The NiCad Clone Detector

James R Cordy
Queen's University, Kingston, Canada
Email: cordy@cs.queensu.ca

Chanchal K. Roy
University of Saskatchewan, Saskatoon, Canada
Email: croy@cs.usask.ca

***Abstract*—The NiCad Clone Detector is a scalable, flexible clone detection tool designed to implement the NiCad (Automated Detection of Near-Miss Intentional Clones) hybrid clone detection method in a convenient, easy-to-use command-line tool that can easily be embedded in IDEs and other environments. It takes as input a source directory or directories to be checked for clones and a configuration file specifying the normalization and filtering to be done, and provides output results in both XML form for easy analysis and HTML form for convenient browsing. NiCad handles a range of languages and normalizations, and is designed to be easily extensible using a component-based plugin architecture. It is scalable to very large systems and has been used to analyze, for example, all 47 releases of FreeBSD (60 million lines) as a single system.**

***Keywords*-tools, clone detection, NiCad, plugin architecture**

I. The NiCad Method

NiCad [1] is a new clone detection method that has been shown to yield both high precision and high recall [2] in detecting near-miss intentional clones. NiCad is a hybrid method that combines language-sensitive parsing with language-independent similarity analysis to yield structurally meaningful near-miss clones.

The NiCad method involves three main stages, *parsing, normalization,* and *comparison*. In the first stage the input sources are parsed to extract all fragments of a given granularity, such as functions or blocks. Each extracted fragment ("potential clone") is pretty-printed into a standard textual form. Spacing and line breaks are normalized and comments are removed to yield a form that exposes Type 1 (exact) clones as textually identical fragments.

In the second stage, extracted fragments can be normalized, filtered or abstracted before comparison. For example, they can be transformed by renaming, standard parenthesization, or removal of declarations.

In the comparison stage, the extracted and normalized fragments are linewise compared using an optimized LCS (longest common subsequence) algorithm to detect similar fragments (clones). Comparison is parameterized by a difference threshold that allows for near-miss detection. For example, a difference threshold of 0.0 detects only exact clones, 0.1 detects those that may differ by up to 10% of their normalized lines, 0.2 by up to 20%, and so on. Unlike most other methods, NiCad builds clone classes directly, as part of comparison, rather than clustering separately.

II. The NiCad Clone Detector

The NiCad Clone Detector (Figure 1) is a free open-source implementation of the NiCad method designed to be flexible, extensible and embeddable in IDEs and other applications. It is designed to have a simple command-line interface that allows for easy scripting and combination with other tools and analyses. NiCad is invoked from the Linux, Solaris, Cygwin or Mac OS X command line by simply giving the desired granularity of processing, the language of the source files to be processed, and the root directory of the source system to be analyzed, for example:

```
% nicad functions java ./JHotDraw54b1
```

At present NiCad supports two granularities, functions and blocks, and five languages, C, C#, Java, Python and WSDL. New languages and granularities can be added simply by adding a new TXL parser or extractor for the new language or granularity to the plugins directory - see Section IV. By default NiCad runs with no normalization, filtering or renaming and simply finds exact and near-miss clones at four difference (UPI) thresholds: 0.0, 0.1, 0.2 and 0.3, corresponding to 0%, 10%, 20% and 30% different lines in the normalized extracted fragments (e.g. Java methods).

A summary of progress and results is printed on the standard output, and the main output is stored in a newly created directory in the same directory as the original system source, in this case *./JHotDraw54b1_functions-clones/* where the results in both XML form and as browsable HTML pages are stored. FIgure 2 shows an example of the web page output of the command above for the 0.3 threshold.

To specify normalization, filtering or renaming, the user adds the name of a configuration file to the command:

```
% nicad functions java ./JHotDraw54b1 blindrenaming
```

The "blindrenaming" part means that the configuration file named *blindrenaming.cfg* is to be used.

Configuration files (Figure 3) allow the user to specify a range of options, including the set of near-miss thresholds to be reported, the minimum and maximum size of clones (in number of pretty-printed lines), the renaming to be

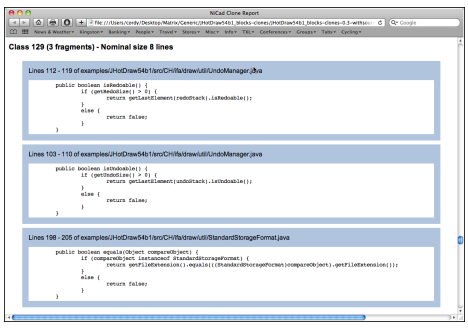

Figure 2. NiCad HTML web page output

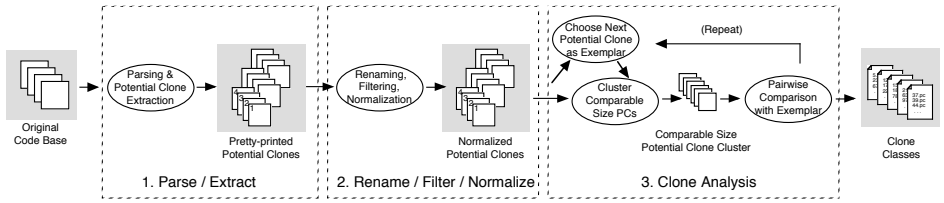

Figure 1. The NiCad Clone Detector

done (blind or consistent), syntactic forms to be filtered out (e.g., declarations), syntactic forms to be abstracted (e.g., expressions), and custom contextual normalizations to be applied (e.g., abstract if conditions).

