A SAT-based parser and completer for pictures specified by tiling [1]

Matteo Pradella [a,*] Stefano Crespi Reghizzi [a,b]

[a] *CNR IEIIT-MI, Via Ponzio 34/5, 20133 Milano, Italy*

[b] *Politecnico di Milano, P.zza L. da Vinci, 32, 20133 Milano, Italy*

Abstract

Pictures or patterns have been formally specified by different methods such as grammars. An alternative approach is based on Tiling Systems (TS) (Wang tiles are an analogous and equivalent formalism), whereby the picture is obtained by first covering it with a specified set of two by two tiles, then by performing a pixel by pixel mapping. TS are a powerful technique: the corresponding pictures can be recognized by non-deterministic cellular automata, which are more powerful then the four ways automata. The difficulty to write such specifications for non elementary pictures, and the NP-complete computational complexity of TS picture recognition have so far blocked any attempt to application. We have implemented a recognizer and generator for TS pictures in a very attractive, unconventional way, by transforming the tiling problem into a SAT (Boolean Satisfiability) one, then using an efficient off-the-shelf SAT-solver. The prototype is fast enough to experiment on reasonably sized samples, and has the bonus of being able to complete or extrapolate a partial or noisy picture. The tool is invaluable to assist in writing picture specification. A series of examples shows how to specify patterns using TS.

Key words: Syntactic pattern recognition, Picture languages, Tiling systems, Wang Tiles, Two dimensional grammars and languages, Picture/Image Generation and Interpolation, SAT-solvers

* Corresponding author. Tel.: +39 0223993495; fax: +39 0223993574.
Email addresses: pradella@elet.polimi.it (Matteo Pradella), crespi@elet.polimi.it (Stefano Crespi Reghizzi).

[1] Work partially supported by MUR, Progetto "Automi e Linguaggi Formali: aspetti matematici e applicativi."

1 Introduction

Syntactic methods (see for instance [1],[2]) have been often considered for performing pattern analysis and recognition, by formally specifying the class of pictures of interest. Pictures or patterns can be specified by different methods, such as grammars or automata. A sample of approaches can be found in [3], including for instance [4], where isometric array grammars are considered for efficient syntactic pattern recognition and picture generation. An alternative, theoretically sound, yet practically unexplored, approach is to use tiling: in the crudest form a specified set of small, say two by two, tiles is listed, which can cover the intended class of pictures. A picture is recognized if, and only if, it can be covered with tiles from the listed set. To overcome the limitations of such rudimentary method, a more flexible formalism, called *Tiling Systems* (TS) has been studied by theoreticians (see e.g. [5], [6], [7]). *Wang Tiles* [8] are an equivalent variant of the formalism, which uses a more traditional concept of tiling where tiles are placed side by side. Such tiles have been used to generate rich textures in [9]. A recent variant of TS/Wang Tiles is presented in [10].

Our work is concerned with a practical experimentation of TS in conjunction with a new approach for performing pattern recognition and image generation or completion, based on powerful logical tools, the SAT-solvers, whose task is to find Boolean values which make a propositional formula true.

With TS the picture is obtained by first covering it with tiles drawn from a listed set of two by two tiles, then by performing a pixel by pixel mapping. Tiling Systems are a powerful technique: the corresponding pictures can be recognized by nondeterministic cellular automata, called *Online Tessellation Automata* [11]. Such abstract machines are more powerful then the four ways automata originally introduced in [12]. Refer to [5] for a recent survey. However TS definitions are hard to write and error-prone for non-elementary pictures. Moreover the NP-complete computational complexity of picture recognition has until now blocked any attempt to realistic experimentation and application of TS, in spite of a large amount of theoretical work.

We have implemented a recognizer/generator for TS defined pictures in a very attractive, unconventional way, by transforming the tiling problem into a Boolean satisfiability one, then using an efficient off-the-shelf SAT-solver. The prototype is fast enough to experiment on reasonably sized samples, and has the bonus of being able to complete a partial picture, by assigning to unknown pixels some values which satisfy the picture specification.

The tool is invaluable to assist in writing picture specification. Several examples are provided, such as the set of geographical maps which can be colored with three colors, and various classes of nested patterns and connected paths. The tool can

be also applied to image reconstruction or noise elimination, by parsing a picture where some pixels are tagged as unknown. Availability of the tool should ease experimentation of tiling based methods for classification and recognition or generation of certain types of pictures, in isolation or combined with more consolidated methods.

The presentation is organized as follows. Section 2 briefly presents formal picture languages and Tiling Systems. Section 3 describes how to encode the TS picture recognition problem into the SAT problem. Section 4 presents a gallery of patterns and sketches a methodology for their specification by TS. It reports experiments with the tool and performances. Section 5 mentions future research and concludes.

2 Picture Languages and Tiling Systems

We provide a gentle introduction to Tiling Systems. The reader may consult [5] for more detailed and formal definitions. Picture languages can be viewed as a generalization of textual languages from one to two dimensions.

The set of pixel values is named the *terminal alphabet* of the picture, denoted Σ. For example we may choose $\Sigma = \{0, 1\}$ to define black and white pictures, but we allow alphabets of any finite cardinality.

A *picture* p is a two-dimensional rectangular array of elements of the terminal alphabet. The *size* $|p|$ of a picture p is specified by the pair $(|p|_{row}, |p|_{col})$ of its number of rows and columns. A *pixel* $p(i, j)$, $1 \leq i \leq |p|_{row}, 1 \leq j \leq |p|_{col}$, is the element at position (i, j) in the array p. Conventionally the indices grow from top to bottom for the rows and from left to right for the columns:

$$p = \begin{array}{|ccc|} \hline p(1,1) & \dots & p(1,|p|_{col}) \\ \vdots & \ddots & \vdots \\ p(|p|_{row},1) & \dots & p(|p|_{row},|p|_{col}) \\ \hline \end{array}$$

A *picture language* is a (possibly infinite or at least very large) set of pictures over the given alphabet. It is useful to introduce the notation $\Sigma^{*,*}$ for the set of all possible picture of any size, over the same alphabet Σ.

For convenience we usually consider the *bordered version* of picture p, obtained by surrounding the picture with the special *boundary symbol* $\#$, which is assumed not to be in the alphabet:

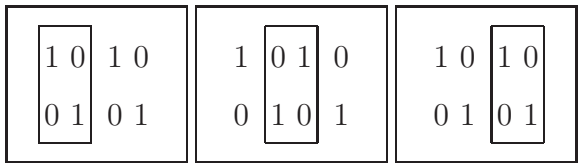


Fig. 1. The tiles (i.e. 2×2 subpictures) of a 2×4 picture.

$$\begin{array}{|ccccc|}\hline \# & \# & \dots & \# & \# \\ \# & p(1,1) & \dots & p(1,|p|_{col}) & \# \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \# & p(|p|_{row},1) & \dots & p(|p|_{row},|p|_{col}) & \# \\ \# & \# & \dots & \# & \# \\ \hline \end{array}$$

We often need to refer to some parts of a picture. A *subpicture* at position (i, j) of a picture p is any rectangular array of pixels, contained in p and having the symbol $p(i, j)$ at its leftmost, topmost corner. From this we obtain the definition of a tile.

Consider a picture p. The set of *tiles* of p, denoted $B_{2,2}(p)$, is defined by:

$$B_{2,2}(p) = \{q : q \text{ is a subpicture of } p \text{ of size } (2, 2)\}.$$

An illustration is in Figure 1.

