1 Introduction

The importance of visual programming languages is now widely acknowledged within the computer science community, as is the importance of giving programming languages a semantics. However, although visual programming languages may be easier for human users to cope with than traditional computer languages, specifying their syntax and semantics for implementation/program verification purposes is hard. This is because traditional formalization tools (e.g., attribute grammars) do not include a spatial vocabulary that could be used to describe diagrammatic representations.

In this abstract we outline an alternative way of formally describing visual languages by showing how the syntax and semantics of a particular visual programming language (Pictorial Janus) can very naturally be specified in the RCC\(^1\) spatial logic. RCC was originally developed for representing and reasoning about physical situations from a qualitative viewpoint. It is based on a calculus of individuals, and a set of eight pairwise disjoint and jointly exhaustive relations between pairs of regions are defined from a single primitive. With very few additional primitives the static semantics of the Pictorial Janus can be defined within RCC. In the remainder of this abstract we first introduce Pictorial Janus and the RCC spatial logic before showing how the syntax of Janus can be described. We finish by giving a brief description of how the procedural semantics of the language is defined within RCC.

2 An overview of Pictorial Janus

Pictorial Janus [8] is a visual form of the concurrent constraint-based language Janus [11]. Concurrent languages are particularly suited to representation in pictorial form as diagrammatic information makes the relationships between the various processes easy to see. Janus

---

\(^1\)RCC stands for either ‘Region Connection Calculus’ or ‘Randell, Cohn and Cui’ depending on one’s viewpoint.
itself is a simple, general purpose language in which a network of asynchronous program agents may be linked by channels along which data and information about channel access rights can be passed. Agents may contain one or more rules which can be used to process incoming messages. Computation in Janus involves selecting appropriate rules to apply to the inputs of an agent and replacing the agent with the chosen rule. To determine which of an agent’s rules to apply, Janus uses a matching process analogous to that found in many Herbrand languages (Prolog, GHC, Strand, etc.). A successful rule match results in the inputs and outputs of the agent becoming the inputs and outputs of the rule, which then replaces the agent itself – this process is called reduction. Rules may contain calls to other agents or even recursive calls to their parent agent.

The syntax of Janus can be entirely represented in pictorial form, which makes it an ideal basis for a visual programming language. Figure 1 shows the Pictorial Janus syntax for constants and functions. A constant consists of a closed contour (the shape is irrelevant) containing a number or string (i.e. what the constant represents) and a single internal port. The internal port is represented by another closed contour abutting the constant but wholly inside it and acts as a handle for the entire object. Ports cannot themselves contain any elements. Functions are represented by closed contours containing a label and an internal port together with any number of external ports. In this case we have illustrated a list constructor-function, cons, which normally takes two arguments and thus requires two external ports. The final part of the figure shows how the cons function can be used to build up a list.

A Pictorial Janus agent is a closed contour containing rules, a call arrow to another agent contour, or a label. It may have any number of external ports but no internal ports. A rule is defined in exactly the same way as an agent but with the additional requirement that it must be contained within an agent. Agents may communicate via channels: directed curves linking two ports (an arrow is used to indicate directionality). Finally, links are undirected curves joining two ports.

The semantics of the language, as described in [8], are somewhat ambiguous. However, the computational model is relatively straightforward. The first picture of Figure 2 shows a simple append program in Janus, together with data in the form of two linked lists.

In Janus the execution of a program is achieved by allowing agents to asynchronously ‘reduce’. Reduction consists of three steps: (1) Matching — the agent’s rules are checked

---

2 We will colour ports black in all figures for clarity.
against its current inputs in order to determine which, if any, are applicable. Matched rules are then expanded to exactly overlap the agent; (2) Commit — matched elements ‘dissolve’ leaving behind a new configuration; (3) Link shrinkage — any remaining links are shrunk to zero length and remaining elements are scaled.

Figure 2 demonstrates the reduction steps that result from applying the append program to two single element lists (a). In the first reduction cycle the recursive (lower) rule is matched and expands to take the place of the append agent (b). The commit stage sees the matched rule/agent contours and other matched elements dissolve leaving a new append agent together with a number of links. Finally, (c) the remaining links are shrunk to zero length, effectively transferring the first element of the lower list to the output. At this point the new append agent is expanded. In the next reduction cycle the base-case (upper) rule of the new append agent is matched with the remaining input (e). Once again, the matched elements are dissolved (f) and the links contracted (g). Reduction is halted as the (irreducible) answer list is all that remains.

