SG-CLIPS: A SYSTEM TO SUPPORT THE AUTOMATIC GENERATION OF DESIGNS FROM GRAMMARS

SHENG-FEN CHIEN, MAGD DONIA
School of Architecture, Carnegie Mellon University. Pittsburgh, PA, USA.

AND

JAMES D. SNYDER, WEN-JAW TSAI

Abstract. SG-CLIPS is a computer tool that supports the automatic generation of designs from a predefined set of grammar rules that encapsulate the composition principles of a certain style of design. It is an open system that accepts any grammar conforming to the conventions described in this paper. We demonstrate the system through an example and discuss the relation to shape grammars and limitations.

1. Introduction

SG-CLIPS is a computer tool that supports the automatic generation of designs from a set of design grammar rules. These rules are defined using the programming environment of the tool, and encapsulate the composition principles of a certain style of design, such as the Bungalows of Buffalo grammar (Downing & Flemming, 1981). The idea behind SG-CLIPS is to provide a running system, with a friendly interface, that allows for the exploration of designs, which can be generated by a certain grammar. It is an open system in the sense that it accepts any grammar that conforms to the constraints and conventions of the programming environment described in this paper.

The system consists of a programming environment, an inference engine and a graphical user interface. The programming environment provides a set of shape primitives (such as point, line and polygon), for representing design objects. These primitives are the elements on which the design rules could match and create further elements which are added to the design representation. The design rules themselves are defined as regular CLIPS (Giarratano, 1993) rules with some special considerations to facilitate rule matching and take advantage of the SG-CLIPS environment. CLIPS is used as the inference engine which determines and executes the rules that match a
certain pattern of shapes to create a new design state. The graphical user interface, implemented in wxCLIPS (Smart, 1994), provides a friendly interface from which the user can load in a grammar file and explore the designs that can be generated by that grammar.

In the following sections, the design of the SG-CLIPS environment is explained in detail, and an illustrative example is shown to demonstrate how it can be used. It is followed by a discussion of the possible uses and limitations of the system and some reflections on how it relates to Shape Grammars (Stiny, 1980).

2. Design Object Representation

In SG-CLIPS, there are 7 different kinds of shape primitives, namely: DesignState, LabeledPoint, Shape, Point, LineSegment, Line, and Polygon. They are represented as objects having the relations shown in Figure 1.

![Figure 1. SG-CLIPS shape primitives in an OMT (Rumbaugh, 1991) diagram](image)

A Shape is an abstract class that defines a generic interface for geometric entities. The classes Point, LineSegment, Line, and Polygon provide concrete implementations for the interface defined by Shape. A LabeledPoint is not classified as a Shape, because it is not a geometric entity and does not have the same interface as of other geometric entities. Shape provides a default implementation to minimize the labor involved in developing new classes. For instance, the default implementation of the “draw” method in Shape draws all its “subParts” such that all the sub-parts of an inheriting class are automatically drawn. A DesignState is simply a collection of Shapes and LabeledPoints which is the result of a rule application.

3. Rule Base and Inference Engine

A rule consists of a left-hand-side (LHS) and a right-hand-side (RHS). A LHS defines the conditions for rule matching using a collection of predicates. The matching is performed by the inference engine. All objects, instances of classes, asserted as facts are subjected to the matching mechanism, regardless of their actual object composition. For instance, matching is performed on the LineSegments composing a Polygon, as well
as the Polygon itself. A RHS defines the actions that should be performed when its corresponding LHS matching is successful.

The matching is performed automatically by the inference engine whenever facts are asserted. A successful matching of the LHS of a rule is then placed in a rule execution (firing) agenda along with the matched objects, ready to be fired. The firing of a rule is the actual execution of the RHS instructions. When objects (facts) are removed from the design representation, the rules in the firing agenda that matched on these retracted objects are retracted from the agenda as well. The placement or order of rules in the firing agenda is controlled by the conflict resolution strategy built into the inference engine.

