Linguistic Support for the Evolutionary Design of Software Architectures [*]

T.C. Nicholas Graham Tore Urnes
Department of Computer Science
York University
4700 Keele St., North York
Canada M3J 1P3
{graham,urnes}@cs.yorku.ca

Abstract

As a program's functionality evolves over time, its software architecture should evolve as well, so that it continues to match the program's design. This paper introduces the architecture language of Clock, *a language for the development of interactive, multi-user applications. This architecture language possesses three properties supporting the easy restructuring of software architectures: restricted scoping supported by a constraint-based communication system, automatic message routing, and easy hierarchical restructuring of architectures. Clock's architecture language has a visual syntax, supported by the* ClockWorks *programming environment.*

1 Introduction

Garlan and Perry describe the process of developing a software architecture as "[exposing] the dimensions along which a system is expected to evolve", and identifying the system's "load-bearing walls" [3]. Implicit in this analogy is that the internals of the architecture's components may evolve over time, but that changing the system decomposition or the interfaces between architecture components is to be avoided.

Evolution of software architectures is, however, important for many kinds of software. As software is moved from one organization to another, or as requirements change over time, an initial software architecture may become cumbersome and inappropriate to the software's evolving functionality [21]. As an extreme case, *interactive* systems are developed using the process of *iterative design* [17], where the design of the software evolves through iterations of user testing and redesign. To support evolution of the functionality of an existing program, architecture languages should therefore support the evolution of the program's architectural structure.

Support for the evolution of software architectures must come at two levels. First, the programmer must be motivated to provide architecture information and to keep it up to date. This means that architecture information must not be just documentation, but must also improve the implementation of the program. Secondly, the architecture language and its supporting tools must permit easy and rapid modification of the architecture structure.

This paper shows how support for evolution can be integrated into a software architecture language. These ideas have been implemented in *Clock* [5], a language for the development of interactive systems, including distributed multi-user systems and groupware. Clock is supported by the visual *ClockWorks* [6] programming environment, which allows the development, refinement and execution of Clock architectures, and provides linkage to a library of predefined Clock components.

Information provided in Clock architectures is used to automatically provide distributed implementations of multi-user applications and to optimize incremental display updating. The time that programmers spend on developing and refining software architectures therefore leads to time saved in network programming and tuning.

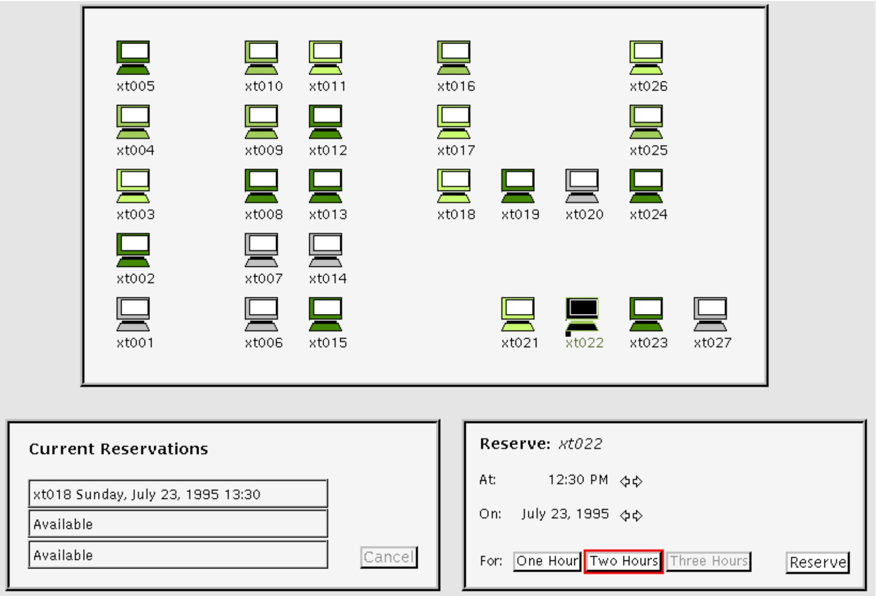
Through our experience with Clock, we have identified a desired list of properties of architecture languages to help them better support the evolutionary development of software architectures. These are:

Restricted communication among components: A language should provide scoping rules restricting the visibility of components. Restricting visibility reduces the potential for direct dependencies among components, which in turn reduces the impact of modifying or replacing components. In Clock, very restrictive scoping rules are made possible by integrating a constraint system into the architecture language.

