

CIDER: A Component-Based Toolkit for Creating Smart Diagram Environments

Anthony R. Jansen, Kim Marriott, and Bernd Meyer

School of Computer Science and Software Engineering

Monash University, Victoria 3800, Australia

E-mail: {tonyj,marriott,berndm}@mail.csse.monash.edu.au

Abstract—Smart Diagram Environments (SDEs) are software applications that use structured diagrams to provide a natural visual interface that behaves as if the computer “understands” the diagram. Unfortunately, despite their potential usefulness, SDEs are not easy to build. We present CIDER a Java toolkit for building SDEs which greatly simplifies this task. CIDER is a generic component-based system which is designed to be easily embedded in Java applications. It provides automatic interpretation of diagrams as they are constructed and manipulated, structure preserving manipulation, and a powerful transformation system for specifying diagram manipulations and user interaction. CIDER’s main innovation is its component-based approach to SDE development which provides substantially increased architectural flexibility to the application programmer.

I. INTRODUCTION

Smart Diagram Environments (SDEs) are software applications that use structured diagrams to provide a natural visual interface [8]. Such an environment behaves as if the computer “understands” the diagram, for example by providing manipulation that takes into account the diagram’s structure and its intended semantics. SDEs are useful in a wide variety of applications such as high-level query languages for spatial information systems, CAD systems, diagrammatic theorem provers, and on-line education. They are particularly useful in domains where well-established diagrammatic notations are in use, such as UML in software engineering or circuit diagrams in electronic engineering.

As a simple example, consider an SDE that is designed to teach students about Finite State Automata (FSAs). It would allow the students to create and modify FSA diagrams, it would visually demonstrate how to construct a FSA from a regular expression and whether a particular input string belongs to the language of the FSA, and visually demonstrate how to construct a deterministic FSA from a non-deterministic FSA, and how to minimise it. It might also provide automatic layout and beautification (i.e. “pretty printing”).

Unlike a standard graphics editor such as `xfig` the SDE would understand the structure of FSAs and automatically interpret the FSA diagram drawn by the user. In principle, the user constructs these diagrams through basic drawing operations from primitive graphical objects, such as lines and circles. Of course, in an ideal system additional support and high-level, composite graphical objects may be available where this simplifies the interaction. As an example consider the FSA shown in Figure 1. The SDE should identify the

possible transitions and recognise that the two concentric circles on the right of the diagram and the text, S3, that lies within them forms a single unit which represents a final state. Later if one of the circles or the text is moved during user editing the other components of the state should move with it as well as any transitions to or from the state. The user should be warned if they have drawn an invalid FSA, for instance if there is an unlabelled transition or no start state.

SDEs are potentially very useful, but unfortunately not easy to build. Luckily, there are a number of common functions that every SDE has to provide and implementing these subtasks can be supported by generic software tools. In particular, most SDEs have to provide modelling of the diagram components, interpretation, structure preserving manipulation, visual transformation, and layout/beautification. A logical approach is to build a generic SDE system which provides these capabilities but which is then customised for a particular diagrammatic language based on a formal high-level specification of that language. This is the idea behind the tools DIAGEN [10], GENED [4], RECOPLA [9], PENGUINS [3] and GENGED [1]. However, given the potential usefulness of SDEs, it is interesting to ask why these system are not commonly used (except by those who developed them). We believe a major reason is the lack of architectural flexibility of the above systems. They all severely restrict how the SDE application programmer must write the remainder of the application and also the application GUI. The problem is that they provide a large single system which “wraps around” the main application code. This appears to be the wrong approach, as the main application code and not the user-interface should take center stage. The RECOPLA system explored a different approach by completely separating the front-end from the application and coupling them via inter-process communication. However, such a loose coupling has its own problems when, for example, code needs to be shared

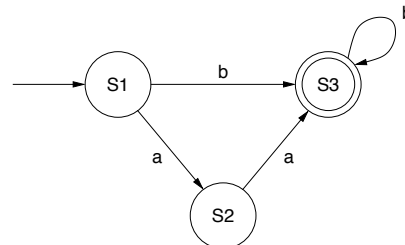


Fig. 1. A simple finite state automaton.

between the front-end and the application. A second reason why previous SDE systems are not widely used is that they are each limited in their capabilities (although in different ways). For instance, only PENGUINS provides (very) limited diagram beautification, while it does not provide transformations. This problem is compounded by their underlying architecture which makes them difficult to extend. Most of these systems are based on the idea of free editing and it is very difficult or impossible to extend them for specialized forms of interaction or to customize the GUI in other forms. However, this is required when building any real world application. Interaction specifications, such as those in GENGED and DIAGEN, do not solve the problem as they can only restrict the admissible interactions but not extend the basic modes of interaction. Finally, existing SDE construction systems are not compatible with commonly used toolkits for GUI development and so require significant investment by an application programmer.