The inference engine continuously executes the RHS of the next available rule in the agenda until the agenda is empty. This implies that there must be an initial fact, or facts, to "motivate" the inference engine. This initial fact triggers more rules to assert more facts which, in turn, trigger more rules and so on. A grammar in SG-CLIPS is a collection of rules and initial facts such that the firing of the grammar rules performs the design tasks encoded in the grammar.

4. SG-CLIPS Rules

There are a few general rules-of-thumb to follow when writing SG-CLIPS rules, which are discussed in this section.

\[ A. \ (\text{defrule} \ \text{rule}\_\text{signature} \ (\text{initial-fact}) \ \\
\Rightarrow \ (\text{assert} \ (\text{classname} \ (\text{attribute-value})+) \ (n\text{Tree})) \ ) \]

\[ B. \ (\text{defrule} \ \text{rule}\_\text{signature} \ \\
\Rightarrow \ (\text{assert} \ (\text{classname} \ (\text{attribute-value})+) \ (n\text{Tree})) \ ) \]

\[ \text{ds} \ \rightarrow \ (\text{object} \ \text{is-a} \ \text{DesignState} \ \text{state} \ ?id) \ \\
\text{?obj} \ ightarrow \ (\text{object} \ \text{is-a} \ \text{Name}) \ (\text{attribute-name}) \ \\
\text{attribute-value} \ \text{symbol} \ \& \ ?\text{predicate} \end{LHS} \ \\
\text{and} \ \text{of} \ \text{LHS} \ \\
\Rightarrow \ (\text{object-pattern-match-delay} \ \\
\text{bind} \ ?\text{shape} \ \text{newly created shape} \)
\text{create new design state} \ \\
\text{bind} \ ?\text{newds} \ \text{send} \ ?\text{ds} \ \text{duplicateWOShape} ?\text{ip}) \ \\
\text{send} \ ?\text{newds} \ \text{addShapes} \ ?\text{shape} \ \\
\text{update tree} \ \\
\text{augmentTree} \ ?\text{newds} \ \\
\text{end} \ \text{of} \ \text{delay} \ \\
\text{end} \ \text{of} \ \text{rule} \]

\text{Figure 2. Sample SG-CLIPS rule templates} \text{1}

There must be an initial rule such as \text{defrule init} shown in Figure 2-A. In addition, a general SG-CLIPS rule should use the template shown in Figure 2-B. Each rule is based on an active DesignState and creates a new DesignState using its RHS instructions. After the creation of a new DesignState, this new Design State should be added to the SG-CLIPS

\text{1 Names in angle brackets should be substituted accordingly. A + sign signifies one or more of the preceding construct. The rest of the code should be interpreted literally. The construct (n\text{Tree}) is used to initialize the graphical display of SG-CLIPS.}
graphical display by issuing (augmentTree ?<newds>). The RHS part of a rule should be enclosed in the construct (object-pattern-match-delay ...) to delay CLIPS pattern matching until all the RHS instructions in the rule are executed. A detailed example of a SG-CLIPS rule is provided later.

5. User Interface and Rule Execution

The SG-CLIPS user interface provides a graphical display of DesignStates and control of rule executions. At the present time, graphical rule construction is not provided. A graphical rule construction requires an elaborate model of rules and automatic code generation. It's an entire new domain of research that has yet to be investigated.

When SG-CLIPS starts, it loads basic design primitive objects and user interface code automatically. The user then loads any domain-specific grammar by specifying the file containing the rules that define the grammar. SG-CLIPS organizes DesignStates as rectangles in a tree structure (Figure 3). The rectangles are color coded: blue for an explored state, green for selected, black for un-explored, and red for the current activated state. At any given time, there can be only one activated state, which is the DesignState scheduled for the next design exploration.

![Figure 3. SG-CLIPS application window](image)

Rule executions are controlled by DesignState explorations. When the "Next State" command is selected, all possible "direct" descendent child states of the current activated DesignState are explored. This is done by bringing all the rules which can be applied to the activated state to the top of the agenda and executed sequentially. Therefore, users can explore the design space according to their own will instead of relying solely on the CLIPS inference engine. Each DesignState can be magnified and displayed...
in a separate window with full details. In addition, user can save a selected state to a file and load the file back into SG-CLIPS for later exploration.

