

3. Fast substructure lookup, to find all characters that contain a given SCML structure. This would be a convenient method for people to look up characters based on substructures they recognize, and would ultimately be a more flexible radical system.

4. Space efficiency, for use on mobile/embedded devices. SCML can form a basis for the data that goes into IMEs, which are often used on portable phones, PDAs, etc. It would also be useful to provide stroke-order diagrams on portable electronic character dictionaries, and other structural lookup capabilities.

5. Standard database ACID properties, because they are useful in most circumstances.

In addition to the above, for the sake of completeness, it is desirable to associate the abstract structures with data from the languages in which they are used. This would allow, for example, the association of Japanese pronunciations and meanings with one variant of a character and Chinese pronunciations and meanings with another. These variants may only have subtle differences between them, such as stroke order, but the database would be able to preserve this information. Linkages to the same prototypical character could also be maintained from both these variants, so that the common roots of both could be preserved. This is discussed more in section 5.1.2.

SCML DB currently allows searching in terms of coarse substructure (which include strokes) only. Fine-grained substructure (see 3.1.3) is preserved but is not searchable, so if one were to search for characters containing '三' results would include 匹 but not 丰 or 玉.

5.1.1 Basic Implementation

The first four characteristics are given by the underlying data structure[10]. At its most basic, the database is a large directed acyclic (multi)graph. It stores characters by inducing them as subgraphs of

[10]The fifth is given by the use of Berkeley DB for the underlying storage.

itself, and keeping external pointers to their root nodes. To do this efficiently, it ensures the uniqueness of every entity in the database, referencing existing structures if they are present, and creating them if they are not. In this respect, it is essentially similar to a trie (prefix tree), in that the entire 'pathway' to the character being added is constructed when adding it to the data structure. The uniqueness is also preserved within characters themselves, so all characters that use a given stroke have an edge leading into that stroke's node. This means that the character graphs, when inside the database, are significantly different from how they are outside of it. There is, however, a bijective mapping between the two graph types, so this alternate representation comes at no cost in data integrity, and has significant benefits that aid search algorithms.

A minor detail to note is that the database preserves only the structure of a character, not its XML representation. Comments, XML formatting, etc. are therefore not preserved. This includes anchor and location IDs, where only uniqueness is preserved, and not the actual ID string. When reading characters out of the database, sequential numeric IDs (unique only within the context of a character) are assigned.

The database graph is divided into two separate structures: the main graph, and the component store. The main graph expresses all the layout and location structure, while the component store maintains components, and their anchors and location specifiers.

Before insertion, components are detected in the input character graph (see 3.1.3,) and are placed into a set, with no duplicates. The character graph is then expressed in terms of the elements in the component set, combining them using layouts, or expressing locations in terms of labeled (with a database-unique location ID) edges leaving components.

After this pre-processing, the insertion relies on recursive calls to the search algorithm (algorithm 3) below. At its most essential, the inserted structure is first searched for using `db_search()`, and if found, its ID is returned, otherwise its children are inserted using the same procedure and then the parent node is added to the graph joining them. If a component (including strokes) is encountered, it is inserted into a separate component set, given a unique ID, and referenced from there (see below.) Using this method for insertion ensures that every entity in the database is unique.

The separate component store keeps each component as an unmodified graph, and maintains a hash table keyed on components' structures. In the case that strokes of a component contain location specifiers and the component is already present in the store, the existing version simply has the location specifiers added to it. The component store keeps track of which locations are provided by the components it stores, for use by the main database graph. As an example, figure 13 demonstrates three successive additions to the database. In 13(a), the 大 component is first added to the database, with no locations specified in it. 13(b) adds a new character to the database, but it includes the same component as 13(a), so no new components are added to the store. It does however use a new location in that component, so that location is added to the component specification and referenced by the database graph. In 13(c)[11], the new character once again includes the 大 component, but it uses a different location in it. Thus the new location is added to the 大 component record in the component store, and is referenced from the main database graph.

References from the main database graph are made by creating 'instance nodes' of a component in the main graph, which include the unique component id, locations used by that particular instance, and outgoing edges to the structures contained in the locations.

By maintaining the entity uniqueness invariant we ensure that if a character structure contains

[11]Note that although the second and third strokes may appear to be attached to different anchors in the latter two characters, this is just a slight variation in the typeface used in this paper. Most typefaces show them to be attached.