By listing the permitted tiles, one can specify a simple family of picture languages, the so called family of *local languages*. uFor instance, with the alphabet $\{0, 1\}$, the tile set

$$\Theta_1 = \left\{ \begin{array}{|cc|}\hline 0 & 1 \\ 1 & 0 \\ \hline \end{array}, \begin{array}{|cc|}\hline 1 & 0 \\ 0 & 1 \\ \hline \end{array} \right\}$$

defines the language, denoted

$$L_1 = LOC(\Theta_1)$$

of rectangular checker boards, of any height and width.

For instance, the picture $\begin{array}{|cccc|}\hline 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ \hline \end{array}$ is valid, since it can be tiled as shown in Figure 1.

It is important to stress that it is not requested that all listed tiles be present in a picture. For example the tile set

$$\Theta_2 = \left\{ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \right\}$$

defines the language

$$L_2 = LOC(\Theta_2)$$

made by the union of two disjoint sets: the previous language L_1 of checkerboards and the language of arrays with all pixels set to 0. Clearly there exists no picture making use of all three tiles.

The use of the bordered version of pictures, instead of plain ones, improves selectivity, by allowing specification of the tiles occurring on the borders. To illustrate, the language of square pictures having all pixels on the main diagonal set to 1, and all the remaining pixels set to 0, is defined by the following tile set:

$$\Theta = \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \right\} \cup$$

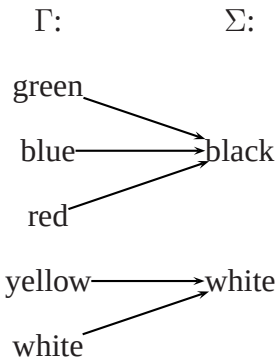
$$\left\{ \begin{bmatrix} \# & \# \\ \# & 1 \end{bmatrix}, \begin{bmatrix} \# & \# \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} \# & \# \\ 0 & \# \end{bmatrix}, \begin{bmatrix} 0 & \# \\ 0 & \# \end{bmatrix}, \begin{bmatrix} 1 & \# \\ \# & \# \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ \# & \# \end{bmatrix}, \begin{bmatrix} \# & 0 \\ \# & \# \end{bmatrix}, \begin{bmatrix} \# & 0 \\ \# & 0 \end{bmatrix} \right\}$$

where the first line lists the inner tiles and the second one lists the border tiles, as they are found scanning a bordered picture clockwise, starting from (1,1). In the following we always refer to bordered pictures. Later we use more concise descriptions of tile sets, by prototypes and predicates.

As the range of patterns, which can be characterized by the presence or absence of certain tiles, is rather poor, a more refined form of picture specification by tiling has been proposed by [13] (see also [5]) under the name of Tiling Systems. The idea is to use a larger alphabet for the tiles, in order to obtain better control on how they fit together. The tile alphabet is then mapped onto the terminal alphabet of the pictures. [2] Such model is equivalent to Wang's tiling model [8], but we prefer to use the definition of [13].

[2] Actually, there is a deep analogy with the basic result of automata theory stating that any finite-state language can be obtained from a local language, i.e. a language defined by domino tiles, by a change of alphabet.

A pixel by pixel operation, called *projection*, is used to map pixels from an alphabet to another one. Let $\pi : \Gamma \to \Sigma$ be a mapping from an alphabet Γ to an alphabet Σ. An example is the transformation from colored to black and white pixels defined by the mapping π:



The *projection* $\pi(p)$ of a picture p of alphabet Γ is a picture p' of alphabet Σ such that the two pictures have the same size and, for each pair of corresponding pixels, it holds

$$p'(i,j) = \pi(p(i,j)).$$

In the example the projection converts "dark" pixels to black, and "light" pixels to white.

Definition 1. [5] A *tiling system* (TS) T consists of

(1) a terminal alphabet Σ;
(2) a tile alphabet Γ;
(3) a set Θ of tiles of alphabet Γ;
(4) a projection $\pi : \Gamma \to \Sigma$.

The picture language $\mathcal{L}(T)$ defined by a tiling system T is

$$\mathcal{L}(T) = \pi(LOC(\Theta)).$$

Recall that $LOC(\Theta)$ is the local language defined by the tile set (using border symbols).
To illustrate we show the TS of a picture language which is not local, that is it cannot be defined using tiles over the terminal alphabet. This will serve as a running example for later sections.

Example 1. Chinese boxes on a background

A picture represent rectangular frames or boxes, placed anywhere in the plane. Frames may be nested one inside the other but they may not overlap, touch each other, or touch the border. The perimeter pixels of a frame are encoded by ■ and the background by blank pixels (denoted □) so that the terminal alphabet is $\Sigma =$

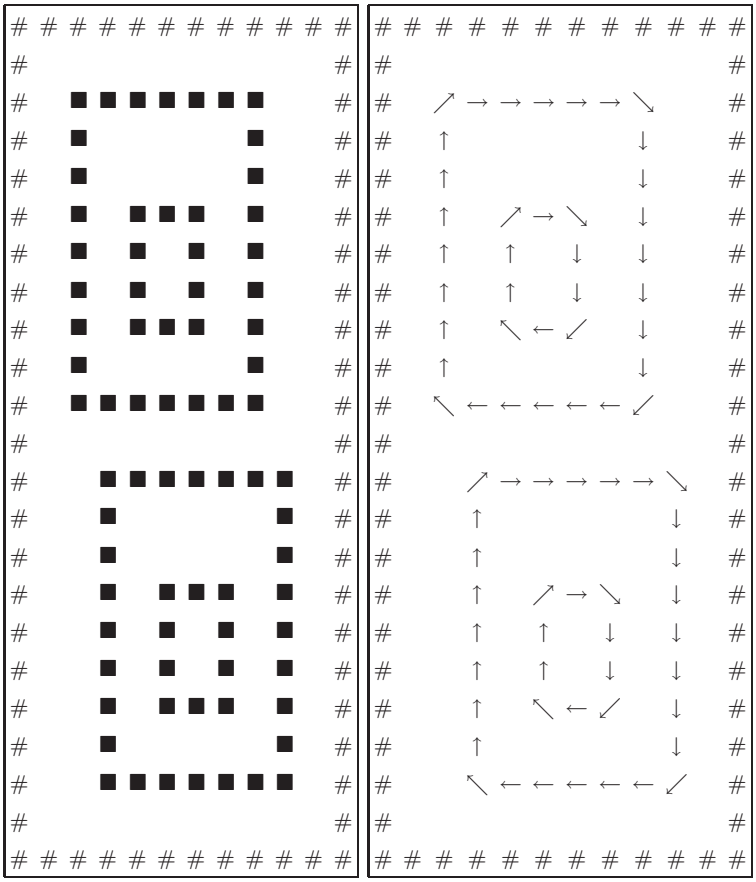


Fig. 2. A picture of the Chinese box's language (left), and the corresponding picture with the tile alphabet (right).

$\{\blacksquare, \square\}$. An example picture is shown in Figure 2, left.

Since in such pictures almost any combination of terminal elements may occur as a tile, a definition based on terminal tiles would be too permissive.

For example, a subpicture containing the forbidden pattern  could be tiled.

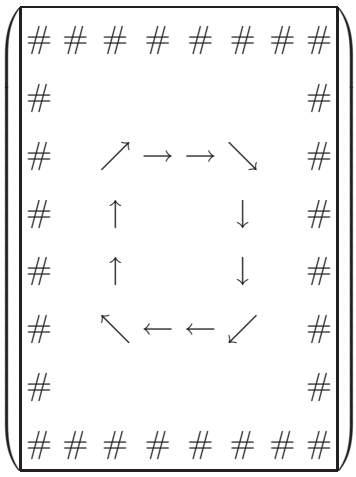
The remedy is to use a larger tile alphabet, in order to avoid confusion between pixels of two adjacent frames. We use the tile alphabet:

$$\Gamma = \{\nearrow, \rightarrow, \searrow, \downarrow, \swarrow, \leftarrow, \nwarrow, \uparrow, \square\}$$

The perimeter of a frame is thus encoded by clockwise turning arrows, as shown in the right part of Figure 2.