Here we describe a new toolkit, CIDER, for building SDEs which overcomes the limitations of these previous toolkits. CIDER provides the following capabilities:

- Automatic interpretation of diagrams as they are constructed and modified based on Constraint Multiset Grammar (CMG) specifications of diagrammatic syntax.
- Structure preserving manipulation. This is provided by an application specific constraint solver.
- A powerful transformation mechanism for specifying diagram manipulations and user interactions that is fully integrated with the incremental parser, allowing transformations to be couched in terms of high-level diagram components yet understood in terms of low-level displayable components.
- Efficiency is achieved by compiling grammars, transformations and constraint code.
- Although CIDER does not yet provide diagram layout or beautification its architecture allows the application to provide this.

However probably the most important innovation of CIDER is its component-based architecture. Rather than wrapping around application specific modules, CIDER consists of generic components which the application wraps around. Such a component-based architecture provides much more flexibility for the application programmer and it does not impose any restrictions on how the main application is programmed. It means for instance, that the application programmer has complete control over the interface, can choose not to use some components of the system and even to extend the toolkit's capabilities by providing additional components, such as a customized constraint solver.

To further facilitate CIDER's integration into real-world applications, Java was chosen as the implementation language as Java is one of the most commonly used languages for implementing applications that require graphical user interfaces. Other benefits of Java include its platform independence, and its support for component-based programming as well as a whole culture of component exchange and re-use. Another

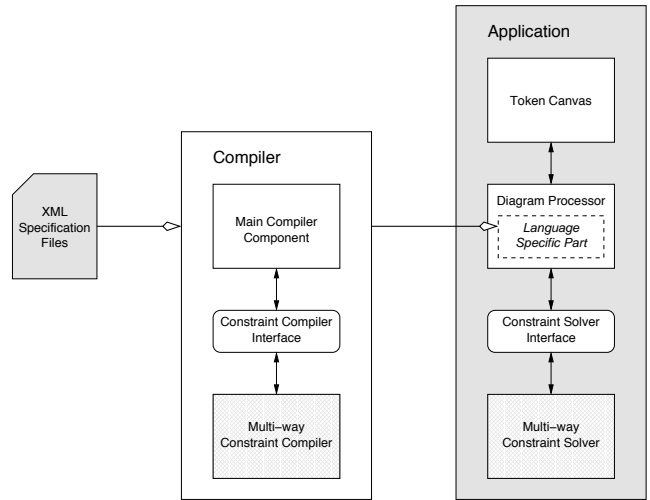


Fig. 2. The components that make up the CIDER toolkit.

feature of CIDER is that it allows the diagrammatic language grammar to be specified using XML.

Designing CIDER in this way was not straightforward since we needed to provide interfaces that allowed CIDER to be used in a very flexible fashion but were as compatible as possible with Java's existing GUI packages and no more complex to use. One benefit of this design is a very low learning threshold for the application designer when starting to use CIDER.

The components that make up the CIDER toolkit are shown in Figure 2, which also indicates how these components are used in the creation of an application. The white boxes indicate components of CIDER, the cross-hatched boxes indicate optional components that can be tailored to extend the capabilities of the toolkit itself. The shading indicates components that must be created by the application developer. The double-headed arrows indicate which CIDER components interact with which other components. In the following sections we will take a more comprehensive look at the capabilities of the CIDER toolkit, in particular looking at the roles played by each of the toolkit's components.