(a) Component but no locations (b) One location (c) Two locations

Figure 13: Additions to the component store.

a given substructure, it must have a path into that substructure. This allows us to define a recursive algorithm for finding a given structure by starting from leaf components and performing repeated "intersect parent sets" operations on sets of nodes. The base case is with component instances, since each has one node dedicated to it in the database. In pseudocode, this algorithm is as follows:

Algorithm 3: `db_search()`

Input: An SCML entity, E
Output: The node id of E in the database, or `null` if not present

```
/* strokes is a map from stroke type to the unique node id of that stroke */
/* get_node(id) returns node information for a given node id */
/* get_parents(id) and get_children(id) return the set of a node's parents and its
children, respectively, given a node id */
if E is a component then /* we have a component reference */
    /* see above */
else /* the entity is a layout */
    P ← ∅;
    foreach c in E.children do
        c_id ← db_search(c);
        if c_id = null then
            return null;
        end
        if P = ∅ then
            P ← get_parents(c_id);
        else
            P ← P ∩ get_parents(c_id);
        end
    end
    foreach p in P do
        if get_node(p).layout_type = E.layout_type and
        get_node(p).child_count = E.child_count then
            return p;
        end
    end
end
return null;
```

Deletion is as simple as calling search, checking if any other characters reference the returned node

(indicated by the presence of incoming edges to that node,) and removing it and any free children if there are no references.

Performing precise run-time analysis of the above algorithms is equivalent to analyzing `db_search()`, which is non-trivial, and will be left to further work. It does however perform well in test situations, and appears at first glance to exploit the shared substructure of characters in a reasonable manner, using it to reduce both space and time requirements.

5.1.2 High-Level Metadata Storage

From most dictionary users' perspectives, a pure SCML database is not very useful. Unknown characters can be looked up, but not much else can be done with the information. This section briefly outlines general approaches to creating a complete general-purpose database on top of SCML.

The first step is to create a separate numbering system by which to allocate high-level linguistic characters. Linguistic characters differ from SCML characters in that they are associated with concrete languages, and because of variants, may be mapped to many low-level SCML characters. This would allow dictionary compilers to express all the existing variant types based on character etymologies, as well as national stroke-order variants and simplified variants. Linguistic characters would then be associated with typical dictionary data, such as readings in different languages, associated words, and so on (with the notable omission of stroke count, which can be retrieved directly from the associated SCML characters.) Radical classifications could be included for historical and research purposes, but would be largely unnecessary as a search mechanism.

SCML's high-level linguistic characters could then be given cross-references to and from code points in other major standards such as Unicode, GB18030, Big5, and JIS.

5.2 Further Work

While the current database implementation achieves the basic requirements, it lacks features that would make it particularly useful in real-world environments. The following sections provide some directions for further work that would improve the database and related applications.

5.2.1 Query Languages and Optimization

An SCML database query language must be able to take advantage of the features SCML provides. It must therefore be able to deal with substructure searches, and would ideally take this further and allow for full wildcard-like structural searches. Due to the verbose nature of the XML structures themselves, in most queries Unicode characters would be used in their place. The language might be based on SQL (as in the examples in figure 14 below,) but would have different keywords and functions to represent common operations on the database.

```
SELECT
DISTINCT(UTF8(structure)),
STROKE_COUNT(structure)
FROM characters
WHERE structure INCLUDES '日'
AND structure INCLUDES '雨'
```

```
SELECT structure,
STROKE_COUNT(structure)
FROM characters
WHERE structure
IN VARIANTS('国')
```

Figure 14: Examples of anchors and locations.

Note that the sample queries are aware (and should ideally be) of the additional metadata discussed in 5.1.2, as this provides more flexibility than pure structural queries would. It, for example, would allow one to count the number of characters that are phono-semantic compounds(形聲[12].)

5.2.2 Stroke Order Folding

To be fully useful for unknown characters, the database needs to be able to find characters even if the queried stroke order is incorrect. The process is somewhat akin to case insensitivity in the latin alphabet, which is generally accomplished by case folding (converting all characters to a predefined case) in indices. It should be possible to perform a similar transformation on SCML characters, by creating a canonical set of stroke order rules, and maintaining two copies of each character in the database. Order-insensitive queries would be transformed according to the canonical rules, and run against the set of pre-transformed characters, whereas order-sensitive queries would run against the unmodified set of characters.