3 An Overview of the RCC Qualitative Spatial Logic

The RCC spatial logic, originally developed to facilitate spatial reasoning within Artificial Intelligence (AI) domains, is loosely based upon Clarke’s [1, 2] ‘calculus of individuals based on connection’, and is expressed in the many sorted logic LLAMA [4]. The most distinctive feature of Clarke’s ‘calculus of individuals’, and of our work, is that extended regions rather than points are taken as fundamental. Informally, these regions may be thought of as infinite in number, and ‘connection’ may be any relation from external contact (touching without overlapping) to spatial or temporal identity. Spatial regions may have one, two, three, or even more than three dimensions, but in any particular model of the formal theory, all regions are of the same dimensionality. Thus, if we are concerned with a two-dimensional model the points at which these regions meet are not themselves considered regions. We introduce a sort symbol REGION to denote the set of all regions.

The basic part of the formal theory assumes a primitive dyadic relation: C(x, y), read as ‘x connects with y’ (where x and y are regions). Two axioms are used to specify that C is reflexive and symmetric, C can be given a topological interpretation in terms of points incident in regions. In this interpretation, C(x, y) holds when the topological closures of regions x and y share at least one point. In this paper (though not in our work as a whole) we make the assumption that all regions are topologically closed (i.e. include their boundaries).

Using the relation C, further dyadic relations are defined. These relations are DC (is disconnected from), P (is a part of), PP (is a proper part of), EQ or = (is spatiotemporally identical with), O (overlaps), DR (is discrete from), PO (partially overlaps), EC (is externally connected with), TPP (is a tangential proper part of), and NTPP (is a nontangential proper part of). The relations P, PP, TPP and NTPP have inverses (here symbolised Pi, PPi, TPPi and NTPPi). Of the defined relations, the set DC, EC, PO, TPP, NTPP, EQ, TPPi and NTPPi form a jointly exhaustive and pairwise disjoint set of ‘base relations’ shown in figure 3.3.

We draw the reader’s attention to the distinction between C(x, y) (x connects with y) on the one hand and O(x, y) (x overlaps with y) on the other. In the latter case, but not necessarily in the former, there is a REGION which is a part of both x and y.

3 In fact, the other relations can be expressed in terms of disjunctions of the base relations e.g. DR is equivalent to EC ∨ DC.
Figure 2: Execution of append
The complement of a \textsc{region}, and the sum, product (or intersection) and difference of a pair of \textsc{region}s are also defined. Here, we will use infix arithmetic operators to denote these functions (e.g. + denotes the sum of two \textsc{region}s and – their difference). We also define a predicate \textsc{maxpart}(y, x) which is true when y is a maximal connected (i.e. \textsc{con}) \textsc{region} of x.

\section{Axiomatizing the Syntax of Janus}

We now show how RCC can be used to describe the syntax of a selection of Pictorial Janus program elements. Owing to space restrictions, we concentrate here on ports, constants and functions. The remaining program elements are discussed fully in [5].

The RCC spatial logic is a logic for talking about \textsc{region}s. Clearly, the \textsc{region}s of interest in the present situation are those corresponding to the various individual picture elements that make up a Janus program. Thus we assume that a picture is made up of primitive features: character strings, lines, directed lines and closed curves. Each of these corresponds to a \textsc{region}, the area actually occupied by the lines composing each of feature in the first three cases (lines in real diagrams have non zero thickness of course), and the area enclosed by the curve in the final case.

One important point to note when reading the axioms and definitions below is that we are assuming a sorted logic where we have a sort, which we will call \textsc{ppes} (Primitive Picture Elements) which is disjoint from the sort \textsc{region} of pure spatial \textsc{region}s which is already assumed in the previous descriptions of the logic (e.g. [10]). The intended inhabitants of the \textsc{ppes} sort are sets of the strings, lines, directed lines and closed curves in the picture under consideration. A \textsc{ppes} which denotes a singleton picture element, we will call a \textsc{ppe} which is a subsort of \textsc{ppes}. Given a picture at a particular time we can identify a particular spatial \textsc{region} with each \textsc{ppes}: thus we can imagine a transfer function which maps \textsc{ppes} to \textsc{region}s. However, we will not yet make this transfer function explicit but will assume that terms denoting \textsc{ppes} are automatically coerced to terms of sort \textsc{region} when required, i.e. when they appear as arguments to the predicates of the spatial logic such as \textsc{pp}, \textsc{tpp}, \textsc{ec}, \textsc{dc} etc\footnote{The only possible confusion would arise with the equality predicate, since we will want not only to equate \textsc{region}s but also \textsc{ppes}, without coercing the latter. In order to handle this we will use two different equality predicates: one for \textsc{ppes} and one for \textsc{region}s.}.