Automatic routing of communication: As architectures evolve, components may be split, removed, or merged with other components. In order to

---

[*] To appear in ICSE'18, March 1996



Current Reservations

xt018 Sunday, July 23, 1995 13:30

Available

Available

Reserve: *xt022*

At: 12:30 PM

On: July 23, 1995

For: One Hour [✓ Two Hours] Three Hours

Figure 1: A terminal reservation system implemented in Clock.

localize the effects of such changes, components should not have to name the recipient of messages they send (or of other communication they initiate.) Thus, when a component is changed, the users of that component will not have to be changed. In Clock, delegation is used to automatically route messages to the correct component.

Easy restructuring of abstraction hierarchies: Software architecture languages typically provide abstraction by allowing components to be grouped in a hierarchical fashion. During evolution of architectures, it should be easy to break apart such hierarchies and refashion them to better reflect the intended structure. The ClockWorks environment provides simple facilities for grouping and ungrouping sets of components.

The paper is organized as follows. The next section introduces the problem of architectural evolution by showing how an example program can evolve due to changing requirements. Section 3 introduces the Clock architecture language on which this work is based. Section 4 shows how integrating a constraint mechanism into the architecture language permits the introduction of restrictive scoping rules and automatic message routing, reducing the impact of architectural change. Finally, section 5 shows how the architecture language can support easy hierarchical restructuring.

Although the work described in this paper was carried out in the context of the Clock language, these properties are not intended to be specific to Clock or to the specific application domain of interactive system development.

2 Architecture Evolution

To help motivate the problem of supporting evolution in software architectures, figure 1 presents an example system implemented in Clock. This example implements a terminal reservation system, in which students are permitted to reserve three one-hour slots per week. As seen in the figure, a terminal map shows the layout of terminals in the terminal room. A colour coding scheme is used to show how long each terminal is available, ranging from grey to indicate unavailable, to dark green, indicating the terminal is available for three hours or more. To reserve a terminal, a student clicks on the terminal to be reserved, selects a duration from one to three hours, and clicks on the *"Reserve"* button.

Figure 2 shows a view of the Clock architecture for this program (more detailed views will be presented throughout the paper). The architecture tree represents the compositional structure of the system, where for example the *TRSView* component is composed of the three components *ReservationPanel*, *ReservationButtons* and *Terminals*. As shown in the figure, these components in turn implement the display of the current reservations, the buttons for making reservations, and the pictorial view of a room of terminals.

Over time, a system such as this one might need to be modified in response to changing requirements. For example the number of terminals available for reservation might become too large to be conveniently represented as a map of a terminal room. The program