5.2.3 Fine-grained Substructure Searches

Providing this feature would be complicated. Take for example the character in figure 15. The character's SCML definition specifies it in terms of 二 with a vertical stroke running through both of its horizontal strokes, but it is clear that the character also induces the other two substructures. Finding the other induced structures becomes an exercise in combinatorics, but as the number can grow exponentially, storing and linking all these explicitly to the primary structure would be inefficient. This would be further complicated by maintaining stroke-order insensitivity (see section 5.2.2 above) in the database.

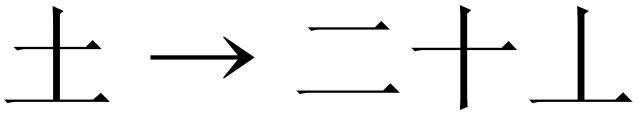

Figure 15: Possible fine-grained substructures for a character.

6 Automated Import

In this final section, I discuss ideas related to importing the large body of existing characters into SCML, in a largely automated fashion. Given the existence of over 120,000 of these, this would likely take several years to do manually even with a team of skilled workers and graphical composition tools. The idea relies on the existence of a freely available outline font called *Han Nom A/B*[13]. The process is similar to OCR, but tries to take advantage of the higher-level vector data provided by the outline font to reconstruct SCML definitions efficiently from the glyph. The font was selected because of its highly regular design (which has led to some aesthetic criticism for it) and its broad coverage (it includes over seventy thousand glyphs covering all the Chinese characters currently defined in Unicode.)

This process essentially consists of three steps: stroke segmentation, where the character is decomposed into strokes; intersection detection, in which intersections between strokes are identified

[12]These are characters in which a phonetic component gives a character its pronunciation and a radical gives it a general class of meaning.

[13]This is the same font used to typeset most of the Chinese characters in this paper.

and stores; aggregation, where the information from the first two steps is composed into a logical structure and stroke order is inferred.

Although the process described here has not yet been implemented, the following three sections describe in moderate detail how I envision it working. It is, however, important to realize that, although helpful, the generated SCML definitions will still need to be checked for consistency and correctness by a human being. It is unrealistic to expect it to consistently infer the correct stroke order, or to always be able to distinguish between similar stroke types, even with the best of automated import algorithms.

6.1 Stroke Segmentation

Segmentation is the process of taking the glyph outline and decomposing it into strokes, ultimately creating a CDL-like(see section 2.1) set of strokes (unordered, as the ordering cannot be inferred easily at this point) with associated types, and control points. This process is simpler than typical OCR because the source is not a raster, so typical image preprocessing steps (edge detection, thinning) are not necessary.

Different stroke types need to be detected individually and in a reverse-topological order. That is, strokes whose geometries include geometries of simpler strokes (such as ㄱ, ㄱ, or ㄷ, which include the geometry of ㅡ) must be detected first and then removed from the outline, to avoid duplicate detection of more complex strokes' subsegments as simpler strokes.

The detection algorithms will vary depending on stroke type, but will generally work in roughly the same manner. Outline fonts consist of a series of closed paths that define regions. These paths, in most current fonts, consist of linear segments and quadratic Bézier splines[14]. I will discuss primarily issues related to linear segments, but in this context quadratic splines are not significantly more complex:

(a) Broken Segments (b) Stroke ornaments

Figure 16: A simple glyph to illustrate segmentation issues.

In figure 16, we can see some of the issues that arise when trying to determine the strokes that compose an outline. Due to the intersection between the horizontal and vertical strokes in the character, the segments that compose the main lengths of the strokes are split into two, creating in this case four principal (non-ornamental) linear segments per stroke.

Linear segments of a path are defined by their start and end points. They have a slope (as an angle, to avoid complications with vertical segments) and, if extended to form a line, a intercept. These segments can be placed into a search tree keyed by the segments' slope angle, with duplicates allowed and sorted by intercept. It is then possible to use this data structure to group segments that are collinear and parallel. In figure 16, the broken principal segments that make up each side of each stroke could be reconstructed easily by finding 'duplicates' (segments with identical slope and

[14]Older Type 1 fonts allow for cubic splines as well.

intercept) in the tree[15]. After merging these, the strokes should have two principal segments each. The pairs can then be easily united into a single stroke, as the data structure will mark them as parallel, and a distance calculation will decide if they are within some threshold distance of one another. Ornaments (see figure 16(b)) can also be detected and checked using more advanced heuristics on the paths between the end points of the two principal segments for each stroke. They must however not be relied upon as terminal ornaments may be occluded by other strokes.