\begin{figure}
\centering
\begin{tabular}{ccc}
\includegraphics[width=0.3\textwidth]{figure3a} & \includegraphics[width=0.3\textwidth]{figure3b} & \includegraphics[width=0.3\textwidth]{figure3c} \\
\textbf{DC}(A,B) & \textbf{EC}(A,B) & \textbf{PO}(A,B) \\
'A is disconnected from B' & 'A is externally connected to B' & 'A is partially overlapping B'
\end{tabular}
\caption{RCC base relations}
\end{figure}
REGIONs denoted by the individual PPEs comprising the PPES. One reason why we want to distinguish PPES from REGIONs is that when animating a Janus program, the various PPES will transform, and, for example, possibly becoming coincident – however we will still want to distinguish the two PPES, although they may denote identical REGIONs—imagine two acetate sheets overlaid on an overhead projector: lines may coincide as the sheets move relative to each other, but the lines remain distinct.

We also need to introduce a sort JPE (Janus Program Elements) whose intended inhabitants are sets of agents, rules, and the other program structures of Janus. These are all PPES, i.e. JPE is a subsort of PPES. A JPE which denotes a single primitive Janus Program structure, we will call a JPE which is a subsort of JPE. In the axioms and definitions below all the variables are implicitly restricted to range over the sorts for which the formula is well sorted, as in the sorted logic LLAMA [4]. As an aid to readability, a variable name is annotated with a * when it ranges over JPE or PPES. This approach requires that we specify the allowable sorts of all the function and predicate symbols. The topological predicates (P, TPP, ∇, Maxpart etc) all require REGIONs for all their arguments. For the other predicates we will specify their sort constraints as we introduce them, with the following notation: sort: α(τ₁, ..., τₙ). For specifying the argument and result sorts of functions, we will use the notation: sort: α(τ₁, ..., τₙ) ⇒ τₙ₊₁.

Each of the different kind of PPEs (i.e. Line, Dline, String, Ccurve, String) is a subsort of PPE. In order to distinguish the beginning and the end of directed lines, we introduce two functions, start and end which denote REGIONs and effectively divide a directed line into two.

\[
\text{sort: start(DLINE)} \Rightarrow \text{REGION} \\
\text{sort: end(DLINE)} \Rightarrow \text{REGION} \\
\forall (x)[\text{TPP(start(x), x)} \land \text{TPP(end(x), x)} \land (\text{start(x) + end(x) = x}) \land \text{DR(start(x), end(x))}]
\]

This is also an appropriate point to state explicitly exactly what the sorts JPE and PPE encompass (we will occasionally use the (sort) predicates JPE(x) and PPE(x) explicitly in order to coerce a variable to the appropriate sort).

\[
\text{JPE(x) } \equiv_{def} \text{Agent(x) } \lor \text{Port(x) } \lor \text{Rule(x) } \lor \text{Constant(x) } \lor \\
\text{Function(x) } \lor \text{Link(x) } \lor \text{Channel(x) } \lor \text{Carrow(x) } \\
\text{PPE(x) } \equiv_{def} \text{Ccurve(x) } \lor \text{Dline(x) } \lor \text{Line(x) } \lor \text{String(x) }
\]

One important question which we do not address here fully is the question of parsing a picture into a finite set of PPEs. In general a picture will be ambiguous. For example, one rule of the append program (figure 2) has a line crossing the circle, effectively dividing it into two. We could either parse the picture as just described, or as two externally connected part-circles with an externally connected line. There are many ways such ambiguities could be eliminated; the shape conventions for particular kinds of Janus primitives assumed by [8] will often disambiguate, as indeed will observing the order in which the picture elements are actually drawn as [8] also notes. However we will not address this issue further here and simply assume the entire picture has been parsed as represented by a set of Line, Dline, Ccurve and String propositions. Of course the inability to make sense of a picture though the axioms below would be a reason to reject the particular parse and try another.