Instead of listing the rather large tile set, it saves effort and improves readability to specify it by prototypes, that is by exhibiting one (or more) pictures of alphabet Γ, such that all and only the permitted tiles are present. One such picture is the following:

$$\Theta = B_{2,2}$$



The projection π simply maps any arrow onto a black square, and a blank onto a blank. Applying the projection to the right part of Figure 2, one obtains the initial picture.

Notice that the choice of the tile alphabet is not obvious and some guidelines and practical criteria are needed for picture specification by TS.

3 Picture Recognition as SAT problem

Syntactic pattern recognition is another name for the classical formal language problem of *syntax analysis* (or *parsing*). For Tiling Systems, this corresponds to the following question: given a TS T and a picture p, if $p \in \mathcal{L}(T)$, then how does T generate p? Quite naturally, having an efficient tool to solve this problem for Tiling Systems would be a first and necessary step to use such formalism in practice: one could define a picture language by means of a TS, and then could use the parsing algorithm to *recognize* a given input picture and its structure. Notice that the "syntactical structure" of picture p is essentially representable as a picture q over the tile alphabet Γ, such that the projection $\pi(q)$ equals p. In fact, as Γ is a richer alphabet than Σ, its patterns display a structure, which is to some extent erased by projection π. To put it differently, the same pixel in different positions may have different "meanings".

Unfortunately, we know from the theory that the problem of parsing Tiling Systems is NP-complete (see [14] and [15] [3]), because some typical and well-known NP-complete problems are easily translated into a TS parsing problem. In fact, it is known that pictures of a TS language can represent configurations of a Turing Machine (see e.g. [5], proof of undecidability of emptiness problem for TS languages).

[3] In that paper Tiling Systems are called *Homomorphisms of Local Lattice Languages*.

On the other hand, a number of useful problems, though NP-complete (or worse), are tackled in practice, for example some classical verification problems such as *Model Checking* [16]. Here we focus on the *Boolean Satisfiability Problem* (SAT for short), one of the best known NP-complete problems. An instance of the problem is a propositional logic formula (i.e. a Boolean expression on a set of propositional variables), and the question is: given the formula, is there some assignment of *true* and *false* values to the variables that will make it true? CNF-SAT is a classical variant of the problem, in which the formula is in *Conjunctive Normal Form* (CNF), i.e. an AND of clauses, where a clause is an OR of simple or negated propositional variables. In recent years, the availability of quite effective tools to solve the CNF-SAT (e.g. the recent MiniSat [17]), based on the seminal Davis-Putnam-Logemann-Loveland (DPLL) algorithm [18], spurred the creation of many verification tools, among which we cite the bounded model checking component of the NuSMV verifier [19].

Familiarity with SAT-solving tools gave us the idea to encode the TS parsing problem into SAT.

3.1 The Encoding

Consider a Tiling System $T = (\Sigma, \Gamma, \Theta, \pi)$. Essentially, given an input picture $p \in \Sigma^{*,*}$, i.e. a picture made of symbols taken from Σ, the parsing problem consists in finding a picture $q \in \Gamma^{*,*}$, having the same size as p, such that:

(1) its projection coincides with p, i.e. $\pi(q) = p$;
(2) its tiling is compatible with Θ, i.e. $B_{2,2}(q) \subseteq \Theta$.

If both conditions are true, then, and only then, $p \in \mathcal{L}(T)$. Notice that q is not necessarily unique.

Notice that this is an instance of typical inverse mathematical problems, which are often computationally challenging.

Now, to encode the problem into SAT, we represent the pixels of the picture q as SAT's propositional variables. In practice, this means that the statement $q(i, j) = a$ (i.e. pixel (i, j) of q contains the symbol a), becomes a propositional variable of the SAT problem.

As an example, consider the Chinese boxes picture represented in the left part of Figure 2. If in p at position (2,2) there is a symbol ■, then necessarily the corresponding pixel in q (i.e. before the projection) must be an arrow (but we do not know which one). Once the encoding is complete, we may ask the SAT-solver to "guess" such picture q. If the SAT-solver succeeds, then the picture p is accepted (and q is returned); elsewise, $p \notin \mathcal{L}(T)$.

To fully exploit the SAT encoding, we also accept partial input pictures. This means that some of p's pixels may be left unspecified (conventionally marked by a "don't care" symbol '?'). With a slight abuse of notation, we say that the inverse projection of a "don't care" symbol in p is Γ, i.e. $\pi^{-1}(?) = \Gamma$. Informally, this means that we do not know anything about that pixel, so any symbol of the tile alphabet could be in q at that position.

The encoding consists of expressing the afore mentioned Conditions 1) and 2), as propositional logic formulas.

Condition 1) states that q must be "compatible" with p, i.e. such that $\pi(q) = p$: [4]

$$F_1 := \bigwedge_{(i,j)\in[(1,1)..|p|]} \begin{pmatrix} \bigvee_{a\in\pi^{-1}(p(i,j))} q(i,j) = a \\ \wedge \\ \underset{a'\in\Gamma}{\mathrm{OnlyOne}}\Big(q(i,j) = a'\Big) \end{pmatrix}$$

F_1 depends only on p and on the projection π. The first AND is used to span the whole picture, while the inner $\mathrm{OnlyOne}$ operator is used to check that one and only one value taken from the alphabet Γ is assigned to q at a given position.

Condition 2) considers the tile set Θ: to accept p, every tile used in q must be a member of Θ.

$$F_2 := \bigwedge_{(i,j)\in[(1,1)..|p|]} \bigvee_{t\in\Theta} \bigwedge_{h,k\in[0,1]} \begin{pmatrix} q(i+h, j+k) \\ = \\ t(h+1, k+1) \end{pmatrix}$$

As in the previous formula, the first AND spans the whole picture. Then, the inner OR states that one of the tiles in Θ must be present at a given position.

The TS-recognition problem is then encoded as the propositional formula $F_1 \wedge F_2$.

Example 2. Consider Figure 2, left side. For brevity, we consider just a small part, the one corresponding to the subpicture at position $(2, 1)$ having size $(2, 3)$, as shown next: 

[4] For conciseness, we introduce the $\mathrm{OnlyOne}$ Boolean function, with any number of arguments. Informally, $\mathrm{OnlyOne}$ is true if, and only if, exactly one of its arguments is true. E.g. $\mathrm{OnlyOne}\,(A, B, C) \iff (A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C)$.