II. DIAGRAM INTERPRETATION

As in virtually all object-oriented graphic programs it is natural to represent a diagram in a SDE as a collection of primitive graphical objects such as circles, lines and text. Most object-oriented graphics toolkits and libraries provide a kind of *object canvas* to support this approach. Such a canvas allows graphic objects to be added, deleted or modified and takes care of applying the appropriate display updates. Unfortunately (and somewhat surprisingly) Java does not provide such a canvas. Thus, the CIDER toolkit provides its own *Token Canvas* component to provide this functionality. Application developers can extend it with new application specific graphical object types. Given a set of primitive graphic objects in an object canvas, one basic SDE functionality is the ability to interpret them; that is the SDE must be able to create a parse forest for the diagram on the canvas [8]. This is the task of CIDER's core component, the *Diagram Processor*.

Of course to parse, the Diagram Processor needs a specification of the diagrammatic notation syntax. CIDER uses a variant of Constraint Multiset Grammar (CMG) formalism, a kind of attributed multi-set grammar [5]. A specification has two parts: symbol definitions and production rules.

Symbols have geometric and semantic attributes. Each type is either *terminal* or *non-terminal*. Terminal symbols correspond to the primitive graphic objects in the diagram while non-terminal symbols are more complex objects built-up from these. For example in the case of FSAs, the definition

```
Circle (terminal)
  radius:real, mid:point;
```

specifies a terminal symbol type `Circle` with `radius` and `mid` as real valued attributes, while

```
Transition (non-terminal)
  from:string, to:string, label:string;
```

specifies a non-terminal symbol type with three string-valued attributes that identify the state the transition comes from, the state it goes to and the transition label.

CIDER extends standard attribute grammar formalisms by providing an *abstract* symbol type. This allows the user to specify inheritance/generalisation relationships between symbols and means that production rules can be polymorphic in a symbol type. It is analogous to the subtype polymorphism provided in object-oriented programming languages. For example, a FSA may contain normal states and final states, which can have transitions between them. For the purpose of defining a transition, whether a state is normal or final is irrelevant. We simply want to define each transition as coming from one state, and going to another state. This is achieved by defining an abstract symbol type called `State`

```
State (abstract)
  radius:real, mid:point, label:string;
```

and two other symbol types representing normal and final states, respectively, that inherit from it

```
NormalState (non-terminal) : State;
FinalState (non-terminal) : State;
```

Both `NormalState` and `FinalState` inherit the attributes of the `State` symbol type. They could also include additional attributes if needed, although this is not required in this example. It is not possible to create an instance of an abstract symbol type directly, only its subtypes can be instantiated. Using inheritance, production rules that are common to `NormalState` and `FinalState` can be factored out, referring to both state types collectively as instances of `State`. CIDER supports multiple levels of abstraction (that is, `State` could itself extend another abstract symbol type).

Production rules are used to specify how non-terminal symbols are composed of other symbols. One of the simplest productions for FSAs is that defining an arc `r` to be an arrow `a` with a textual label `t`. It specifies that the arrow and text have the same mid-point and that the arc has the same start, end and mid-point as the arrow and label as the text:

```
r:Arc ::= a:Arrow, t:Text
  where (
    a.mid == t.mid
  ) {
    r.start := a.start;
    r.mid := a.mid;
    r.end := a.end;
    r.label := t.label;
  }
```

During parsing this production is used to generate new instances of the `Arc` symbol type from `Arrow` and `Text` symbols which have a common mid-point. The attributes of the new `Arc` symbol are set by the expressions given between the braces. We call `a` and `t` the right-hand side (RHS) symbols and `r` the left-hand side (LHS) symbol.

More generally production rules are of the form

$$R_1, \dots, R_m ::= S_1, \dots, S_n \text{ where } C \text{ using } E$$

A rule is applied by testing if there is an assignment of symbols in the parse forest to S_1, \dots, S_n , whose attributes satisfy the conditions C . If the evaluation is successful, the LHS symbols R_1, \dots, R_m are produced. The attribute values of these symbols are computed from the attribute values of the RHS symbols according to the expressions in E .