6. An Illustrative Example

In this section, we demonstrate how to use the SG-CLIPS environment through an example. We begin by illustrating how to encode a shape rule into an SG-CLIPS rule and proceed to explain how to execute these rules in the environment. The example is taken from shape grammar rules of Buffalo bungalows. Particularly, we discuss the encoding of rule #8 shown in Figure 4 below. This rule generates, between the kitchen and one other primary but non-dining space, an internal hall and a staircase that borders the rear exterior wall. In Figure 4, label r marks the rear of the building; labels t and h** are used to control the order of rule applications and labels k and d denote the kitchen and dining space, respectively.

![Figure 4. Buffalo bungalows shape grammar rule #8](Redrawn from Downing & Flemming, 1981)

6.1. ENCODING A SHAPE RULE

To begin encoding, we need to identify key features in the LHS shapes. First, we know that the label r should exist for the rule to be applicable. Next, there is a kitchen space (represented in SG-CLIPS as a polygon with a labeled point k) that is adjacent to the rear of the building; and there is a non-dining space (a polygon not labeled d). Finally, this non-dining space is aligned with the kitchen space along the rear wall; and these two spaces are adjacent to each other. These features are encoded in the LHS part of the rule shown below. The actions to create the internal hall and staircase form the RHS of the same rule (starting at the "=>" symbol). Diagrams on the right side of the sample code illustrate particular shapes under consideration within the adjacent portion of the code.

```clips
(defrule rule8
  :comment "find a design state with label t and identity rear of the building"
  ==(stateLabel `" Label t and identity rear of the building"
  ==(label k (state 0 (x 0.0) (y 0.0) (z 0.0))
  ==(ds `" Label d`
  ==(ds `" Label t and identity rear of the building"
  ==(label r `(send ?ds isMember ?stateLabel))
  ==(ip `" Label p`
  ==(ip `" Label r`
  ==(ip `" Label d and label r are member of the state label")

```
6.2. EXECUTING RULES

The encoded shape grammar rules and an initial shape should be stored in files. These files can be loaded into the SG-CLIPS environment through the “Load Grammar...” command. Once the grammar is loaded, the system displays the initial state and makes it active. When an active state contains patterns specified in a particular rule, this rule becomes active and can be executed. The “Next States” command allows users to apply rules that are applicable to the active state.

Figure 5. DesignStates before (A) and after (B) the execution of rule r8

Figure 5-A shows a DesignState that contains patterns specified in the rule (r8) illustrated in the previous section. If this state is active, the rule (r8) can be applied to generate a new DesignState shown in Figure 5-B. To make a state active, users simply highlight it in the SG-CLIPS application window (Figure 3) and select the “Activate State” command.

7. Discussion

SG-CLIPS was conceived to allow the exploration of designs generated from grammars. Once a grammar is defined according to the system programming conventions, the user can very easily generate many different designs in a relatively short amount of time. The system supports the
management of design alternatives generated, thus enabling the user to backtrack and explore design alternatives in parallel and examine the rules responsible for the generation of each design state.

Given that premise, the system can serve as a useful tool for architecture and design students to encourage the exploration and development of design grammars. In addition, by exploring the generated designs and evaluating them against the style of design captured by the grammar, the system can be used as a test environment to verify and refine design grammars.

7.1. LIMITATIONS

SG-CLIPS has a number of limitations. Although all shape primitives have a three-dimensional representation, their operations, such as the ones used to determine the intersection of two shapes or co-linearity of two lines, have been designed and tested for two-dimensional cases only. The user interface is only capable of displaying two-dimensional objects, and the grammars implemented and tested with the system so far have all been two-dimensional grammars as well.

