Procs of 1995 IEEE Symposium on Visual Languages, Germany, 1995, IEEE CS Press, pp. 53-60.

ViTABaL: A Visual Language Supporting Design by Tool Abstraction

John C. Grundy[†] and John G. Hosking[††]

[†]Department of Computer Science
University of Waikato
Private Bag 3105, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

[††]Department of Computer Science
University of Auckland
Private Bag 92019, Auckland, New Zealand
john@cs.auckland.ac.nz

Abstract

We describe a visual language and environment for designing and implementing systems using the tool abstraction paradigm. This paradigm permits systems to be constructed from toolie and abstract data structure components, using an event response mechanism to handle inter-component interaction. This approach leads to systems more easily adapted to functional specification changes than with conventional design.

1. Introduction

In a recent paper, Garlan et al [4] introduce the *tool-abstraction* (TA) paradigm for constructing computer systems that support functional evolution. In this approach groups of *abstract data structures* (ADSs) are shared by a collection of co-operating *toolies*. Each toolie supplies part of the overall system function. Interaction is via event propagation, with toolies being notified appropriately when shared data is modified. Figure 1 shows an example system designed using TA.

Systems supporting TA implementation include spreadsheets, production systems, active data and structure-oriented environments [4]. However, no existing tools support the *design* of software using TA. In this paper we describe ViTABaL (Visual Tool-Abstraction Based Language), a hybrid visual programming environment for both designing and implementing TA-based systems. The paper commences with a brief introduction to TA. In Section 3, we describe the ViTABaL visual language and its environment. Section 4 describes support for textual elaboration and implementation of the visual designs. Implementation is briefly described in Section 5, and run-time visualization in Section 6, before discussion and conclusions are presented.

2. TA vs data-based abstraction

To illustrate TA, we introduce the KWIC (Key Word In Context) index system example used in [4] and [16], to illustrate the benefits of data abstraction over functional decomposition. To quote from [16]:

"The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be *circularly shifted* by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order"

Parnas [16] uses data abstraction to specify a design for this system, with the following modules:

- Line Storage, implementing a sequence of lines, with routines to access/delete chars, words, and lines.
- Input, to read and store the original lines.
- Circular Shifter, to access characters, words, and lines of circular shifts of the stored lines.
- Alphabetizer, to acess shifted lines in lexical order.
- Ouput, to print the circular lists in lexical order.

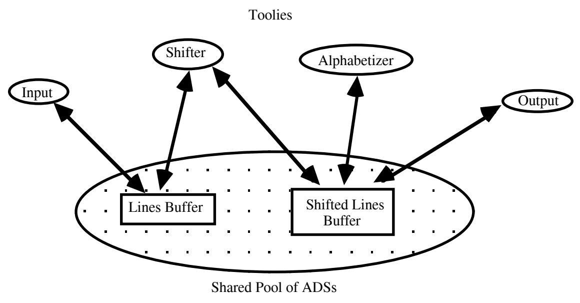


Figure 1. KWIC system modelled by TA (after [4]).

The program invokes Input to read and store the lines using Line Storage. Circular Shifter routines are then used which retrieve lines from Line Storage. Functions from Alphabetizer sort the shifted data, accessing the data via Circular Shifter. Output accesses the sorted list via Alphabetizer. Each module hides data representation and algorithm choices from its users. Parnas' design is quite robust against some, mainly data, specification changes, such as packed vs. unpacked characters, and monolithic vs. incremental alphabetization. However, as is argued in [4], Parnas' design is not robust against *combinations* of functional specification changes, such as:

- Adding the ability to omit shifted or original lines that start with one of a number of "noise" words.

- Including only shifted or original lines starting with one of a specified list of words.

Each such change is fairly straightforward to add, but combinations of such changes interact awkwardly requiring major design changes as "logically independent requirements are difficult to implement without intertwining logically independent implementations" [4].

Figure 1 shows the TA design of [4] for the KWIC system. The input and shifted/alphabetized data are kept as shared ADSs. When a toolie modifies the common data through operations on the ADS, other dependent toolies are notified and invoked indirectly. Note that the implementations of the shared structures are still hidden. Their functionality may, however, be modified by the toolies as explained below. Toolies used in KWIC are:

- Input, reads lines, inserting each in the line buffer.
- Shifter, which is invoked by termination of the insert operation of line buffer, creates the line shifts.
- Alphabetizer, is triggered by completion of shifter activities. It sorts the lines and inserts them into the shifted line buffer.
- Output, prints out the shifted lines.

Modification of the TA design is straighforward. An Omit toolie can be triggered by inserts on the line buffer, causing the insert to abort if the line starts with a word from an omit list. Other code need not change. The same approach can be taken for an Include toolie. Interaction between the two modifications is no longer problematical, as both are triggered by the insert operation (as long as their interaction is managed correctly by the toolie scheme).