The ability to produce multiple output symbols means that the CIDER toolkit provides *unrestricted* (type 0) CMGs [7]. It is relatively rare that the full power of type 0 grammars is required. Context-free and context-sensitive rules are much more common. For convenience existentially quantified symbols can be used to specify context-sensitive rules. An example is the following production, which defines a transition from one state to another state, where the states are existentially quantified so that they are not reduced when the production is applied.

```
t:Transition ::= a:Arc
  where (
    exists s1:State, s2:State
      where (
        OnState(a.end, s2.mid, s2.radius) &&
        OnState(a.start, s1.mid, s1.radius)
      )
  ) {
    t.from := s1.label;
    t.to := s2.label;
    t.label := a.label;
  }
```

The CIDER toolkit provides the standard Java comparison operations on RHS symbol attributes ($=$, \neq , $<$, $>$, \leq and \geq) for use as conditions in the production rules. However, for flexibility the conditions can also include calls to any pre-existing or user-defined Java method. Similarly the CIDER toolkit allows attribute values of the LHS symbol to be set by pre-existing or user-defined Java methods.

One potential problem with incrementally parsing diagrams in an SDE is *ambiguity* when recognizing sub-diagrams in a partial diagram. For instance, a diagram of a FSA with two concentric circles on a piece of text may be recognized as a final state or as a normal state and an additional circle. In static parsing such ambiguity may be resolved by considering

all possible maximal interpretations and then choosing the interpretation which explains all elements in the diagram. In the context of an SDE such ambiguity is bad for at least two reasons. First, it makes parsing considerably more complex since multiple parse trees must be considered. Second, and even more importantly, it makes it extremely difficult to provide immediate feedback about what has been recognized so far during diagram construction since the ambiguity may only be resolvable once the complete diagram has been drawn.

For this reason productions in the CIDER toolkit can also use conditions which test for the non-existence of a symbol. These are called *negative context tests* and may be nested. In order to avoid any ambiguity when recognising normal and final states, we can add a negative context test to the production for normal states specifying that, for a normal state to be formed, there must not exist a second circle which shares its mid-point with the circle defined among the production’s RHS symbols:

```
n:NormalState ::= c1:Circle, t:Text
  where (
    c1.mid == t.mid &&
    not exists c2:Circle
      where (c2.mid == c1.mid)
  ) {
    n.mid := c1.mid;
    n.radius := c1.radius;
    n.label := t.label;
  }
```

III. DYNAMIC PROCESSING

While constructing the parse forest from a static diagram is important, the point of an SDE is to allow diagrams to be manipulated, with users able to add, remove and modify diagram components. To deal with this, CIDER uses incremental parsing and constraint solving to ensure that the diagram’s interpretation is kept consistent with the display.

A. Diagram Interpretation

The symbols in the Token Canvas can be modified by three types of action: new terminal symbols can be added, terminal symbols can be removed, and attributes of symbols in the canvas can be modified. These actions generate events which are handled by the Diagram Processor. If an application developer wants to perform further processing they can register event handlers to listen for these events.

To help maintain consistency of the parse forest, the Diagram Processor keeps the production instance used to create each non-terminal symbol along with the symbol. When any of the three actions take place, the Diagram Processor goes through each of the non-terminal symbols in the parse forest and removes all symbols for which the associated production instance is now invalid. This can be for two reasons: the condition no longer holds or else one of the input symbols or existentially quantified symbols is no longer valid.

For obvious efficiency reasons, production conditions are only rechecked if the action may have invalidated them. Consider, for example, the production to produce a `NormalState` symbol (defined in a previous section), which has the following two constraints.

```
(1) c1.mid == t.mid
(2) not exists c2:Circle
     where (c2.mid == c1.mid)
```

If the action in question was the addition of a new `Circle` symbol to the diagram, then condition (2) would need to be rechecked, as it may no longer be true that there are no circles in the diagram that meet the condition `c2.mid == c1.mid`. If the action was that the `mid` attribute of a `Text` symbol was modified, and that symbol was the one used in the production (t), then condition (1) would need to be rechecked as it depends on `t.mid`. If the `mid` attribute of a `Circle` symbol was modified, and that symbol was the one used in the production (c1), then both conditions (1) and (2) would need to be rechecked as they both depend on `c1.mid`. If the `mid` attribute of a `Circle` symbol was modified, but that symbol was not used by the production, then condition (2) will still need to be rechecked.