This process is repeated until all strokes have been detected (that is, all paths in the character have been covered) and classified.

6.2 Detecting Intersections

Once the strokes have been detected, their associated vector data is reconstructed on the plane, ideally creating an exact copy of the original outline glyph, with a separate closed path for each detected stroke. The individual stroke paths are then checked pairwise for intersections using CAG (Constructive Area Geometry) operations. As this would normally be an $O(n^2)$ operation in the number of strokes, and CAG is computationally expensive, the process of selecting candidates for intersection can be made significantly more efficient by employing an R-tree[16] on stroke bounding boxes.

One point to note is that an attempt must be made to avoid false intersections. As can be seen in figure 17, these are small intersections that aren't part of the basic character structure, but which occurred in the glyph due to mistakes, aesthetic choices, or cramped space. It should be possible to detect them by extending the strokes and ensuring the area of intersection is above a certain threshold.

(a) False intersection between strokes 1 and 3. (b) False intersection between stroke 4 and 8.

Figure 17: False intersections in glyphs.

The locations of detected intersections can then be classified based on where they lie in the strokes they fall on, and assigned named positions based on strokes' specifications.

6.3 Aggregation and Stroke Order

Once glyphs have been decomposed into strokes and intersections between these have been determined, it is possible to find maximal sets of intersecting strokes and thus to reconstruct components. Components can then have their bounding boxes' locations compared in order to aggregate them: if one component's bounding box is strictly left/right/above/below another's, then the components can be composed using layouts; if there is intersection in the bounding boxes, then a location is needed to express the relationship.

At this point, an SCML graph has been generated. It may, however, not have a correct stroke order. For row and column layouts, physical position implies stroke order, so that cannot be changed,

[15] In a larger, more complex character, the algorithm would have to verify that the duplicates lie on the same closed path to prevent false positives.

[16] An R-tree is a data structure used for spatial indexing. It does not have good worst-case performance but performs well in most cases. If worst-case performance becomes an issue, the more efficient Priority R-tree[21] can be used.

but free layouts and locations can have correct graphical representations with incorrect stroke order. It should nonetheless be possible to correct the order by applying standard[17] stroke order rules (e.g., "Write from left to right, and from top to bottom" or "Outside before inside"[22].)

7 Further Work

There is much room for further work in this field. By its design, SCML is a model, and although this model has specific semantics, it has no prescribed views or applications. This paper explored some applications, but countless others can be developed on top of the model. This section will cover three other areas of research that are either based on the SCML model or might use it in a useful manner.

7.1 Similarity Metrics

Similarity metrics provide a means to compare two characters. They can be useful to standards organization in the process of determining whether or not to unify two characters that differ slightly, or to order results of a database query, if someone is searching for all characters within a certain similarity distance from a known one.

There are two main classes of similarity algorithms: structural and geometric. The former operate directly on the SCML graphs, whereas the latter work on the implied geometries to determine equivalence. These latter should be mostly unnecessary due to structural conventions used in defining SCML graphs, but might sometimes be necessary to compare structures that are commonly considered equivalent but that have different stroke orders.

Structural similarity algorithms could be formulated like string edit distance algorithms, but would require a much more complicated definition of an *edit*, in the context of character structures. This is certainly possible and would provide a useful tool for several other SCML applications.

7.2 Input Method Editors

Several structural input methods exist to aid Chinese character input. Some are based on unique ideas, such as the four-corner code, while others like Wubihua rely on simple concepts such as stroke order. An SCML database could help these by providing them with a consistent body of data that could be processed to generate data specific to their application.

It might also be possible to develop entirely new IMEs that rely on SCML data. An effective handwritten interface would be feasible, for example, as it would be possible to record strokes as they occur (and distinguish between strokes by pen presses) while maintaining the written stroke order intact.

7.3 Optical Character Recognition

It may also be possible to use SCML as a basis for a constructive approach to Chinese character OCR. If it were possible to dissect bitmaps into strokes as was done on outline fonts in section 6.1 (granted, this would be significantly harder,) then a series of steps could be performed to reconstruct an SCML definition, which could then be looked up in a database.