TA is quite different to object-oriented (OO) design, which is inherently data abstraction (encapsulating functionality with data). Some OO models support rudimentary TA mechanisms via active data, such as Smalltalk's MVC [13], but these are not sufficient for implementing general purpose TA systems.

Systems which currently utilise TA are application-specific. These include active data in OO systems (shared data is objects, toolies are methods) [6, 13], spreadsheets (shared data is cells, toolies are formulae) [4], structure-oriented editors (data is abstract syntax trees, toolies are attribute equations) [17], and rule-based systems (data is shared data pool, toolies are rules) [2].

We now describe an environment for general purpose TA-based design, and also elaborate on the KWIC design, TA notation and how toolies and ADSs interact.

3. Event propagation views

ViTABaL event propagation views describe the interconnections between toolies and ADSs. These interconnections can be annotated with icons indicating the kinds of event propagation(s) between the connected toolies. For example, Figure 2 shows an event propagation view describing the KWIC system. In this example we have added a "kwic" toolie to indicate starting and finishing of the KWIC system, and added two additional ADSs, an input_file and an output_file. This diagram is a screen dump from the environment we have built to support programming with ViTABaL.

Figure 3 shows the notation used to describe toolies, ADSs and event connections between toolies. This extends the notation in [4] to support specification of event types. ViTABaL makes no semantic distinction between toolies and ADSs – they are implemented and behave in the same way. The distinction is useful in terms of their usage. Toolies typically encapsulate behaviour in that they respond to events to carry out some system function. ADSs typically encapsulate data abstraction and respond to events to store, retrieve or modify data. We use the term toolie to mean toolie or ADS, except where a distinction is required.

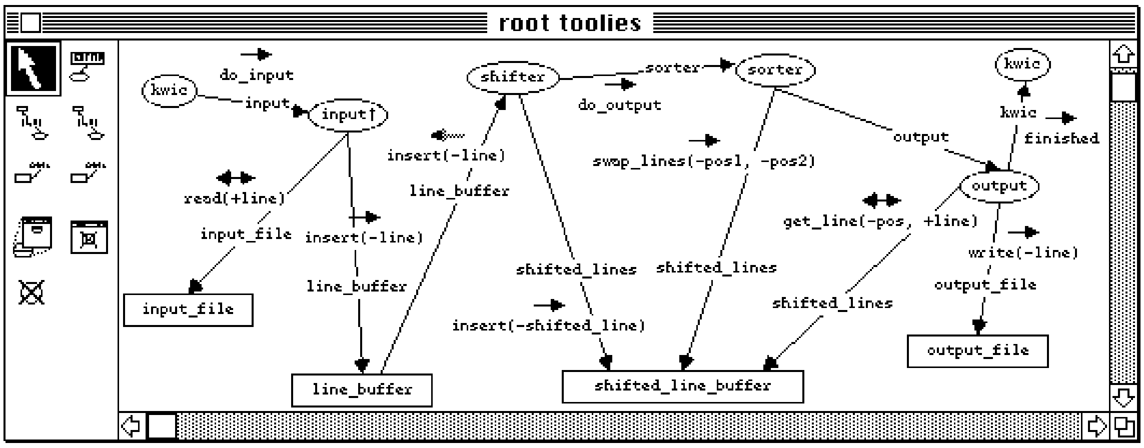


Figure 2. The KWIC system expressed in a ViTABaL Event Propagation view.

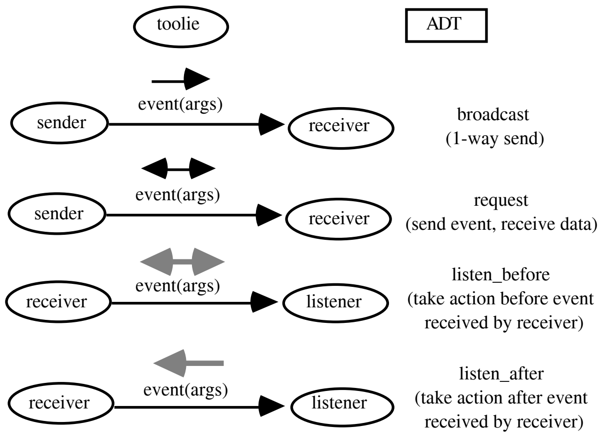


Figure 3. ViTABaL event propagations.

Both event names and data values are propagated along event connections between toolies, unlike most visual languages which use dataflow, such as Prograph [3] and Fabrik [12], or control flow, such as PICT [5]. The event names and argument data types in a view can be hidden if desired, just showing the kind of propagation along a toolie connection.