If rechecking discovers that the production is no longer valid, the production and its LHS symbols need to be removed from the parse forest. Before any symbol is removed, it must be checked to see if it is used by another production further up in the parse forest, and if this is the case, then that production and its LHS symbols must be removed first. As these changes could percolate in the parse forest it is obvious that even a single action can result in large changes to the parse forest.

If an attribute value is updated other attribute values which depend upon it are first updated before the checking described above is performed. To understand why such updating is necessary consider a `NormalState` symbol which is produced from a `Circle` and a `Text` symbol. If the `label` attribute of the `Text` symbol is modified, the Diagram Processor will then also need to update the `label` attribute of the `NormalState` symbol so that it matches the new `label` value of the `Text` symbol. This must be done to keep the parse forest consistent, since the production rule for the `NormalState` symbol stipulates that its `label` attribute must be equal to the `label` attribute of the `Text` symbol used to produce it.

The Diagram Processor uses a built-in one-way constraint solver to automatically update attribute values. More exactly, each assignment in the expression component of a production rule adds a one-way constraint of form $x = f(y_1, \dots, y_n)$ to this constraint solver. The one-way constraint solver then updates the value of attribute x whenever one of the attributes y_1, \dots, y_n changes value, using the function f to compute the new value. Such one-way constraints have been widely used in GUIs and can be extremely efficiently solved [2]. Note that, as their name suggests, one-way constraints are directional: if x changes its value, the values of y_1, \dots, y_n will not be updated.

After the update is performed the Diagram Processor executes all productions that have now become applicable. New symbols added to forest can, of course, invalidate negative context conditions in other productions that then need to be reset. Updating stops when the parse forest is once again in a stable, consistent, and maximal state. Note that termination of this process obviously requires a stratified grammar and that CIDER performs a stratification check at compile-time.

B. Structure-preserving Manipulation

When performing direct manipulation it would be desirable for the SDE to behave as if it understood the structure of the diagram and to preserve this structure during editing. For example, consider again the production for a `NormalState` symbol. If the `Text` component of a `NormalState` symbol is moved (that is, its `mid` attribute is modified), the production becomes invalid as the `Text` and `Circle` RHS symbols will no longer have the same `mid-point` value. It would be desirable to bind the `mid` attribute of the `Text` and `Circle` symbols so that when one is modified, the other is automatically modified to the same value, allowing the `Circle` and `Text` to move together and thus ensuring that the `NormalState` production remains valid.

Unfortunately, general one-way constraint solving is not powerful enough to provide this feature. The problem is that one-way constraints are inherently directional, but in the above example we would like the `Circle` to be updated if the `Text` is moved and vice versa. To provide this we need a *multi-way constraint solver*.

The CIDER toolkit does not actually include a multi-way constraint solver of its own, but rather provides a generic interface that allows the application developer to choose whatever multi-way constraint solver is most appropriate for use with a given diagrammatic language since there is no best choice. Of course, the use of an external multi-way constraint solver is entirely optional. We have integrated the QOCA constraint solver [6] with CIDER. It supports linear constraints (such as $x + 2y \leq 5$), but cannot handle constraints involving higher order polynomials (such as $x^2 - y^2 = 7$). However, linear constraints were powerful enough to meet the requirements of our example application.

Another benefit of using a multi-way constraint solver with the Diagram Processor is that production constraints can be allowed a certain level of tolerance, which is then automatically corrected for by the constraint solver. For example, when evaluating the production for a `NormalState` symbol, the `mid` attributes of the input `Text` and `Circle` symbols would not need to be exactly the same. Slight differences between the two `mid` attributes would be allowed during constraint testing. Upon successful evaluation of the production, these differences would then automatically be corrected by the constraint solver, which would enforce the constraint without any tolerance so that the two values match exactly. This is important because SDE users cannot always be expected to draw precise diagrams.

One interesting issue arising with the use of an external constraint solver is how it cooperates with the built-in one-way constraint solver when attribute values are updated. Ideally, every constraint that is involved in structure-preservation should be handled by the multi-way solver. If this is not the case, CIDER adds additional multi-way constraints for equations handled by the one-way solver to maintain consistency (ensuring that none of the dependent one-way variables can be changed by the multi-way solver).