The resulting F_1 formula, limited to the pixels of the previous subpicture, is:

$$
\begin{array}{c}
\ldots \wedge q(2,1) = \square \wedge \\
\left(\begin{array}{c}
q(2,2) = \nearrow \vee q(2,2) = \rightarrow \vee \\
q(2,2) = \searrow \vee q(2,2) = \uparrow \vee \\
q(2,2) = \swarrow \vee q(2,2) = \leftarrow \vee \\
q(2,2) = \nwarrow \vee q(2,2) = \downarrow
\end{array} \right) \wedge \\
\mathrm{OnlyOne} \left(\begin{array}{c}
q(2,2) = \nearrow, q(2,2) = \rightarrow, \\
q(2,2) = \searrow, q(2,2) = \uparrow, \\
q(2,2) = \swarrow, q(2,2) = \leftarrow, \\
q(2,2) = \nwarrow, q(2,2) = \downarrow, \\
q(2,2) = \square
\end{array} \right) \wedge
\end{array}
$$

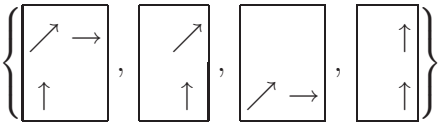
$$
\begin{array}{c}
\left(\begin{array}{c}
q(2,3) = \nearrow \vee q(2,3) = \rightarrow \\
\vee q(2,3) = \searrow \vee q(2,3) = \uparrow \vee \\
q(2,3) = \swarrow \vee q(2,3) = \leftarrow \vee \\
q(2,3) = \nwarrow \vee q(2,3) = \downarrow
\end{array} \right) \wedge \\
\mathrm{OnlyOne} \left(\begin{array}{c}
q(2,3) = \nearrow, q(2,3) = \rightarrow, \\
q(2,3) = \searrow, q(2,3) = \uparrow, \\
q(2,3) = \swarrow, q(2,3) = \leftarrow, \\
q(2,3) = \nwarrow, q(2,3) = \downarrow, \\
q(2,3) = \square
\end{array} \right) \wedge
\end{array}
$$

$$
\begin{array}{c}
q(3,1) = \square \wedge \\
\left(\begin{array}{c}
q(3,2) = \nearrow \vee q(3,2) = \rightarrow \\
\vee q(3,2) = \searrow \vee q(3,2) = \uparrow \vee \\
q(3,2) = \swarrow \vee q(3,2) = \leftarrow \vee \\
q(3,2) = \nwarrow \vee q(3,2) = \downarrow
\end{array} \right) \wedge
\end{array}
$$

$$\mathrm{OnlyOne}\left(\begin{array}{c} q(3,2) = \nearrow, q(3,2) = \rightarrow, \\ q(3,2) = \searrow, q(3,2) = \uparrow, \\ q(3,2) = \swarrow, q(3,2) = \leftarrow, \\ q(3,2) = \nwarrow, q(3,2) = \downarrow, \\ q(3,2) = \square \end{array}\right) \wedge$$
$$q(3,3) = \square \wedge \ldots$$

In the input picture at position $(2,1)$ there is a blank, and $\pi(\square) = \square$, therefore $q(2,1)$ must be a blank. At position $(2,2)$ there is a $\square$, so $q(2,2)$ can be any of the arrows. The other pixels are analogously translated.

Next consider the F_2 corresponding to the previous subpicture. For brevity we restrict consideration to just a few of the tiles in Θ, namely:



We obtain the formula F_2:

$$\ldots \wedge \left(\begin{array}{c} q(2,1) = \nearrow \wedge q(2,2) = \rightarrow \wedge \\ q(3,1) = \uparrow \wedge q(3,2) = \square \\ \vee \\ q(2,1) = \square \wedge q(2,2) = \nearrow \wedge \\ q(3,1) = \square \wedge q(3,2) = \uparrow \\ \vee \\ q(2,1) = \square \wedge q(2,2) = \square \wedge \\ q(3,1) = \nearrow \wedge q(3,2) = \rightarrow \\ \vee \\ q(2,1) = \square \wedge q(2,2) = \uparrow \wedge \\ q(3,1) = \square \wedge q(3,2) = \uparrow \\ \vee \ldots \end{array}\right) \wedge \qquad (1)$$

$$\left(\begin{array}{c} q(2,2) = \nearrow \wedge q(2,3) = \rightarrow \wedge \\ q(3,2) = \uparrow \wedge q(3,3) = \square \\ \vee \\ q(2,2) = \square \wedge q(2,3) = \nearrow \wedge \\ q(3,2) = \square \wedge q(3,3) = \uparrow \\ \vee \\ q(2,2) = \square \wedge q(2,3) = \square \wedge \\ q(3,2) = \nearrow \wedge q(3,3) = \rightarrow \\ \vee \\ q(2,2) = \square \wedge q(2,3) = \uparrow \wedge \\ q(3,2) = \square \wedge q(3,3) = \uparrow \\ \vee \ldots \end{array}\right) \wedge \ldots \tag{2}$$

Subformula 1 considers a tile placed at position $(2,1)$, so the examined pixels are those at positions $(2,1)$, $(2,2)$, $(3,1)$, $(3,2)$. The first row of the formula represents the first tile, the second row of the formula represents the second tile, and so on. Analogously, Subformula 2 considers tiles placed at position $(2,2)$.

3.2 The Tool

The TS parsing tool accepts as its input a file containing a TS specification T, and a picture p (or just its size). As output, the tool offers a picture such that its projection coincides with p, if p is in the language of T, nothing otherwise. When the user provides just the picture size, the tool returns a picture having the same size and such that its projection belongs to the language of T (this modality is also called *picture generation*). This is analogous of parsing a rectangle entirely made of "don't care" symbols, and is a useful test to see if the TS language contains any picture having a fixed size.

Internally, the tool is composed of a *Core* module, and a couple of utility modules: *Input*, and *Back-parser*.

The Core module accepts as input the TS description (i.e. the tile set Θ, and the projection π, both expressed as lists), and the picture p to be parsed. As anticipated, the input picture may be completely defined, or contain "don't care" symbols. As output, the core module produces the formula $F_1 \wedge F_2$ presented in the previous

section, in *CNF-DIMACS* format [5] .

The Input module provides convenient utilities to define tile sets, such as usual set operations (union, intersection, complement, difference), and the $B_{2,2}$ operation to extract tiles from a given prototype.

After formula generation, the TS parsing tool calls the SAT-solver, which supplies as output either UNSAT (i.e. the formula contains a contradiction, hence p is not in the language of the given TS), or a suitable assignment to propositional variables. Next the Back-parsing module is called after the SAT-solver to parse the assignment to propositional variables, in order to obtain a picture q such that $\pi(q) = p$. The Back-parsing module finally generates q as a LaTeX table.

The tool is written in the Scheme programming language. This choice makes it possible for the experimenter to use the logical constructs of Scheme to create tile sets in a compact and easy way. We defer to the next section the discussion on tile sets definition.

4 Experiments

4.1 Practical tile specification

Next we present a short gallery of TS definitions of interesting picture languages. As we proceed we state and illustrate the following techniques for specifying tile sets.

(1) *Explicit:* the first and most simple technique is the one used in [5], in which tile sets are exhaustively listed. Usually, this technique defines *minimal* tile sets, i.e. sets containing all the needed tiles, and only them. Its major shortcoming resides in its error proneness: an explicit tile set is hard to read by humans, because of its very size. It is easy to write a wrong tile in it, but hard to spot it. Moreover, modifications of the sets are for the same reason hard.

(2) *By prototype:* a natural way of expressing tile sets is by using a prototype picture. In this case, we automatically derive the tile set applying the $B_{2,2}$ operator to the prototype. It is used for instance to define the Chinese boxes TS (Figure 2). This technique represents tile sets in a very readable and easily modifiable way. Its main shortcoming is that for complex Tiling Systems it

[5] CNF-DIMACS is a *de facto* standard input format for SAT-solvers, and was defined by the *Center for Discrete Mathematics and Theoretical Computer Science*, a collaborative project of Rutgers University, Princeton University, AT&T Labs, Bell Labs, Telcordia Technologies, and NEC Labs.

is not trivial (and sometimes not possible at all) to find one small example picture containing all the needed tiles.

(3) *Set operations:* tile sets are indeed sets, therefore it is straightforward to use union, intersection, and complement on them. For instance, paper [5] uses this technique together with exhaustive listing. Others [15] specify most examples *ex negativo*: they start from a small explicit tile set and consider all the tiles of the same alphabet which are *not* in the given set. Sometimes the needed tile set is so large that is more convenient to build it by giving forbidden tiles. As far as shortcomings are concerned, tile sets defined through this technique are sometimes non-minimal, because contain *unproductive* tiles. These are tiles that cannot be used with the others. For instance, consider the example of Figure 2. If we add the tile $\boxed{\begin{matrix}\# \uparrow \\ \# \downarrow\end{matrix}}$ to the tile set, its language is unchanged, because this tile is unproductive.