Each option specified in the configuration file invokes a NiCad plugin to be run on the potential clones (extracted fragments) before comparison. Plugins are transformations, such as renamers, normalizers and abstractors, implemented in TXL and stored in the NiCad plugin directory (Section IV). Users can add any new normalization they wish as a TXL program using the language grammars provided. The filtering and abstraction plugins are special, in that they are generic - they automatically handle any set of forms named in any language's grammar.

Blind (all identifiers *"X"*) and consistent (same identifiers *"Xn"*) renamings, as well as filtering (removal) and abstraction (replacement by *"S"*) of any syntactic form *S* before comparison is provided for all languages.

III. Incremental NiCad

NiCad also provides an incremental mode, in which a system that has already been analyzed is re-analyzed only for new clones. In this mode, NiCad reports only clones that have arisen as a result of the new version - that is, those that cross the boundary between the old and new versions of the system. If only changed files are given as the new system, this allows for very efficient incremental clone detection.

Incremental clone detection can also be used to compare two different systems for cross-system clones - for example,

```
# NiCad configuration file
# blindrename-filterdeclarations-abstractexpressions

# Set of thresholds we are interested in
thresholds="0.0 0.1 0.2 0.3"

# Sizes of clones we are interested in
minsize=5
maxsize=500

# Kind of renaming to be applied
rename=blind

# Kind of filtering to be applied
filter=declaration

# Kind of abstraction to be applied
abstract=expression

# Custom contextual normalizer to be applied
normalize=none

# End of NiCad configuration parameters
```

Figure 3. Example NiCad configuration file

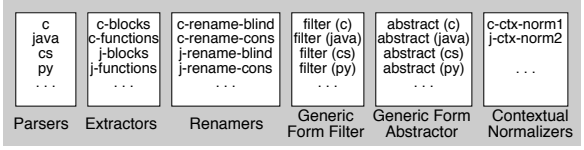

Figure 4. NiCad plugin architecture

if we use Linux as the original system and FreeBSD as the "new version", NiCad will report clones that cross between Linux and FreeBSD. Incremental NiCad simply adds another command line argument for the new version:

```
% nicadincr blocks c linux/linux-2.6.24.2
              freebsd/8.0-RELEASE consistentrename
```

IV. Plugin Architecture

NiCad is designed with a plugin architecture to allow for easy addition of new languages, granularities, renamings and custom normalizations (Figure 4). NiCad recognizes new plugins by naming convention - for example, to add support for language *L* at granularity *functions*, we name the parser/extractor *L-functions-extract.txl* and put it in the TXL plugins directory, and NiCad will be able to use it in detecting function clones in L systems right away. The supplied extractors use robust parsing to allow for minor syntax errors and unrecognized forms in input. Seriously malformed files (typically very few) are reported and ignored.

Custom context-dependent normalizations can be added by copying a generic normalization template and adding TXL rules for the user's new normalization. Again, the new normalization program (e.g., *mynormalization.txl*) is placed in the TXL plugins directory, and NiCad will be able to use it immediately when *normalize=mynormalization* is specified in a configuration file.

V. Demonstration

In this demonstration we will interactively run NiCad on real systems to demonstrate its features and performance. NiCad is very efficient in its resource usage, and can handle even the largest systems (over 60 million lines) in 2 Gb of memory on a standard single-processor laptop. While parsing and extraction is the most expensive stage, it needs be done only once on a given system to support many clone analyses. There are of course many other tools to compare to, but there is no room to do that here. The reader is referred to existing published reviews of comparable systems [3].

References

[1] C.K. Roy and J.R. Cordy, "NICAD: Accurate Detection of Near-miss Intentional Clones using flexible pretty-printing and code normalization", in *ICPC 2008*, pp. 172-181, Amsterdam, Netherlands, June 2008.

[2] C.K. Roy and J.R. Cordy, "A mutation / injection-based automatic framework for evaluating code clone detection tools", in *Mutation 2009*, pp. 157-166, Denver, USA, Apr. 2009.

[3] C.K Roy, J.R. Cordy and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", *Sci. Comput. Program.* 74(7), pp. 470-495, May 2009.