IV. TRANSFORMATIONS

We also want to facilitate processes such as animation, reasoning, and automated user interactions. This is achieved through the use of transformations. One of the more common uses of transformations is for diagram animation. For example, consider the processing of input strings in a FSA diagram. For a given input string (which is associated with a certain state), if there is a transition from that state whose label matches the first character of the input string, then the input string can be processed as shown in the example in Figure 3.

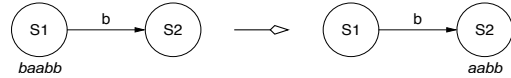


Fig. 3. A simple transformation involving a FSA input string.

However, transformations can also be used for more complex tasks, such as diagrammatic reasoning. For example, consider an application that also allows FSAs to be created from regular expressions. The use of transformations makes this possible. As an example, Figure 4 illustrates a transformation that converts a disjunctive expression. One of the motivations for developing CIDER is to investigate the usefulness of transformations for such tasks.

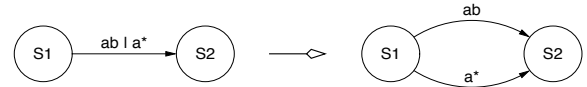


Fig. 4. A transformation used in converting a regular expression into a FSA.

The basic idea behind a transformation is to change the parse forest of a diagram from one valid state to a different valid state. However, while the SDE user sees the diagram change in a single step, the processing that is involved can be considerably more complex. For example, modifying a parse forest involves actions such as removing symbols, adding new symbols, and modifying attribute values. This raises the question of how transformations are to be formally specified.

It is desirable to specify high-level transformations on the level of non-terminal symbols. Such specifications must be well-defined to ensure consistency among terminal symbols, and that the resulting diagram is displayable. This is not trivial. For example, if we apply a transformation to a FSA diagram that involves the creation of a non-terminal `NormalState` symbol, we do not want this symbol to be added to the parse forest as a terminal symbol since this would mean that it could not be treated as a composite of a `Circle` and a `Text` symbol, and as such its behaviour would be inconsistent with other `NormalState` symbols. Rather, if a `NormalState` symbol is to be created, the required terminal symbols (`Circle` and `Text`) should be created first, and then the production to produce the `NormalState` should be applied to these symbols. This would result in the parse tree of a `NormalState` symbol produced by a transformation being identical to that produced by the application user.

Transformations are carried out by the Diagram Processor,

and are of the form

$$S_1, \dots, S_n \text{ where } C \implies \text{apply}(A_1, \dots, A_m)$$

Transformations are initially evaluated by testing if, for some multiset of input symbols, S_1, \dots, S_n , the attributes of the LHS symbols satisfy the condition C . In this respect they are similar to production rules but unlike productions they are not applied automatically. Transformation conditions can use existentially quantified symbols and negative context.

If the initial evaluation of a transformation is correct, then the LHS symbols are removed from the parse forest. Before each LHS symbol is removed it must be checked to see if it is used by another production further up in the parse forest, and if this is the case, then that production and its LHS symbols must be removed first. Also, if a LHS symbol is non-terminal, the entire sub-tree of this symbol is removed. For each symbol removed it must be checked whether any remaining symbols in the parse forest are affected.

When the LHS symbols have been removed, the next step is to apply the actions A_1, \dots, A_m . There are three types of action: add a new terminal symbol to the parse forest (and provide initial values for all of its attributes), modify the attribute of a symbol, and apply a particular production to a specified set of symbols. For example, consider again the case of a transformation that involves the creation of a `NormalState` symbol. The action list would look like the following.

```
NTS -> nc:Circle(20, <45,30>)
NTS -> nt:Text("S4", <45,30>)
AP -> NormalStateProduction
      using (nc -> c1, nt -> t)
```

The first line creates a new terminal symbol (NTS) which is a `Circle` called `nc` with its initial radius and mid-point defined between the parentheses. The second line creates a new `Text` symbol called `nt`, with its initial label and mid-point also specified. The last two lines apply a production `AP` which is the `NormalState` production (defined earlier in this paper). It specifies that the production should only be attempted with the newly created `Circle` symbol, `nc`, mapped to the RHS symbol `c1` in the production, while the newly created `Text` symbol, `nt`, is mapped to the RHS symbol `t`. Note that it is also possible to create multi-level parse trees.