In the next subsections we define all the Janus Program elements (JPEs): agents, rules, symbols, the standard = for the latter case, and = for the former.
4.1 Ports

“A port is a closed contour with nothing inside”.

\textbf{sort:\Port (JPE)}
\Port (port) \equiv_{d.f.} \text{Ccurve}(port) \land -\exists (y)[\text{PPE}(y) \land \text{PP}(y, port)]

Two kinds of ports are distinguished: internal and external ports. We define binary predicates for each of these two concepts since these notions are relative to another JPE.

\textbf{sort:\Iport (JPE, JPE)}
\Iport (port, x) \equiv_{d.f.} \Port (port) \land \text{TPP}(port, x)
\Eport (port, x) \equiv_{d.f.} \Port (port) \land \text{EC}(port, x)

It is necessary to ensure that the only PPEs that can overlap ports are other ports. In particular, one internal port may overlap one external port. This occurs when, for example, a constant is attached to an external port of a function: the internal port of the constant is overlaid onto the external port of the function (see figure 1). The following axioms enforce this.

\[ \forall (port, w) \, \Iport (port, w) \rightarrow \]
\[ \neg \exists (y)[\text{PPE}(y) \land y \neq port \land (\text{PO}(y, port) \lor y = port)] \lor \]
\[ (\exists (y, z)[\Iport (y, z) \land \forall (x)[(\text{PPE}(x) \land y \neq port \land (\text{PO}(y, port) \lor y = port) \rightarrow x = y)])] \]

\[ \forall (port, w) \, \Eport (port, w) \rightarrow \]
\[ \neg \exists (y)[\text{PPE}(y) \land y \neq port \land (\text{PO}(y, port) \lor y = port)] \lor \]
\[ (\exists (y, z)[\Iport (y, z) \land \forall (x)[(\text{PPE}(x) \land y \neq port \land (\text{PO}(y, port) \lor y = port) \rightarrow x = y)])] \]

Two further axioms are necessary to ensure that a port can only be an external port for one particular JPE or an internal port for one particular JPE. Note that this does not prevent a port from being both an internal port for one JPE and an external port for another. This is the case in figure 1 — each of the overlapping ports can be parsed as either an internal port of one JPE or an external port of the other. In practice, we would simply take the results of one parse and ignore the other as the two are essentially isomorphic.

\[ \forall (port, x) \, \Eport (port, x) \rightarrow \neg \exists (y)[x \neq y \land \Eport (port, y)] \]

\[ \forall (port, x) \, \Iport (port, x) \rightarrow \neg \exists (y)[x \neq y \land \Iport (port, y)] \]

Some JPEs may have more than one external port; a set of external ports will denote a multipiece REGION all of whose maximal subparts are external ports.

\textbf{sort:\Eports (JPEs, JPE)}
\Eports (ports*, x) \equiv_{d.f.} \forall (subpart)[\text{Maxpart}(subpart, ports*) \leftrightarrow \Eport(subpart, x)]

4.2 Constants

“A constant has a closed contour, an internal port, no external ports, with a string or number inside”.

For simplicity, we assume here that numbers are strings. It will be convenient to be able to easily identify the contour, the internal port and perhaps the label. Therefore we will define both a rank one and a higher rank predicate for such structures.
sort: Constant(JPE)

sort: Constant(JPE,JPE,PPE)

Constant(const) \(\equiv_{def} \exists (iport, label)[\text{Constant(const, iport, label)}]\)

Constant(contour, iport, label) \(\equiv_{def} \exists C\text{curve(contour)}\)

\(\exists (eport)[\text{Eport(eport, contour)}]\)

\((\forall (port)[\text{Iport(port, contour)} \rightarrow \text{port} = \text{iport}] \lor \exists (port', x)[\text{Eport(port', x)} \land \forall (port)[\text{Iport(port, contour)} \rightarrow \text{port} = \text{iport} \lor \text{port} = \text{port'}]])\)

Note that we make sure that the set of PPEs that form the constant includes the port and the string, not just the contour, in order to make sure that when we want to ‘delete’ a constant (as part of the procedural semantics) we will delete all of it not just the contour. Similar considerations will motivate the definitions of the other JPEs below.