[17] There are actually three main standards with slight variations, but the general principles are the same.

8 Conclusion

In this paper, I have presented a new approach to Chinese character representation that is fully divorced from aspects of the visual representation of characters. This is of interest to several major sectors of the CJK and international computing worlds. I have explored the benefits such an approach brings to the field and have discussed my preliminary implementations of some SCML-based applications, as well as going into significant detail on possible approaches to the implementation of other applications.

Due to limited time and the vast number of characters, it was impossible for me to study all the existing characters, so future research might reveal limitations of the basic model that require slight changes to the existing constructs. I am nevertheless confident that SCML's basic approach of representing characters as graphs is both feasible and useful, and could significantly help many users in the CJK computing world.

A Appendix: Stroke Types and Associated Information

The following is a list of all stroke types available in SCML, along with the named positions and locations currently defined on each stroke type (see 3.1.2.) Naming of stroke types is based on the system used in CDL[9]. Sample glyphs for each stroke, inspired by common typeface styles, are provided to illustrate the correspondence between logical strokes and their typical visual representations. They are by no means the only possible style for strokes, but are the ones currently used as models by the renderer implementation (for those strokes that have been implemented.)

Name	Sample Glyph	Anchor Positions	Locations
h		begin, inside, end	above, below
hg		begin, inside, end, hook-tip	above, below
h1		begin, inside, end	above, below
p		begin, inside, end	above, below
t		begin, inside, end	above, below
ta		begin, end	left, right
pa		begin, inside, end	left, right
pb		begin, inside, end	left, right
d		begin, inside, end	above, below
s		begin, inside, end	left, right
s1		begin, inside, end	left, right
sg		begin, inside	left, right
n		begin, inside, end	left, right
na		begin, inside, end	above, below
xg		begin, inside	left, right
pn		begin, inside, end	above, below
sp		begin, inside, end	left, right
pg		begin, inside	left, right
wg		begin, inside	left, right
sz		begin, inside0, corner, inside1, end	inside, left, below

Continued…

Name	Sample Glyph	Anchor Positions	Locations
swg		begin, inside0, inside1, end	inside, left, below
hz		begin, inside0, corner, inside1, end	above, inside, right
hzl		begin, inside0, corner, inside1, end	above, inside, right
hzg		begin, inside0, corner, inside1	above, inside, right
szz		begin, inside0, corner0, inside1, corner1, inside2, end	left, above, below, right
hzz		begin, inside0, corner0, inside1, corner1, inside2, end	above, left, right, below
hzwg		begin, inside0, corner, inside1, inside2	above, left, inside, below
hxwg		begin, inside0, corner, inside1, inside2	above, left, inside, below
hpwg		begin, inside0, corner0, inside1, corner1, inside2	above, left, right
hzzp		begin, inside0, corner0, inside1, corner1, inside2, corner2, inside3, end	above, left, right
hzzzg		begin, inside0, corner0, inside1, corner1, inside2, corner2, inside3	above, left, right
szwg		begin, inside0, corner0, inside1, corner1, inside2	left, right
st		begin, inside, end, hook-tip	left, right

Continued…

Name	Sample Glyph	Anchor Positions	Locations
hzt		begin, inside0, corner0, inside1, end, hook-tip	above, left, right
pd		begin, inside0, corner, inside1, end	above, below, right
pz		begin, inside0, corner, inside1, end	above, below, right
pz1		begin, inside0, corner, inside1, end	above, below, right
tn		begin, inside0, corner, inside1, end	above, below, right
tn1/hzw		begin, inside0, corner, inside1, end	above, below, right
hzwg1		begin, inside0, corner, inside1	above, below, right

B Appendix: XML Schema for SCML

The following is an XML schema for SCML, expressed in the RELAX NG compact language. The language is discussed in detail in section 3.2.

```
start = SCML
SCML = element scml { Entity }
Entity = Layout | Stroke | Location
Layout = element layout { attribute type { text }, Entity* }
Location = element location { attribute id { text }, Entity, LocationAnchor }
Stroke = element stroke { attribute type { text }, (Anchor | StrokeLocation)* }
Anchor = element anchor { attribute id { text }, attribute at { text }, empty }
LocationAnchor = element anchor { attribute id { text }, empty }
StrokeLocation = element location { attribute id { text }, attribute at { text }, empty }
```

References

[1] Unicode glossary [online, cited May 28, 2007]. Available from World Wide Web: http://www.unicode.org/glossary/.