Specifying productions to be applied in a transformation can present problems. This is because the productions requested may not be applicable. In this case the entire transformation is deemed unsuccessful, and the parse forest is restored to its state before the transformation was attempted. This may make the notion of applying productions within transformations seem a bit unwieldy at first. However, because incremental parsing continues during a transformation, this method ensures that a transformation will never be applied unless the resulting parse forest is guaranteed to be in a stable and consistent state. To support rolling back a transformation, the Diagram Processor implements an *undo* feature.

Figure 3 gave a simple example of how an input string is processed in a FSA diagram. What if the FSA in question

is non-deterministic and there are two possible transitions? A standard transformation would just choose the first valid possibility that it found and apply the transformation accordingly, which means the diagram would behave non-deterministically. The CIDER toolkit also allows for *parallel* transformations to be performed. This means that for a non-deterministic FSA, it can simultaneously process all input string possibilities in a single step, as illustrated in Figure 5.

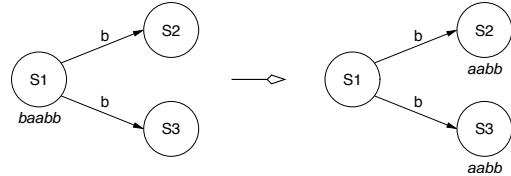


Fig. 5. A parallel transformation involving a FSA input string.

It would be desirable to specify a single transformation that processes an entire input string and reports on whether it was accepted by a FSA. To facilitate such complex transformations the CIDER toolkit allows the application programmer to specify compound transformations. These provide the standard regular expression operations: disjunction, concatenation and repetition (not to be confused with *parallel* transformations, which behave differently).

V. COMPILATION

For improved run-time performance, CIDER compiles the grammar and transformation specifications rather than interpreting them. The compiler produces Diagram Processor components that are tailored for a particular diagram notation (see Figure 2). Apart from delivering substantial performance improvements, a separate compile phase also allows thorough validity checks of the diagram specification. For example, the CIDER Compiler constructs a symbol type dependency graph from all of the productions to ensure that there are no negative dependencies that might cause infinite loops. An attribute dependency graph is also constructed from all of the productions to make sure that no cyclic attribute updates are present. The Compiler checks that the productions and transformations are well-defined and do not refer to attributes or symbols that have not been declared, and ensures that abstract symbol types are used correctly.

Like all of the CIDER toolkit components, the Compiler is written in Java. It can receive the input specification in a format very similar to that used in the examples in this paper, and convert this into an XML format before processing. The Compiler outputs Java code to be added to the Diagram Processor component.

In the ideal case all parts of the specification should be compiled. However, since the application developer can use arbitrary constraint solvers, the Compiler cannot always know how to compile multi-way solver constraints.

In uncompiled form a multi-way constraint is passed to the constraint-solver as a string, such as `[c1.mid] - [c2.mid] == 0.0`, which must be parsed and validated by the constraint solver each time it is used (i.e. at runtime).

TABLE I
CODING REQUIRED TO CREATE THE EXAMPLE FSA EDITOR.

Application Component	Lines of Code	
	Basic Editor Functionality	SDE Specific Functionality
Terminal Symbol draw functions	166	0
Production Constraint functions	0	15
Transformation Constraint functions	0	143
Main FSA Editor class	351	35

It would clearly be more efficient for these strings to be parsed and validated in the compile phase. To support this, the application developer has the option to provide a *Constraint Compiler*, with which the CIDER Compiler can communicate via the Constraint Compiler interface.

VI. AN EXAMPLE EDITOR

Using the CIDER toolkit, we have created an example FSA editor that includes incremental interpretation, structure preserving manipulations, and implements transformations that allow input strings to be processed in parallel (as described earlier), as well as allowing a FSA to be created from a regular expression. A screenshot of the editor is given in Figure 6.

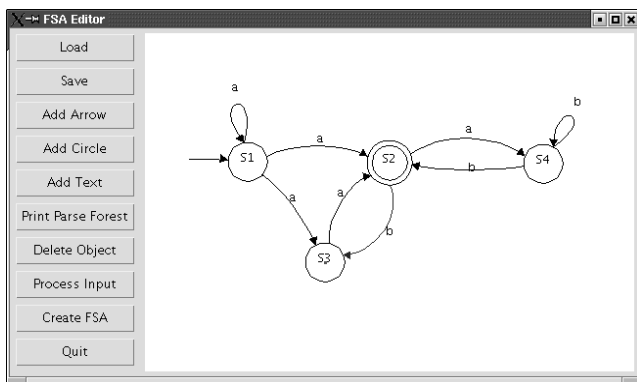


Fig. 6. A screenshot of the FSA editor created with the CIDER toolkit.