Though not a major shortcoming, the presence of unproductive tiles impacts on the performance of the parser. Indeed, our tool must consider all the given tiles, hence the resulting propositional formula is larger than needed. Moreover, it may happen that the tiles intended as unproductive turn out the other way, causing subtle changes to the intended picture. Our tool helps to reveal this kind of errors: the user can use the ability to generate random pictures as a test bench to spot unwanted tiles.

(4) *Logic expressions:* another classical way of expressing sets is by using logic expressions to implicitly constrain their content (e.g. the well known notation $\{x \in \mathbb{N} : \exists y \in \mathbb{N}(x = 2y)\}$ defines the set of all even natural numbers). The same technique can be used for tile sets: logic constraints can predicate on relationship between pixels used in the tiles. This approach is very compact and expressive, but usually defines non-minimal tile sets. Its shortcomings are the same as the previous case.

Our tool supports all four techniques. In practice, the four techniques are often combined together: sometimes it is convenient to start from a prototype picture, and then to add further tiles by union with the tiles coming from another kind of expression, e.g. logic or explicit. We show some such combinations in the next examples.

4.2 Examples

In this section we present some example of tool usage, because we think they illustrate both what kind of languages are definable by Tiling Systems, and practical techniques for expressing tiles. The examples are chosen to cover all techniques. Moreover, we show how we can exploit the tool capability of completing partial pictures.

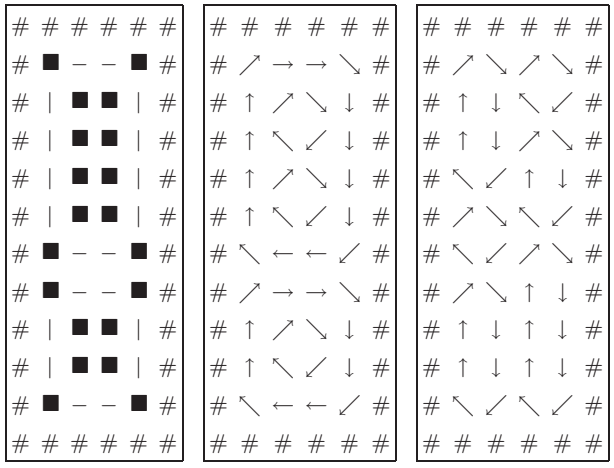


Fig. 3. A picture of the Chinese boxes (no b.g.) language (left), its parsing output (center), and a picture with the same size completely generated by the tool (right).

Example 3. Chinese boxes (no background)

Figure 3 presents a variant of the Chinese boxes introduced in Figure 2. In this variant, we drop background symbols. This means that we define a language consisting of rectangles (also multiple ones) which either are $\boxed{\begin{smallmatrix}\blacksquare & \blacksquare \\ \blacksquare & \blacksquare\end{smallmatrix}}$ or contain other rectangles. To make the actual rectangles more visible, we use different symbols to mark their upper, lower ($-$), left, and right ($|$) pixels, while we keep the $\blacksquare$ symbol for corners.

First, we introduce the tile set. It encodes the idea that the perimeter of a box corresponds to a closed rectangular path: staring from e.g. the top-left corner, one may either continue straight ahead (symbol $\rightarrow$), or choose to make a down turn (symbol $\searrow$). Then, it is possible to continue in that direction ($\downarrow$), or to make a left turn ($\swarrow$). And so on, going back to the top-left corner. This means that the path is a rectangle. We use Technique 4 to define its tile set: the constraint is an OR of implications.

$$\Theta = \left\{ \boxed{\begin{smallmatrix} i & j \\ k & l \end{smallmatrix}} : \begin{array}{c} i \in \{\rightarrow, \nearrow\} \Rightarrow j \in \{\rightarrow, \searrow\} \\ \vee \\ j \in \{\downarrow, \searrow\} \Rightarrow l \in \{\downarrow, \swarrow\} \\ \vee \\ l \in \{\leftarrow, \swarrow\} \Rightarrow k \in \{\leftarrow, \nwarrow\} \\ \vee \\ k \in \{\uparrow, \nwarrow\} \Rightarrow i \in \{\uparrow, \nearrow\} \end{array} \right\}$$

Next, we have the projection, where the symbols used to encode turns become black squares, while oriented lines become simple lines.

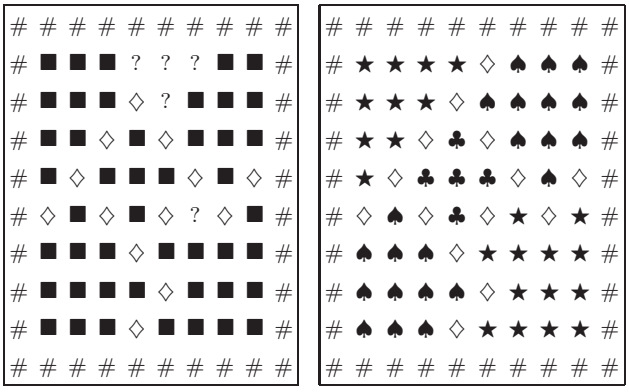


Fig. 4. A picture of the Three colors language (left), and the corresponding picture with the tile alphabet (right).

$\pi(x) =$ ■, if $x \in \{\nearrow, \searrow, \nwarrow, \swarrow\}$;

$\pi(x) = -$, if $x \in \{\rightarrow, \leftarrow\}$;

$\pi(x) = |$, if $x \in \{\uparrow, \downarrow\}$.

Figure 3, left side, shows a picture, consisting of a top vertical box containing two smaller boxes, and a bottom box containing a single smaller box. The picture is correctly recognized, and the central part of the figure reports the output of the tool.

To offer a simple example of picture generation, we used the tool by entering only the picture size, $(4, 10)$ in this case. The right part of Figure 3 contains the output of the tool.

Example 4. Three colors map coloring

This example is a simple encoding of the well-known problem of *map three-coloring*. As input, we give a monochromatic map in which states are regions filled with ■ and bordered by ♢ (see Figure 4, left). Such a picture is said *three-colorable* if every state can be assigned a color from the set $\mathrm{Colors} = \{\spadesuit, \bigstar, \clubsuit\}$, so that its neighbors have different colors. If this is the case, then the tool should offer as output a suitably colored map. We found the original version of this example in [15]. [6]

To make the tile set simpler, we assume boundaries run at 45 degrees slopes (i.e. like NE-SW, or NW-SE). Such boundaries are easily encodable as single tiles, without requiring a more complex Γ alphabet. The example in [15] uses 3×3 tiles, but our tool only offers 2×2 tiles, coherently with [5]. Theoretically, it is possible to

[6] It is worth to notice that [15]'s version contains some errors, which we easily detected thanks to our tool. In fact, writing tile sets by hand is an error prone task. Using a tool such as the one presented here is a great aid for a TS specifier.

translate 3×3 tiles into 2×2 tiles on a richer Γ alphabet, but this discussion is outside the scope of the present paper.

Nonetheless, the tile set is quite large and it is easier to describe it as the complement of the union of three simpler tile sets: $\Theta = \overline{\Theta_1 \cup \Theta_2 \cup \Theta_3}$ (i.e. by Technique 3).