There are four types of events that can be propagated along event connections:

- broadcast ($\overset{\rightarrow}{\text{event(args)}}$). The named event and optional argument data values are sent from the sender toolie to the receiver toolie. As the sender needs no information from the receiver, if the sender and receiver toolies are executing concurrently, the sender can potentially continue executing before the receiver responds to the event.
- request ($\overset{\leftrightarrow}{\text{event(args)}}$). The named event and argument data value(s) are sent from the sender toolie to the receiver toolie. The sender must wait for the receiver to respond and return any requested values. Supplied and returned arguments are indicated by a '-' meaning supplied by sender and a '+' meaning returned by receiver (e.g. read(+line) and insert(-line) in Figure 2).
- listen_before ($\overset{\leftrightarrow}{\text{event(args)}}$). The listener is sent the named event and argument data values before the receiver actually responds to the event. The listener can even modify the event sent to the receiver, allowing the event response behaviour of the receiver to be modified without actually modifying the receiver code. This is a powerful technique, not supported by other TA or visual languages, allowing toolies to "listen" to events sent to other toolies and possibly modify them. Other language support for such notions, such as CLOS-style wrappers, are limited to simply listening to but not modifying the event.
- listen_after ($\overset{\leftarrow}{\text{event(args)}}$). The listener is sent the named event and argument data values after the receiver has responded to the event. In this case the listener will not modify the behaviour of the receiver's response to the event, but will take some action based on what event and data values the receiver has just responded to. This is similar to MVC-style dependency mechanisms, but has the added advantage of the complete named event and data values being sent to the listener.

Event propagation diagrams show instances of toolie types connected to describe a particular system's functionality. The toolie types and their event responses can be reused in different situations. For example, the input or shifter toolies from the KWIC system may be reused in quite different systems to input data into a line buffer ADS and to shift these lines. ADSs will often be reused in many different systems. For example, the input_file and output_file ADSs from Fig 2 are instances of the same character_file Abstract Data Type (ADT).

Reuse of the toolie type in a different situation may mean an instance of this type is connected to toolies of a variety of different types. For example, instances of the character_file ADT may be connected to any toolie which sends it read/write events. The input toolie is more restricted in its usage, supporting a do_input response and requiring two other toolies responding to read(+line) and insert(-line) events. The KWIC design in Figure 2 could be modified to support quite different line input or storage mechanisms by using instances of other ADTs. ViTABaL checks the send/receive events of each toolie before generating code to ensure each toolie is sent valid events and argument data values.

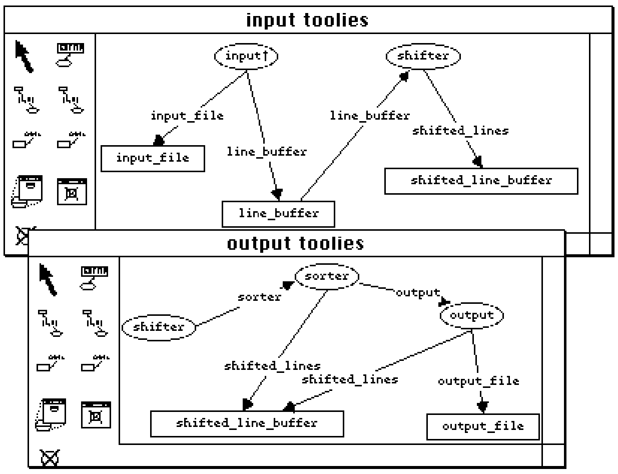


Figure 4. Multiple views of complex systems.

ViTABaL supports multiple views of a system, allowing designers to construct complex systems in parts. Views can share information and can be used to provide both high-level and detailed views of a system. For example, Figure 4 shows two additional views of the KWIC system, one showing the toolies and ADSs associated with line input and shifting, the other showing toolies associated with line sorting and output. In these views the event propagation icons have been hidden, thus showing only the basic interconnections of the KWIC system toolies. The event connection names can also be hidden, if desired. All event propagation views are kept consistent by the environment. For example, renaming a toolie instance, event connection or event propagation in one view is reflected in all other views which share this information.

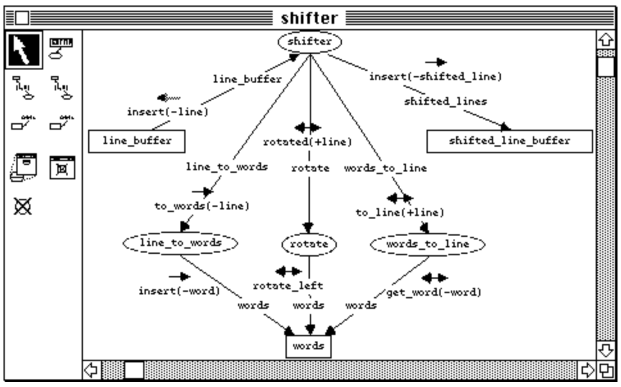


Figure 5. Hierarchical toolie specification.