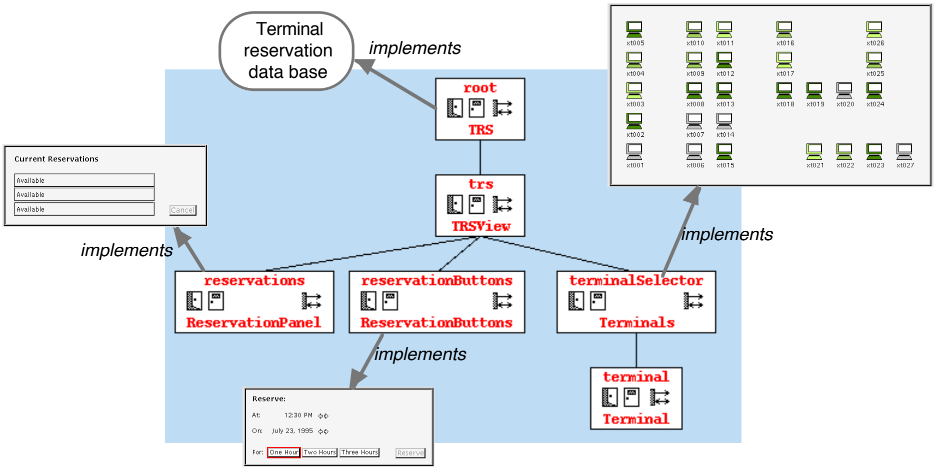
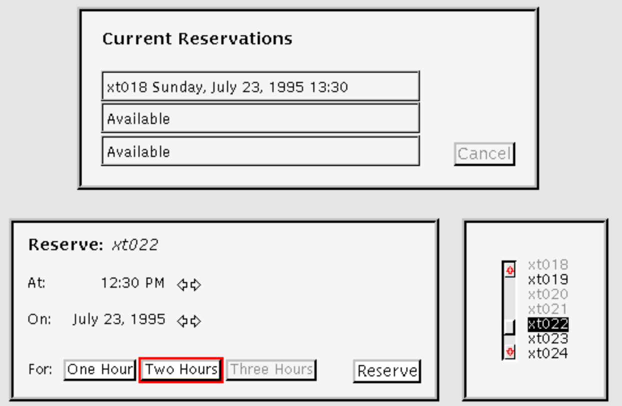


Figure 2: The Clock architecture of the terminal reservation system of figure 1.



Current Reservations

xt018 Sunday, July 23, 1995 13:30

Available

Available

Reserve: *xt022*

At: 12:30 PM

On: July 23, 1995

Figure 3: The terminal reservation system of figure 1 is modified to use a text browser rather than a map.

might be modified to the version of figure 3, in which a scrollable list of terminals is provided. In this version, only the terminals which are available for the selected duration can be reserved; all other terminals are greyed out.

Ideally, since the change in functionality of this version is limited to the way in which the list of terminals is presented, the modification should be as simple as replacing the *Terminals* component with a new component implementing the terminal browser. This updated architecture is shown in figure 4. In fact, the *ReservationPanel*, *ReservationButtons* and *Terminals* components are highly interdependent, potentially spreading the ramifications of the change over many parts of the program. To understand these interdependencies, consider the effect of cancelling the reservation of terminal *“xt018”* from figure 1:

- The *TRS* component must be informed of the cancellation;

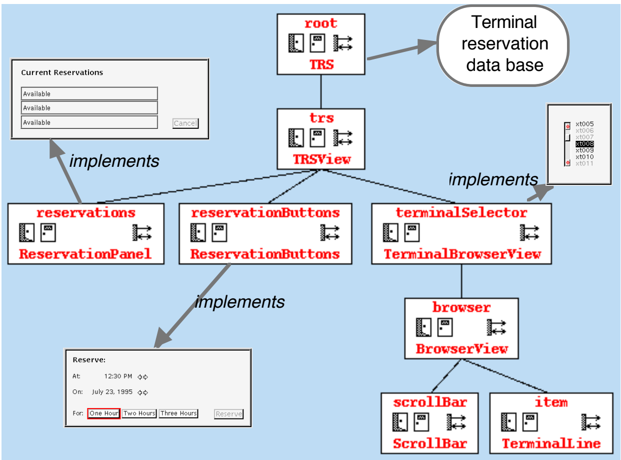


Figure 4: The Clock architecture of the modified terminal reservation system of figure 3.

- The colour of the *“xt018”* terminal must be changed to indicate that the terminal is now free for a longer period of time;
- The inactive *“Three Hours”* button must be reactivated, since the user now has three free reservation slots available instead of just two.

A survey of existing user interface software has concluded that such interdependence among components leads to a “spaghetti” program structure, providing a “maintenance nightmare” [13].