The first tile set describes the erroneous situation of having the same color across a border (this uses Technique 1):

$$\Theta_1 = \left\{ \boxed{\diamondsuit\ \spadesuit \\ \spadesuit\ \diamondsuit}, \boxed{\diamondsuit\ \bigstar \\ \bigstar\ \diamondsuit}, \boxed{\diamondsuit\ \clubsuit \\ \clubsuit\ \diamondsuit}, \boxed{\spadesuit\ \diamondsuit \\ \diamondsuit\ \spadesuit}, \boxed{\bigstar\ \diamondsuit \\ \diamondsuit\ \bigstar}, \boxed{\clubsuit\ \diamondsuit \\ \diamondsuit\ \clubsuit} \right\}$$

The second tile set encodes straight N-S or E-W borders. The third tile set considers the color used within a state. In both cases we are using Technique 4.

$$\Theta_2 = \left\{ \boxed{i\ \ j \\ k\ \ l} : \begin{array}{c} i = \diamondsuit = j \\ \vee \\ k = \diamondsuit = l \\ \vee \\ i = \diamondsuit = k \\ \vee \\ j = \diamondsuit = l \end{array} \right\};$$

$$\Theta_3 = \left\{ \boxed{i\ \ j \\ k\ \ l} : \begin{array}{c} i, j \in \text{Colors} \wedge i \neq j \\ \vee \\ k, l \in \text{Colors} \wedge k \neq l \\ \vee \\ i, k \in \text{Colors} \wedge i \neq k \\ \vee \\ j, l \in \text{Colors} \wedge j \neq l \end{array} \right\}$$

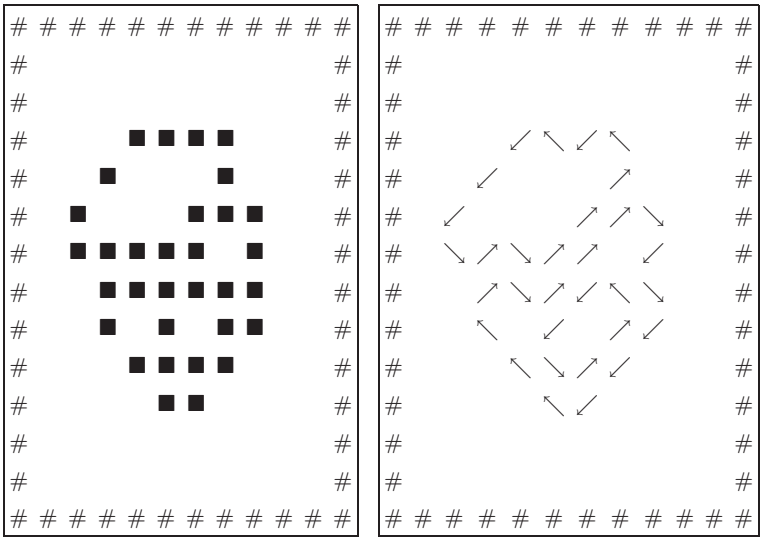


Fig. 5. A picture of the Contour lines language (left), and the corresponding picture with the tile alphabet (right).

Notice that Θ is built upon the complement of the three auxiliary tile sets, hence they are expressed in a *negative* form (e.g. Θ_2 accepts only straight N-S or E-W borders).

The projection is quite simple: colors become black squares, while the bordering symbol $\diamondsuit$ remains the same:

$\pi(c) = \blacksquare$ for $c \in \mathrm{Colors}$;

$\pi(\diamondsuit) = \diamondsuit$.

It is worth mentioning that the actual tile set input to the tool is expressed almost exactly as above, the only difference being purely syntactic (Scheme uses a full-parenthesized prefix notation, and tile sets are implemented as lists).

Figure 4, left shows an example input picture. Some of the pixels are marked as "don't cares" (e.g. the one at position $(2, 5)$). "Don't cares" stand for uncertainty on the pixel values: for example this could represent a situation in which some of the bits of the pixel encoding contain a checksum to check its correctness. In this case, if we receive a pixel with a wrong checksum, we mark its content as "?", since we are not sure about it. Figure 4 (right) presents the resulting output of the parsing tool: the input picture is correctly colored and completed (e.g. the pixel at position $(1, 5)$ is marked as a boundary).

Example 5. Contour lines on a topographic map

The last example is that of Contour lines. Consider a topographic map in which only the contour lines are shown. The pixels belonging to a contour line are marked

as ■ symbols, while the other symbols are blanks (see Figure 5, left).

As tile symbols, we use the four arrows $\nearrow, \searrow, \nwarrow, \swarrow$, and ideally we consider a kind of "flux" entering and exiting at each corner of a given pixel. Intuitively, the reader may think it like a water stream which follows the arrows, going in or out of a given tile. For example, a pixel containing a $\searrow$ represents a flux coming from its upper-left corner and going to its lower-right corner.

The tile set considers each path as a flux, and takes care of the balance between the flux "entering" the center of a tile (i.e. the lower-right corner of its upper-left pixel) and that "exiting" it. The first part of the union consider such balance: if a flux enters the center of the tile, then it must leave it; if a flux leaves, then it must be coming from somewhere. The second part of the union considers the cases in which the entering fluxes are two (obviously they cannot be more than that), and fluxes which avoid the central point (the last two tiles).

$$\Theta = \left\{ \begin{bmatrix} i & j \\ k & l \end{bmatrix} : \begin{array}{c} i = \searrow \Rightarrow \\ \mathrm{OnlyOne}(j = \nearrow, k = \swarrow, l = \searrow) \\ \vee \\ i = \nwarrow \Rightarrow \\ \mathrm{OnlyOne}(j = \swarrow, k = \nearrow, l = \nwarrow) \\ \vee \\ j = \swarrow \Rightarrow \\ \mathrm{OnlyOne}(i = \nwarrow, k = \swarrow, l = \searrow) \\ \vee \ldots \end{array} \right\}$$

$$\cup \left\{ \begin{bmatrix} \searrow & \nearrow \\ \swarrow & \nwarrow \end{bmatrix}, \begin{bmatrix} \nwarrow & \swarrow \\ \nearrow & \searrow \end{bmatrix}, \begin{bmatrix} \searrow & \nearrow \\ \swarrow & \nwarrow \end{bmatrix}, \begin{bmatrix} \nwarrow & \swarrow \\ \nearrow & \searrow \end{bmatrix}, \begin{bmatrix} \nearrow & \nwarrow \\ \searrow & \swarrow \end{bmatrix}, \begin{bmatrix} \swarrow & \searrow \\ \nwarrow & \nearrow \end{bmatrix} \right\}$$

The tile set is defined through a combination of Techniques 1, 3, and 4.

The projection is the usual, mapping arrows to back squares and leaving blanks alone.

$\pi(x) =$ ■, if $x \in \{\nearrow, \searrow, \nwarrow, \swarrow\}$;

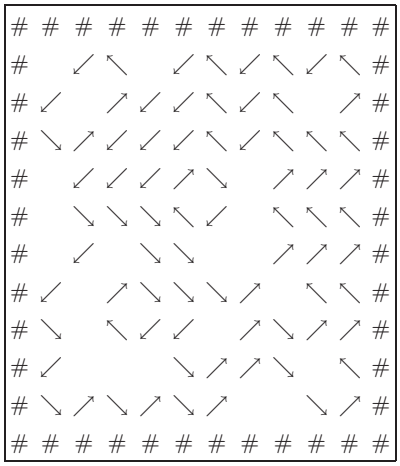


Fig. 6. A generated tile alphabet picture of the Contour lines language.

$\pi(\square) = \square$.

An example picture is presented in Figure 5, left. The tool recognizes it and its output, given as non-intersecting fluxes, is presented on the right side.

Last, Figure 6 shows the output of the generation of a 10×10 picture.