Hierarchical views are also supported for defining the behaviour of complex toolies, as shown in Figure 5. For example, the KWIC shifter toolie could be composed of toolies to split lines into words and shift the words and an ADS to store the words. The listen_after(insert(-line)) event response of the shifter can thus be specified in terms of another, more detailed event propagation view. The implementation of the words storage for shifter can then be changed in this view without affecting the shifter interface in other views.

ViTABaL allows designs to be easily modified to express specification changes which are not easily or efficiently expressed in other languages. For example, consider the following combination of design changes: i) some lines read in should not be stored, for example blank lines; ii) lines input should be capitalised; iii) The sorter should work incrementally rather than in batch mode; and iv) the input and shifter toolies should run concurrently, possibly on different machines. Figure 6 shows the event propagation view modifications made to express these changes to the KWIC system.

To implement (i) and (ii), omit and upper_case toolies are added. The omit toolie does a listen_before(insert(-line)) on the line_buffer. If input tries to insert any invalid lines, omit prevents line_buffer responding to the insert. The upper_case, toolie also does a listen_before(insert(-line)) on the line_buffer. Serialisation annotations on the event connection allow these toolies to be synchronised appropriately, ensuring the omit toolie does the first listen.

For iii) incremental sorting, the sorter does a listen_before(insert(-line)) on the shifted_line_buffer. It translates this into an insert_in_order(-line) operation. If that event isn't supported by the line buffer ADS, the sorter could step through each line in shifted_line_buffer by sending it a request get_line(-position,+line) event, until it finds the right place to insert the line.

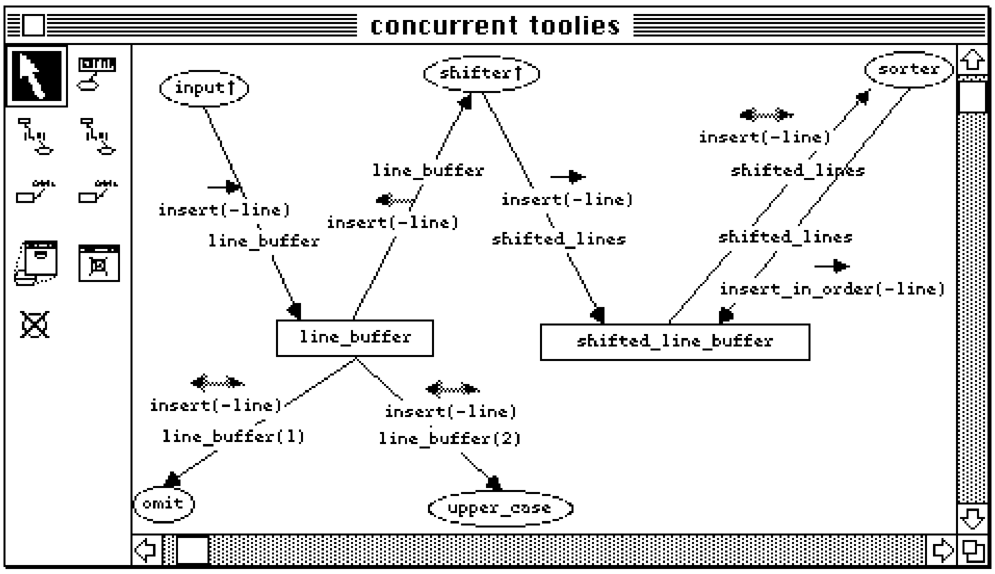


Figure 6. Changes to the KWIC system expressing incremental, concurrent and modified toolie operation.

To implement iv) the input and shifter toolies have been specified to run concurrently, by adding a '†' annotation to their icons. They now require shared, concurrent access to the line buffer and hence ViTABaL generates code which synchronises the events they send to line_buffer. The line_buffer ADS and its omit and upper_case toolies can be run on a third machine, or could be run on one of the input or shifter machines.

As noted in [4], systems supporting data-based abstraction cannot be easily changed in this way to support quite different operation or to modify responses to events. Existing TA systems and existing visual languages similarly do not support facilities to allow designers to modify the behaviour of a system without either requiring system functions to be rewritten or requiring inefficient solutions to be used.