4.3 Functions

“A function has a closed contour, one internal port, some external ports and a label”. As with constants, we define both a rank one and a higher rank predicate.

sort: Function(JPE)

sort: Function(JPE,JPE,JPES,PPE)

Function(func) \(\equiv_{def} \exists (contour, iport, eports^*, label)[\text{Function(contour, iport, eports^*, label)} \land \text{func} = \text{contour} + \text{eports}^*]\)

Function(contour, iport, eports^*, label) \(\equiv_{def} \exists C\text{curve(contour)}\)

\(\exists (eport, contour)[\text{String(label)} \land \text{P(label, contour)} \land \text{Iport(iport, contour)} \land \forall (port)[\text{Iport(port, contour)} \rightarrow \text{port} = \text{iport}] \lor \exists (port', x)[\text{Eport(port', x)} \land \forall (port)[\text{Iport(port, contour)} \rightarrow \text{port} = \text{iport} \lor \text{port} = \text{port'}]])\)

Collectively, we will call functions and constants, functors.

sort: Functor(JPE)

Functor(x) \(\equiv_{def} \text{Function(x)} \lor \text{Constant(x)}\)

5 Further Work

Thus far we have only shown how RCC could be used to specify the visual syntax of Pictorial Janus; however, equally interesting and important, is the task of formalizing the procedural semantics of the language. In the space remaining, we give a brief overview of our approach.

We view computation as proceeding through a cycle of different states that loosely correspond to the match, commit and link shrinkage steps of Janus described in section 2. Each step contains its own particular set of extra constraints on spatial relationship changes. For example, normally rules must be proper parts of agents (i.e. they must be completely contained within an agent). However, during matching steps this constraint must be relaxed so that a rule can expand to become EQ to its surrounding agent.

The spatial relationships between the JPE’s of a program are stored in an internal database. During the commit and link shrinkage steps the database is updated to reflect
the changes in spatial relationship between the JPE’s. This effectively amounts to asserting and retracting RCC relations between elements. In cases where program elements cease to exist, as when links shrink to zero length during the link shrinkage step, we simply delete all RCC relationships that mention the element in question. A Janus program runs through these program state cycles until no further changes in spatial relationship can be made — i.e. all the agents are fully reduced and the program has run to completion.

6 Final Comments

We have described how a spatial logic, originally developed for representing and reasoning about physical systems can be used to provide a formal description of the Pictorial Janus visual programming language. We believe that not only is this development quite natural, but also it is very important to have formal specifications of visual programming languages, just as any other programming language. The approach may also provide an approach to the prototyping of visual languages. We have started to build a program which uses direct encoding of our RCC specification to parse Pictorial Janus programs presented in terms of a set of regions and the list of RCC relations that hold between them. Having parsed the program, we intend to use our RCC version of the semantics in conjunction with a spatial simulation system to execute programs.

Of course, one route to provide such a formal specification is via a textual language as Kahn and Saraswat propose, so that Pictorial Janus is translated to (textual) Janus and then a semantics given to this language. Which is the more desirable will depend on many things. For example if the actual implementation of Pictorial Janus is in terms of textual Janus then this latter route might be rather appropriate. However, it is clear that the implementation need not take this approach and in such situations it seems perhaps unnecessarily cumbersome to take this route rather than the more direct approach we have presented. Moreover, turning away from Janus in particular, one could imagine some other visual language for which there was no textual variant and in such situations it would arguably be obviously best to take the direct approach. If the user of the visual language does not think in terms of an underlying textual language, then a semantics which does not involve a textual language is better.

In addition to our research, Haarslev has also worked on formalizing visual languages in terms of spatial logics [7]. Haarslev uses a simplified version of Clementini et al’s spatial logic [3] to describe Pictorial Janus program elements and then translates these descriptions into a description logic for program verification purposes. Haarslev does not attempt to provide a spatial logic description of the procedural semantics of the language. In fact, verified programs are translated to textual Janus for execution. We believe that our approach is closer in spirit to the ideals of visual programming as we view program specification, parsing and execution all in terms of a single common spatial language: RCC. This approach allows us to break free from the need to use any non-spatial language to describe the syntax or semantics of Pictorial Janus.

References