The system is also limited by the performance of CLIPS which can be significantly slower and consume more memory than iterative programming environments, such as C++ (Stroustrup, 1991). wxCLIPS, the graphical user interface environment, is relatively limited in its capabilities and support for sophisticated user interaction and updates. Our experience indicates that wxCLIPS is difficult to scale beyond what has been already implemented in SG-CLIPS.

7.2. RELATION TO SHAPE GRAMMARS

Shape Grammars are a set of rules that define a language by which shapes can be generated that share some common characteristics, or conform to a certain style of design. Each rule matches on a certain pattern and replaces it with another; enabling other rules to match and so on until a pattern is created where no more rules can match. Given that the grammar is constructed correctly, the resulting pattern or design has to conform to the style whose rules of composition are encoded in the grammar. A shape in a grammar is defined as a set of maximal lines where each line is unique in the sense that it cannot be combined with another line in the shape to form another single line (Krishnamurti, 1992).

The specification of a shape grammar or implementation of a shape grammar interpreter is difficult for several reasons. First, the computational complexity makes it difficult to devise a method of designer interaction that is intuitive and timely. Second, rule application in shape grammars is
ambiguously defined. To date, most shape grammars have relied on people’s innate ability to do shape recognition (constraint satisfaction). Using rule one from Figure 6, the English equivalent might be “replace a rectangle with the same rectangle and a rectangle inscribed inside of it.” However, the complexity arises from the fact that it is not easy to say what a rectangle is. In this case, the line lengths and incident angle constraints would all have to be specified to ensure that this rule applied only to rectangles. Essentially, rules must classify shapes before they can be matched and applied. Designing a grammar is complicated by the fact that no robust computational shape grammar systems have been implemented.

![Figure 6. A simple shape grammar (Redrawn from Stiny, 1980)](image)

Because we dealt with the time constraints of a single semester course (see Acknowledgments) we decided not to implement a pure shape grammar interpreter. Our approach was motivated by a simple observation that shape rules attempt to classify shapes during the matching process. The first activity we did was to extract and name all the predicates and constraints implied by a particular shape grammar. Then we collated these predicates together and found a common set that could be used by all grammars. Next we developed an object model (see Design Object Representation) that incorporated all the predicates we discovered but pre-classified the shapes we knew we would expect. We found these shape primitives sufficient to implement the grammars presented in the course because a rich set of operations and predicates were defined for the shapes. For example, operations like union and difference of shapes are provided as well as range of capabilities found in analytic geometry. In addition, any constraints placed on a set of shapes can be easily represented by combining the defined predicates.

It is important to point out that this implementation strategy is not a shape grammar or a grammar system at all. Rather, it captures the effect of a grammar system. In other words, the resulting shapes are the same, but the

---

2 Even though Gips & Stiny (1980) define rule application, without the incorporation of the constraint satisfaction mechanism being specified, rule application remains computationally ambiguous.

3 Stiny frequently mentions classifications of shapes in his papers. Under the shape grammar formalism, shapes can be classified by the constraints they satisfy. However, he never specifies how these constraints are specified or integrated into the formalism.
path traversed to get there is drastically different. This difference is rooted in how the left-hand and right-hand side of the rules are represented and implemented. In true grammar systems, the rules can be applied in either a top-down direction (derivation) or a bottom-up direction (parsing). Theoretically, the same concerns and capabilities are present in pure shape grammars, however, this idea has not yet been completed tested due to the fact that shape grammar implementations are sparse, where as, string grammar implementations (being the most successful example in applying parsing and derivation) have been studied for decades.

Acknowledgments

SG-CLIPS was conceived during the “Architectural Languages” course taught by Professor Ulrich Flemming at the School of Architecture, Carnegie Mellon University in the Spring semester of 1995. We would like to thank Michael Cumming, Michael Shealey, Bige Tuncer, Emre Yavuz, and Ye Zhang for their contributions in establishing the object structure that underlay the SG-CLIPS environment, as well as Professor Ulrich Flemming for his comments and encouragements.

References


* This has been proven to be true only if the rules are reversible. See Krishnamurti & Stouffs (1993) for complete details.