4. Event response views

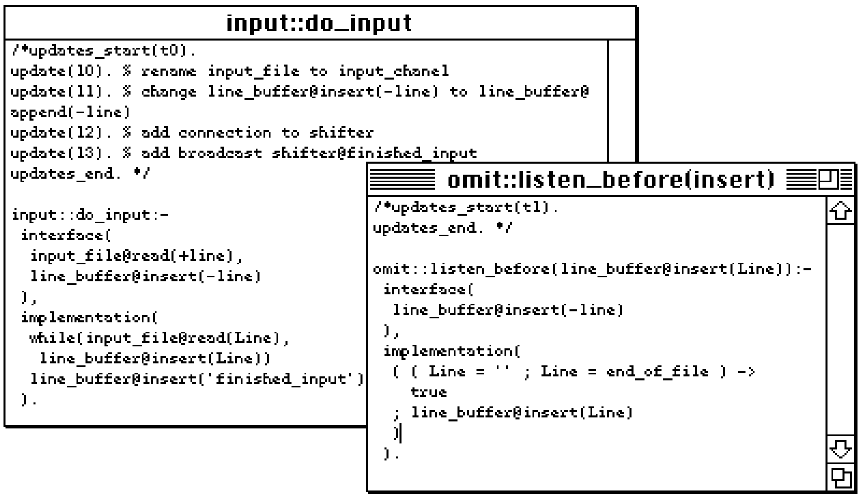
Event propagation views specify toolie instance interconnections and event propagation information between connected toolies. The response that toolies make to events can be specified in two ways. Hierarchical event propagation views can be used to decompose complex responses into smaller toolies and ADSs. Simpler event responses are currently specified in *textual* event response views (although a more visual Prograph-like approach could readily be adapted for this). Figure 7 shows the event response views for input::do_input and omit::listen_before(insert(-line)).

The event response name (e.g. input::do_input) specifies the toolie type name, the event name and variables to hold argument data values. The interface specifies "event ports" which, when instantiated, connect instances of this toolie type to other toolies; events are sent to these connected toolies via the port. Port names do not refer to specific instances nor even to specific toolie types. The name is generic, and ViTABaL determines the actual connected toolie instance/type from the event propagation views. For example, the line_buffer@insert(-line) interface simply indicates that the input::do_input event response can send an insert(-line) event to some toolie using its line_buffer port. This port name must be bound to a specific connection in the event propagation views. This binding is specified in a dialog when toolie icons are connected, but is not normally shown in event propagation views.

This approach means that, for example, the input toolie could be connected to a binary tree ADS to store lines, and as long as the binary tree ADT supports an insert(-line) event response, the system will function correctly. This is not supported by most TA languages.

The implementation section is Prolog code supplemented with event broadcast/request calls. The input::do_input implementation reads lines from the input file and inserts them into the line buffer. The omit::listen_before(line_buffer@insert(Line)) response checks to see if the line is blank (``) or invalid (end_of_file), and if so does not insert the line into the line buffer. ViTABaL translates the event broadcast/request calls into event propagations to the correct toolie instances.

ViTABal maintains consistency between the textual event response views and the visual event propagation views. The header section of the input::do_input event response shows four *change descriptions* indicating changes made to another view which affect the event response specification. The first two are simple for the environment to automatically implement: renaming the input_file port and changing line_buffer@insert(-line) broadcasts to line_buffer@append(-line).



```
/*updates_start(t0).
update(10). % rename input_file to input_chanel
update(11). % change line_buffer@insert(-line) to line_buffer@
append(-line)
update(12). % add connection to shifter
update(13). % add broadcast shifter@finished_input
updates_end. */

input::do_input:-
 interface(
  input_file@read(+line),
  line_buffer@insert(-line)
 ),
 implementation(
  while(input_file@read(Line),
   line_buffer@insert(Line))
  line_buffer@insert('finished_input')
 ).
```

```
/*updates_start(t1).
updates_end. */

omit::listen_before(line_buffer@insert(Line)):-
 interface(
  line_buffer@insert(-line)
 ),
 implementation(
  ( ( Line = '' ; Line = end_of_file ) ->
    true
  ; line_buffer@insert(Line)
  )
 ).
```

Figure 7. Textual event response views for KWIC toolies.

The third change, adding an explicit connection to the shifter toolie, means a new port must be added, although it is not known which event(s) of shifter may be used. The fourth change, adding a broadcast to the shifter toolie, can not be automatically implemented by the environment, as it does not know where to put this broadcast in the implementation text nor which variable should be passed as an argument ('-line' indicates the argument type only). This change must be manually implemented by the programmer, but the ViTABaL at least indicates this manual change is required.

5. Implementation

ViTABaL is implemented by specialising the MViews framework [7, 8]. MViews provides a collection of object-oriented classes written in Snart, an object-oriented Prolog [15]. These classes provide abstractions for specifying data repositories to represent language structure and semantics, multiple textual and graphical views of language structures, and editors for manipulating view information. MViews has been used to build a variety of environments, including SPE and Cerno [10], EPE [1], OOEER and MViewsER [11], MViewsDP [10], and HyperPascal [14].