4.3 Experimental results and performance

In this section we consider the current implementation of the prototype tool, its performance, and how to improve it.

The main steps the tool performs are the following:

(1) *Formula generation:* the tool reads its input and generates the corresponding propositional formula $F = F_1 \wedge F_2$ as presented in Section 3. As far as computation complexity is concerned, the execution time of this step (and the resulting formula size) is linear with respect to the input picture size (i.e. the number of its pixels).
(2) *CNF translation:* F is then translated into conjunctive normal form using an external simplification tool (SAT2CNF, part of the Alloy Analyzer [7]). Its output is a CNF-DIMACS formula F'. Time complexity: polynomial with respect to the input picture size. [8] SAT2CNF preserves satisfiability of the original formula, and keeps the transformation polynomial by adding fresh propositional letters.
(3) *SAT-solver:* the SAT-solver MiniSat [17] is called on input F'. The output of the SAT-solver is then read by the Back-parsing module to produce the output

[7] http://alloy.mit.edu
[8] We do not know the details of algorithm used by SAT2CNF.

	Execution time (s)		
Input picture	Formula generation	CNF translation	SAT-solver
Figure 2, left	0.78	0.86	0.11
Figure 3, left	4.67	3.62	7.70
Figure 4, left	0.60	0.39	0.18
Figure 5, left	1.40	1.51	0.36
Figure 6	1.40	1.24	2.10

Fig. 7. Parsing times of the example pictures.

picture. Time complexity: exponential (see e.g. [20]).

The second step is provisional in the prototype and not really necessary, being the encoding for Tiling Systems always the same. This means that we could write, instead of formulas F_1 and F_2 of Section 3, directly a conjunctive normal form formula. We have opted for ease of implementation, and left this optimization for future consideration.

The three steps communicate by using text files (in DIMACS format) that can be quite large. This impacts especially on the speed of steps 1 and 2.

The chosen implementation language, Scheme, is not very fast. Scheme is good for rapid prototyping, but we would obtain better performance for the first step by using a lower-level language, such as C.

Surprisingly, in spite of the above limitations, the overall performance of the tool is acceptable, and allowed us to experiment with reasonably sized pictures. First, we report in Figure 7 a table with the execution time of the tool on the examples presented in this paper. [9]

Notice that the dispensable CNF translation step is quite expensive, while the SAT-solver step, despite theoretically being the most expensive, is in many cases the least.

Moreover, notice that the case of Figure 3 is dominant, despite the input picture being quite small, compared to the others. The main reason of this behavior is the expression of the tile set. In fact, the implemented tile sets appear simple, and works correctly, but is much bigger than needed. In particular, the formulas used to constraint the content of the tile set consider only two pixels of a given tile, and leave unspecified the other two. For this reason, this tile set contains a big number

[9] We ran the experiments on a PC equipped with AMD Athlon 64 X2 4600+, 2 Gb RAM, Linux OS; SAT-solver: Minisat, v. 2.0 beta.

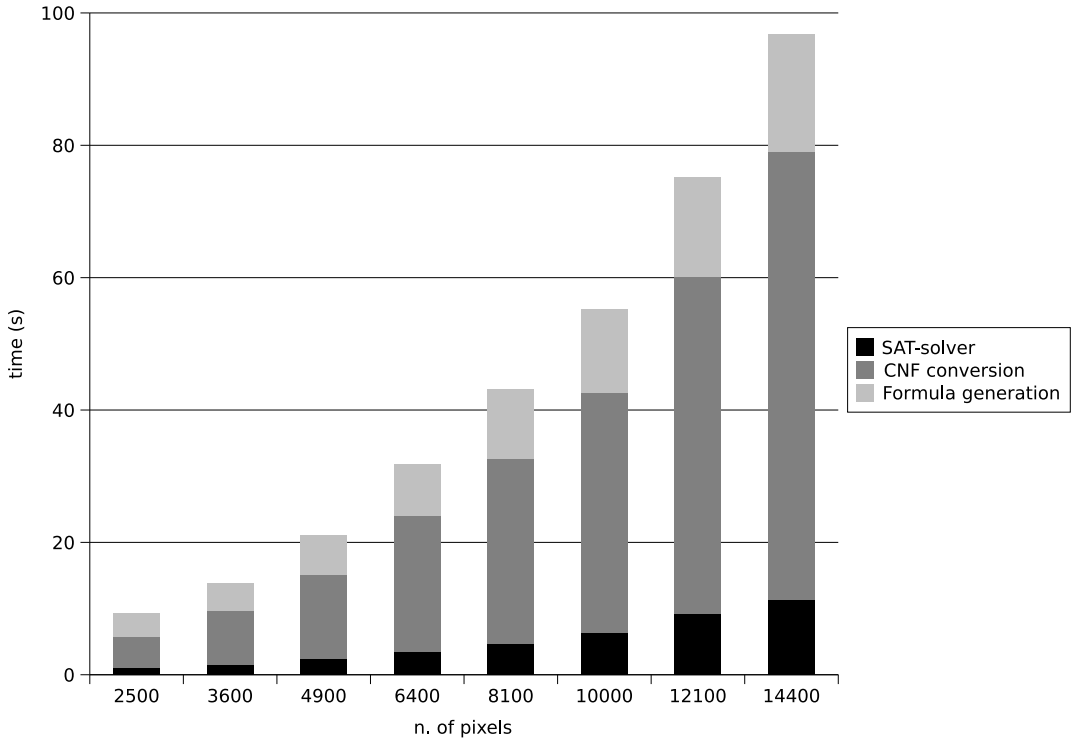


Fig. 8. Parsing times of square homogeneous pictures, TS of Figure 2

of tiles (2866), many of which unproductive, such as $\begin{smallmatrix}\leftarrow & \leftarrow \\ \downarrow & \downarrow\end{smallmatrix}$. An equivalent, but more compact and efficient for the tool, tile set could be written by using a more complex set expression in which only the usable tiles are taken into account.

Next, we report the use of the tiling systems of Figures 2 and 4 to recognize bigger square pictures having sizes from 50×50 to 120×120. All the input pictures considered are homogeneously made of a single symbol (i.e. $\square$ for Figure 2 and $\blacksquare$ for Figure 4). The measurements are reported in Figures 8 and 9, respectively. Figure 10 summarizes the memory consumption of the SAT-solver. As before, they are quite encouraging, as the most expensive step is the dispensable CNF-translation. The SAT-solving step for the 120×120 picture is in both cases less than 20 seconds. For the TS of Figure 2, we also tried two less trivial pictures of side 120: one containing 5 rectangles in random positions, and a totally random (incorrect) one. As expected, formula generation and CNF translation steps times are similar for all pictures with side 120. On the other hand, the SAT-solver step took a bit more time (about 30%) for the first case, and much less time for the second case (about 1/4 of the time reported in Figure 8). The latter result can be putatively be explained because the formula obtained from a random picture is likely to be falsified by just checking a few clauses, corresponding to a small part of the picture. Such findings justify our choice to present measurements just for homogeneous pictures.