Table I lists the amount of code that had to be written in creating the example editor, distinguishing between code that was needed to produce basic editor functionality, and code that was needed for SDE specific functionality (such as parsing and applying transformations). In addition, the application developer needed to provide an input specification consisting of 11 production rules (with their symbol definitions) and 13 transformations, which took up 232 and 181 lines respectively.

Some additional code (456 lines) was needed for the QOCA constraint solver and its compiler interface. However, this must be considered as a generic extension to the toolkit and not as specific to this application.

VII. CONCLUSIONS AND FUTURE WORK

The need for generic software support for SDE construction is widely recognized, yet previous SDE construction systems are not widely used. In this paper we have examined the reasons why and identified lack of flexibility arising from their underlying architectures as the main problem. We have presented CIDER, an SDE construction toolkit which holds the

promise to overcome these architectural problems by using a component-based approach. CIDER is also designed to be as complete as possible, i.e. to cover most common tasks of SDE construction, in particular object-oriented modelling of diagrams, incremental interpretation, structure-preserving direct manipulation, diagram transformation and layout.

The analysis of our example SDE implementation does indeed demonstrate that CIDER simplifies the task of SDE construction significantly. The bulk of code clearly is for the basic Editor functionality. Providing the fundamental SDE capabilities, such as incremental interpretation and structure-preserving manipulation requires only 10% additional lines of Java code and 11 production rules. The case for the existing transformations specification is less clear. We needed 181 lines to specify 13 transformations, and it may be possible to achieve this in a better way (for instance, by directly coding them in Java). However, one advantage of an explicit specification is that it is more amenable to formal analysis.

Of course, the development of CIDER is not finished. Two further lines of work in particular will be pursued. Firstly, we will extend the toolkit to provide beautification and automatic layout. Second, we will perform further case studies to evaluate the use of CIDER for more complex SDEs such as for UML diagrams, and further investigate transformations.

Our main goal in constructing CIDER is to foster the development of SDEs by providing a publicly available tool for their implementation. We believe that the current system which is freely available on the web under GNU public license¹ overcomes traditional architectural problems and already provides a solid basis for SDE implementation and that the targeted extensions will continue to extend its scope and usefulness.

REFERENCES

- [1] R. Bardohl, M. Niemann, and M. Schwarze. GenGed – a development environment for visual languages. In *Applications of Graph Transformations with Industrial Relevance*, pages 233–240, 2000.
- [2] B.T. VanDer Zanden et al. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):776–796, 2001.
- [3] S.S. Chok and K. Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of the 11th ACM Symposium on User Interface Software and Technology*, 1998.
- [4] V. Haarslev. A fully formalized theory for describing visual notations. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 261–292. Springer, New York, 1998.
- [5] K. Marriott. Constraint multiset grammars. In *IEEE Symposium on Visual Languages*, pages 118–125, 1994.
- [6] K. Marriott and S.S. Chok. Qoca: A constraint solving toolkit for interactive graphical applications. *Constraints*, 7(3/4):229–254, 2002.
- [7] K. Marriott and B. Meyer. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8(4):374–402, 1997.
- [8] B. Meyer, K. Marriott, and G. Allwein. Intelligent diagrammatic interfaces: State of the art. In P. Olivier, M. Anderson, and B. Meyer, editors, *Diagrammatic Representation and Reasoning*. Springer-Verlag, London, 2001.
- [9] B. Meyer, H. Zweckstetter, L. Mandel, and Z. Gassmann. Automatic construction of intelligent diagrammatic environments. In *HCI'99: 8th Int. Conf. on Human-Computer Interaction*, Munich, August 1999.
- [10] M. Minas and G. Viehstaedt. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *IEEE Workshop on Visual Languages*, pages 203–210, 1995.

¹Website: <http://www.csse.monash.edu.au/~tonyj/CIDER/>