All views are kept consistent using the MViews view consistency mechanisms. These allow some view updates to be automatically applied to graphical and textual views by the environment. Other changes which can not be automatically carried out are presented to programmers as change descriptions. Users manually implement these changes to restore view consistency [8].

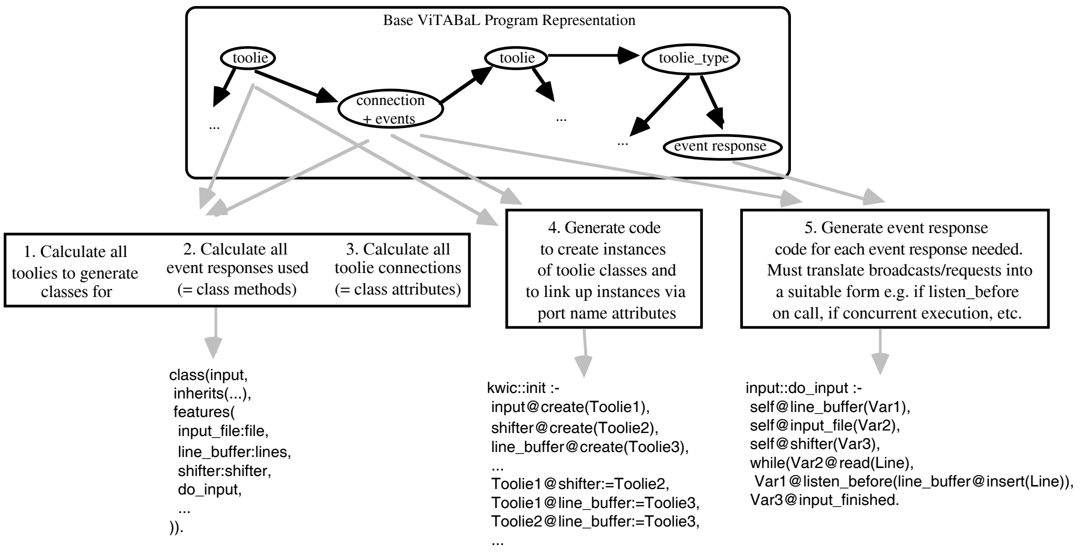
For efficiency, ViTABaL adopts a compilation approach and translates ViTABaL programs into Snart programs. These can be efficiently executed, with generated code being almost as efficient as hand-generated Snart programs. The process of translating a ViTABaL program into a Snart program is illustrated in Figure 8.

Toolie classes, their attributes and methods, and code to create instances of toolie classes and link instances with variables used by event response ports are generated. Event responses are translated into Snart code with appropriate kinds of method calls between toolie objects. Broadcast/requests with no listeners on the receivers and no concurrency between sender/receiver are translated into Snart method calls. Listened events must be sent to listen_before/after methods of listeners, which may then call the real receiver method for the event response.

Concurrency requires broadcasts and requests to be sent along communication channels. Broadcasts can be non-blocking if the receiver and toolies it is connected to don't send requests to the sender. We have run the KWIC system on two Macintosh computers, splitting up the input and shifter toolies. After removing concurrency annotations, ViTABaL regenerates code which runs efficiently on one machine.

ViTABaL generates code to "register" each toolie and toolie connection, allowing programmers to choose toolie objects and events they want visualised or traced. At present ViTABaL regenerates all toolie classes and methods whenever a view is modified. This could be changed to incremental regeneration for efficiency.

ViTABaL performs consistency checks before generating code, to ensure a specification is correct. This includes checking event propagations are supported by appropriate event responses in receiving toolies, ensuring the correct number and type of event arguments are sent, and ensuring event propagations enacted by event responses have suitable receiver instantiations specified in event propagation views.



Base ViTABaL Program Representation
toolie
connection + events
toolie
toolie_type
...
...
...
event response
1. Calculate all toolies to generate classes for
2. Calculate all event responses used (= class methods)
3. Calculate all toolie connections (= class attributes)
4. Generate code to create instances of toolie classes and to link up instances via port name attributes
5. Generate event response code for each event response needed. Must translate broadcasts/requests into a suitable form e.g. if listen_before on call, if concurrent execution, etc.

```
class(input,
 inherits(...),
 features(
  input_file:file,
  line_buffer:lines,
  shifter:shifter,
  do_input,
  ...
)).
```

```
kwic::init :-
 input@create(Toolie1),
 shifter@create(Toolie2),
 line_buffer@create(Toolie3),
 ...
 Toolie1@shifter:=Toolie2,
 Toolie1@line_buffer:=Toolie3,
 Toolie2@line_buffer:=Toolie3,
 ...
```