To conclude, experimental evidence is that the step that in principle should be dom-

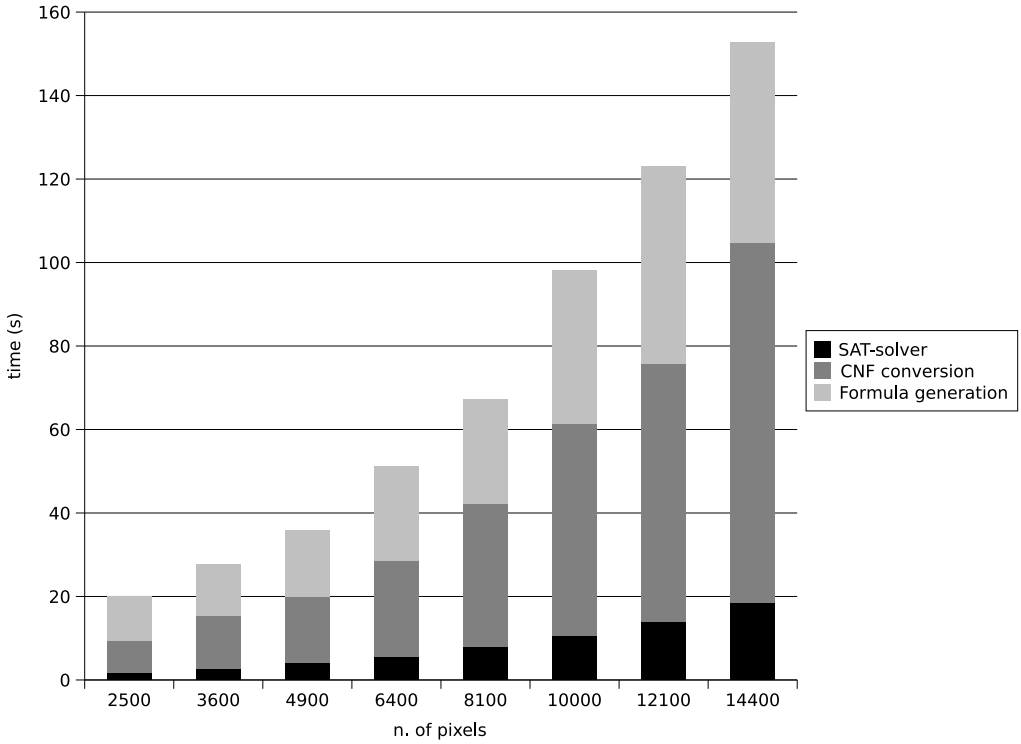


Fig. 9. Parsing times of square pictures, TS of Figure 4

SAT-solver: memory occupation (Mb)

n. of pixels	TS of Figure 2	TS of Figure 4
2500	29.9	38.0
3600	43.0	53.5
4900	59.8	74.2
6400	77.0	97.4
8100	95.2	122.1
10000	125.3	154.3
12100	142.0	182.2
14400	172.2	218.3

Fig. 10. Memory consumption of the SAT-solver

inant, SAT-solver is in reality almost negligible, with respect to the other two steps. In future developments, one of them, CNF translation can be entirely suppressed, while the other, Formula generation, has much opportunity for optimization. This step is also easily parallelizable: different processors could be allocated to the generation of sub-formulas corresponding to different parts of the input picture. Of course experiments on larger and more complex pictures will be needed.

5 Conclusions

In our opinion this work offers two interesting contributions to research on syntactic pattern recognition.

First we gave evidence that Tiling Systems can be used in practice to specify non-trivial classes of pictures. We argued that different notations and combination thereof are convenient for defining the tiles to be used.

Second, we showed how to encode the pattern recognition or picture parsing problem as a SAT solving one, and designed a practical tool [10], capitalizing on the mature algorithmic know-how of off-the-shelf SAT solvers.

An attractive, unusual feature of our pattern recognizer is the ability to generate a picture or to complete a partial one. This can be exploited for interpolation, or to perform error corrections on missing pixels. At the extreme, this ability allows the user to generate pictures of prescribed size, in order to test the correctness of the pattern family definition.

We intend to experiment with the tool for picture interpolation or inpainting. Moreover, we envision the possibility of using the tool to recognize pictures in which pixels or blocks of pixels are encoded together with checksum bits. In this case, the tool could be used to recognize a picture that contain transmission errors: wrong checksums are treated as “don’t cares”.

At present our technique has not been experimented on very large and complicated pictures. We plan to improve its practical efficiency and compare it with currently used approaches.

As far as tool developments, we intend to improve the performance of our tool, as explained before, in order to process larger pictures and more complex patterns. Moreover comparative experiments using other existing SAT-solvers may reveal better heuristics for tiling problems.

Finally we hope that by making it practical to experiment with syntactic definitions based on tiling systems, the validity of this approach will be assessed, possibly in combination with methods based on other approaches, such as grammars or statistics.

[10] Available from the first author: http://www.elet.polimi.it/upload/pradella.

Acknowledgments

We thank Philippe Salembier for his comments and application suggestion.

References

[1] E. Tanaka, Theoretical aspects of syntactic pattern recognition, Pattern Recognition 28 (7) (1995) 1053–1061.

[2] K. S. Fu, Syntactic Pattern Recognition and Applications, Prentice-Hall, Englewoods Cliffs, 1982.

[3] N. G. Bourbakis, Special issue on languages for image processing and pattern recognition, Pattern Recognition 32 (2) (1999) 253.

[4] K. Morita, K. Imai, Uniquely parsable array grammars for generating and parsing connected patterns, Pattern Recognition 32 (2) (1999) 269–276.

[5] D. Giammarresi, A. Restivo, Two-Dimensional Languages, in: A. Salomaa, G. Rozenberg (Eds.), Handbook of Formal Languages, Vol. 3, Beyond Words, Springer-Verlag, Berlin, 1997, pp. 215–267.

[6] R. Siromoney, K. Subramanian, V. Dare, D. Thomas, Some results on picture languages, Pattern Recognition 32 (2) (1999) 295–304.

[7] A. Cherubini, S. Crespi Reghizzi, M. Pradella, P. San Pietro, Picture Languages: Tiling Systems versus Tile Rewriting Grammars, Theoretical Computer Science 356 (1-2) (2006) 90–103.

[8] C. Allauzen, B. Durand, Tiling problems, in: E. Borger, E. Gradel (Eds.), The classical decision problem, Springer-Verlag, 1997.

[9] M. Cohen, J. Shade, S. Hiller, O. Deussen, Wang tiles for image and texture generation, ACM Transactions on Graphics (Siggraph'03 conference proceedings) 22 (3) (2003) 287–294.

[10] K. S. Dersanambika, K. Krithivasan, C. Martín-Vide, K. G. Subramanian, Local and recognizable hexagonal picture languages., International Journal of Pattern Recognition and Artificial Intelligence 19 (7) (2005) 853–871.

[11] K. Inoue, A. Nakamura, Some properties of two-dimensional on-line tessellation acceptors, Information Sciences 13 (1977) 95–121.

[12] M. Blum, C. Hewitt, Automata on a two-dimensional tape, in: IEEE Symposium on Switching and Automata Theory, 1967, pp. 155–160.

[13] D. Giammarresi, A. Restivo, Recognizable picture languages, International Journal Pattern Recognition and Artificial Intelligence 6 (2-3) (1992) 241–256, special Issue on *Parallel Image Processing*.

[14] H. Lewis, Complexity of solvable cases of the decision problem for predicate calculus, in: Proc. 19th Symposium on Foundations of Computer Science, 1978, pp. 35–47.

[15] K. Lindgren, C. Moore, M. Nordahl, Complexity of two-dimensional patterns, Journal of Statistical Physics 91 (5-6) (1998) 909–951.

[16] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 1999.

[17] N. Eén, N. Sörensson, An extensible SAT-solver, in: SAT 2003: Sixth International Conference on Theory and Applications of Satisfiability Testing, 2003.

[18] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Communications of the ACM 5 (7) (1962) 394–397.

[19] A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without BDDs, Lecture Notes in Computer Science 1579 (1999) 193–207.

[20] M. Alekhnovich, E. Hirsch, D. Itsykson, Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas, in: Proc. of ICALP 2004, Vol. 3142 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 84–96.