[2] Chinese writing '8,000 years old' [online]. May 2007 [cited May 19, 2007]. Available from World Wide Web: http://news.bbc.co.uk/2/hi/asia-pacific/6669569.stm.

[3] Ideographic Rapporteur Group. CJK extension C1 V4.2 D4, may 2005. Available from World Wide Web: http://www.cse.cuhk.edu.hk/~irg/irg/extc/IRGN1113-CJK_C1_V42-D42.pdf.

[4] Richard Cook. Typological encoding of chinese: Characters, not glyphs [online, cited May 26, 2007]. Available from World Wide Web: http://unicode.org/iuc/iuc19/a367.html.

[5] Unicode Kangxi Radicals. Available from World Wide Web: http://unicode.org/charts/PDF/U2F00.pdf.

[6] Mark Spahn and Wolfgang Hadamitzky. *The Learner's Kanji Dictionary*. Charles E. Tuttle Co., 2004.

[7] James W. Heisig. *Remembering the Kanji I: A Complete Course on How Not to Forget the Meaning and Writing of Japanese Characters*, volume 1. Japan Publications Trading Company, 2001.

[8] CDL: Character description language [online]. Available from World Wide Web: http://www.wenlin.com/cdl/.

[9] Tom Bishop and Richard Cook. *CDL: The Set of Basic CJK Unified Stroke Types*, May 2004. Available from World Wide Web: http://www.wenlin.com/cdl/cdl_strokes_2004_05_23.pdf.

[10] Tom Bishop and Richard Cook. *A Specification for CDL*, Oct 2003. Available from World Wide Web: http://www.wenlin.com/cdl/cdl_spec_2003_10_31.pdf.

[11] Hanglyph index [online, cited May 21, 2007]. Available from World Wide Web: http://www.hanglyph.com/en/hanglyph-index.shtml.

[12] *HanGlyph: A Reference Manual*. Available from World Wide Web: http://www.hanglyph.com/en/hanglyph/reference.pdf.

[13] The Unicode Consortium, Joan Aliprand, Julie Allen, Joe Becker, Mark Davis, Michael Everson, Asmus Freytag, John Jenkins, Mike Ksar, Rick McGowan, Eric Muller, Lisa Moore, Michel Suignard, and Ken Whistler. *The Unicode Standard, Version 4.0*. Addison-Wesley Professional, 2003. Available from World Wide Web: http://www.unicode.org/versions/Unicode4.0.0/.

[14] Jin Wook Kim, Kwang In Kim, Bong Joon Choi, and Hang Joon Kim. Decomposition of chinese character into strokes using mathematical morphology. *Pattern Recognition Letters*, 20(3):285–292, 1999.

[15] Hung-Hsin Chang and Hong Yan. Analysis of stroke structures of handwritten chinese characters. *Systems, Man and Cybernetics, Part B, IEEE Transactions on*, 29(1):47–61, 1999.

[16] Zhizheng Liang and Pengfei Shi. A metasynthetic approach for segmenting handwritten chinese character strings. *Pattern Recognition Letters*, 26(10):1498–1511, 2005.

[17] Rong He and Hong Yan. Stroke extraction as pre-processing step to improve thinning results of chinese characters. *Pattern Recognition Letters*, 21(9):817–825, 2000.

[18] Relax ng compact syntax tutorial [online, cited May 29, 2007]. Available from World Wide Web: http://relaxng.org/compact-tutorial-20030326.html.

[19] lp_solve reference guide [online, cited May 28, 2007]. Available from World Wide Web: http://lpsolve.sourceforge.net/5.5/.

[20] Cursive script (East Asia) [online, cited May 27, 2007]. Available from World Wide Web: http://en.wikipedia.org/w/index.php?title=Cursive_script_%28East_Asia%29&oldid=130627504.

[21] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. In *SIGMOD Conference*, pages 347–358, 2004.

[22] Stroke order [online, cited May 28, 2007]. Available from World Wide Web: http://en.wikipedia.org/w/index.php?title=Stroke_order&oldid=133077494.