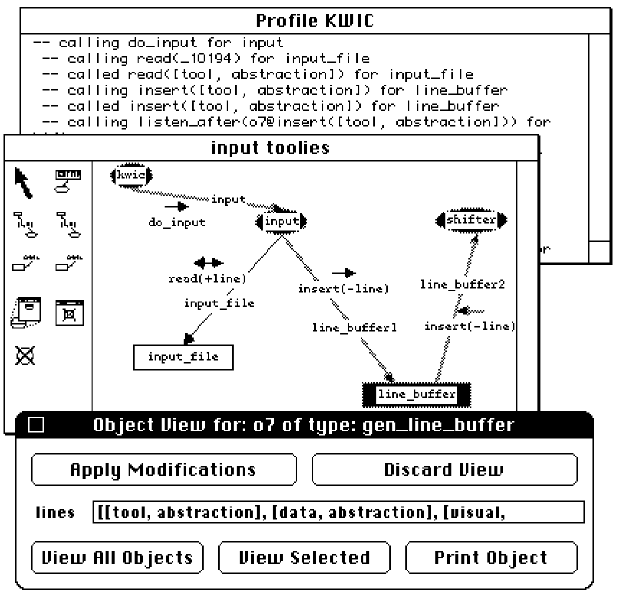
```
input::do_input :-
 self@line_buffer(Var1),
 self@input_file(Var2),
 self@shifter(Var3),
 while(Var2@read(Line),
  Var1@listen_before(line_buffer@insert(Line)),
 Var3@input_finished.
```

Figure 8. Compilation of ViTABaL programs to Snart class implementations.

6. Program visualisation

Executing compiled ViTABaL systems can be visualised in a variety of ways. The low-level state of toolies and ADSs can be displayed in windows which show attribute values of objects associated with toolie instances. Event response code can be traced and stepped through to check the implementation of event responses.

Event propagation views can be animated. As a system executes, the flow of events along toolie connections is shown and the "active" toolies are highlighted. Spy points placed on toolies, toolie events and event connections allow programmers to examine toolie event flow and data values. Profilers attached to toolie events show a list of the event names and data values passing through connections and onto event responses. Toolie state, event propagation animation and profiling is illustrated in Figure 9, where shifter is about to respond to listen_after(line_buffer@insert(-line)).



```
-- calling do_input for input
 -- calling read(_10194) for input_file
 -- called read([tool, abstraction]) for input_file
 -- calling insert([tool, abstraction]) for line_buffer
 -- called insert([tool, abstraction]) for line_buffer
 -- calling listen_after(o7@insert([tool, abstraction])) for
```

lines [[tool, abstraction], [data, abstraction], [visual,

Figure 9. Visual debugging of ViTaBAL systems.

We are adapting the timing diagram views provided by the Cerno-II program visualisation system [9] to visualise event propagations through toolies. We are also developing techniques to visualisie concurrently executing toolies, including producing timing diagrams with multiple propagation threads. Another area of work is editable timing diagram views, used to specify toolie concurrent execution and serialisation. These will be kept consistent with graphical and textual view modifications and will be used to ensure correct toolie event flows.

7. Discussion

ViTABaL provides many advantages over data abstraction-based textual and visual languages for designing and implementing large systems with evolving functionality. It is much easier to reuse toolies (units of functionality) as they can be linked to a variety of different toolies in different situations. Their behaviour can even be modified without altering the original response code, by using listen before/after connections. This has the added advantage that as systems evolve no extra code need be built into toolies or ADSs that doesn't relate directly to their purpose. This is in contrast to data abstraction-based languages, which require such additional code in order to retain efficient solutions.

As ViTABaL generates event broadcasting/request calls as necessary, programmers don't have to build special structures to support listened before/after and concurrent event propagations. The kind of code generated by ViTABaL is difficult to write directly in an error free fashion using Snart or any other data-based abstraction language. This is because programmers must take into account possible concurrency or listening by other toolies. To allow for the range of possibilities these data abstraction-based language solutions either become inefficient or inflexible compared to ViTABaL.

As ViTABaL allows complex events and data values to flow down toolie interconnections, ViTABaL event propagation views tend to be higher-level and less complex than dataflow visual languages, such as Prograph [3] and Fabrik [12]. A single event flow in or out of a ViTABaL toolie would be represented as several data flow pins in these languages. For example, consider the incremental KWIC sorter toolie which uses multi-argument insert and search events on the shifted list buffer ADT. To model this in Prograph would require a view specifying the source of the individual data values, each flowing into a process modelling the ADT. This results in more complex views than ViTABaL's.

ViTABaL event response ports allow a toolie to be reused more readily than Fabrik's typed data flows or Prograph's inter-object method calls. A toolie need only receive and respond to a valid event name and argument data types, no matter what toolie this was sent from. ViTABaL event propagation views specifying concurrent toolie operation are less complex than those of other visual languages, such as Meander [18], as ViTABaL generates the concurrent toolie event propagation code.

ViTABaL offers many advantages over other TA approaches, particularly its visual event propagation and program visualisation views. The visual nature of ViTABaL event specifications allows programmers to construct TA-based systems by interacting directly with toolies and event flows, rather than working with textual equations or grammars. Other languages supporting TA construct these connections via dependency analysis (attribute grammars, [17]), patterns and rule resolution (production systems, [2]), or active data rules (Smalltalk MVC, [13]). These are much more difficult to visualise, both statically and dynamically, than are ViTABaL event propagation and execution views. ViTABaL also supports easier definition of complex systems via multiple views (both hierarchical and multiple perspectives), together with visual debugging.

ViTABaL allows listen_before connections to be simply serialised. It also allows concurrent toolie execution to be easily specified. These are possible due to the visual, high-level nature of event propagation views. ViTABaL generates non-TA language code, in the form of Snart programs, which can execute more efficiently then other TA implementation approaches. This is because where toolie events are not concurrent or listened to, ViTABaL does not have to generate synchronisation or listener code. In contrast, systems using attribute grammars, active data and rule-based systems always include extra dependency management code, even if it is not required by parts of the system.

Reuse and modification of ViTABaL specifications are both more readibly performed than with other TA approaches. Toolie instances can be reused in multiple systems with their port bindings re-specified to different toolie types. Their behaviour can also be modified via listen before/after events. Changing between batch and incremental processing is straightforward, necessitating a change in only the event response of a single toolie, and not the toolies and ADSs that it is connected to.

Acknowledgements

John Hosking acknowledges the financial assistance of the Auckland University Research Committee in pursuing this research. Both authors acknowledge the helpful comments of their colleagues Rick Mugridge and Jim Kiper in the preparation of this paper.

References

[1] Amor, R., Augenbroe, G., Hosking, J.G., Rombouts, W., and Grundy, J.C., "Directions in modelling environments," to appear in *Automation in Construction*, 1995.

[2] Balzer, R., "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, vol. 11, no. 11, 1257-1268, November 1985.

[3] Cox, P.T., Giles, F.R., and Pietrzykowski, T., "Prograph: a step towards liberating programming from textual conditioning," in *Proceedings of the 1989 IEEE Workshop on Visual Languages,* IEEE CS Press, 1989, pp. 150-156.

[4] Garlan, D., Kaiser, G.E., and Notkin, D., " Using Tool Abstraction to Compose Systems," *COMPUTER*, vol. 25, no. 6, 30-38, June 1992.

[5] Glinert, E.P., and Tanimoto, S.L., "PICT: An interactive, graphical programming environment," *COMPUTER*, vol. 17, no. 11, 7-25, November 1985.

[6] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Environment.* Reading, MA: Addison-Wesley, 1984.

[7] Grundy, J.C. and Hosking, J.G., "A framework for building visual programming environments," in *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE CS Press, 1993, pp. 220-224.

[8] Grundy, J.C. and Hosking, J.G., "Constructing Integrated Software Development Environments with Dependency Graphs," Working Paper, Department of Computer Science, University of Waikato, 1994.

[9] Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B., *Visual Object-Oriented Programming.* Burnett, M, Goldberg, A., Lewis, T. Eds, Manning/Prentice-Hall, 1995, chap. 11.

[10] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Supporting flexible consistency management via discrete change description propagation," Working Paper, Department of Computer Science, University of Waikato, 1995.

[11] Grundy, J.C. and Venable, J.R., "Providing Integrated Support for Multiple Development Notations," in *Proceedings of CAiSE'95,* LNCS, Springer-Verlag, Finland, June 1995.

[12] Ingalls, D., Wallace, S., Chow, Y.Y., Ludolph, F., and Doyle, K., "Fabrik: A Visual Programming Environment," in *Proceedings of OOPSLA '88,* ACM Press, 1988, pp. 176-189.

[13] Krasner, G.E. and Pope, S.T., "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, 8-22, 1988.

[14] Lyons, P., Simmons, C., and Apperley, M., "HyperPascal: Using visual programming to model the idea space," in *Proceedings of the 13th New Zealand Computer Society Conference,* Auckland, August 1993, pp. 492-508.

[15] Mugridge, W.B., Grundy, J.C., Hosking, J.G., and Amor, R., *Snart94 Reference/User Manual*, Department of Computer Science, University of Auckland, 1995.

[16] Parnas, D.L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, 1053-1058, December 1972.

[17] Reps, T. and Teitelbaum, T., "Language Processing in Program Editors," *COMPUTER*, vol. 20, no. 11, 29-40, November 1987.

[18] Wirtz, G., "A Visual Approach for Developing, Understanding and Analyzing Parallel Programs," in *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE CS Press, 1993, pp. 261-266.