Clock’s architecture language so localizes the effects of change that the modification is as simple as changing the architecture from the version of figure 2 to that of figure 4. Following an overview of the Clock architecture language, the remaining sections show how Clock’s architecture language localizes the effects of

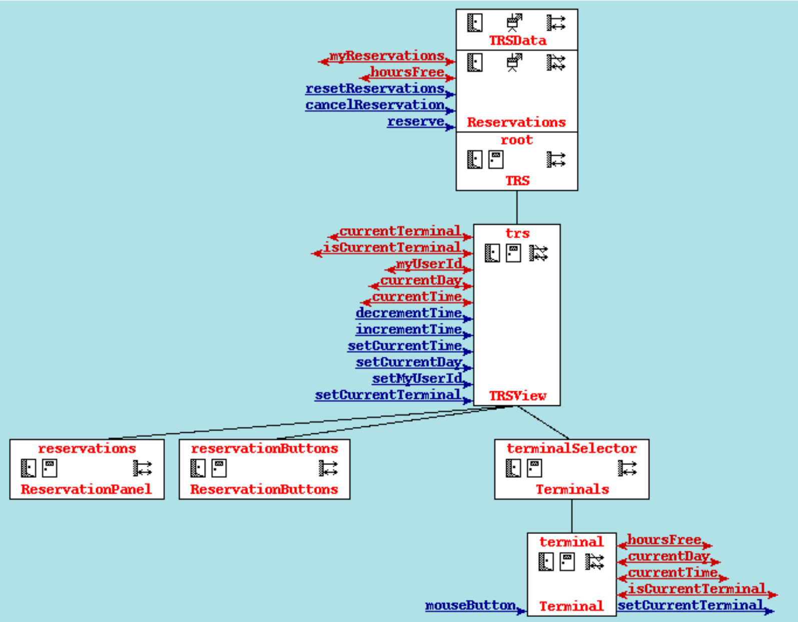


Figure 5: A more detailed view of the architecture of the terminal reservation system from figure 1.

architectural change.

3 The Clock Architecture Language

The Clock architecture language was designed to support the evolutionary design of architectures for interactive, multi-user systems. As discussed in section 1, the language provides the properties of restricted scoping to reduce direct dependencies among components, automatic message routing via delegation, and easy modification of the hierarchical structure of architectures. Clock is supported by the visual ClockWorks programming environment.

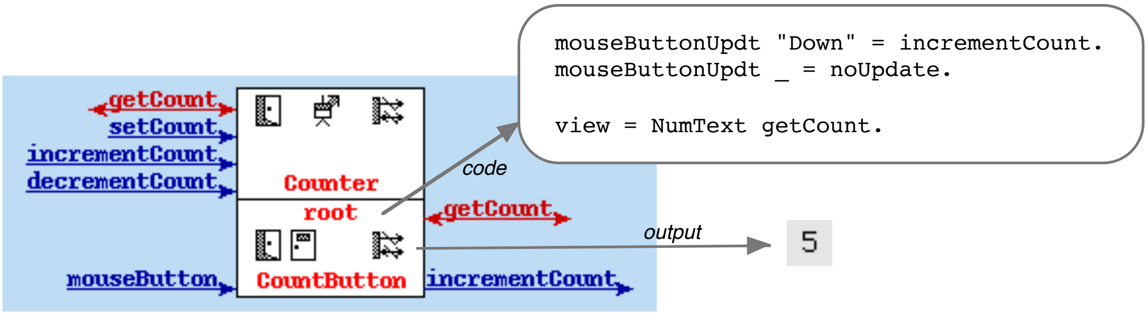
As shown in the more detailed view of figure 5, Clock architectures consist of a tree of communicating components. Components may respond to user input (or input from other components), and may produce graphical output to be placed on the display. Components may also contain instances of abstract data types (*ADT's*), such as the the *Reservations* data base attached to the *TRS* component. Each component has a name (e.g., *trs*) and a class (e.g., *TRSView*.)

Components may be grouped to form higher-level components. The contents of groups can be seen by opening them. The *open door* (“▯”) icon opens groups to reveal more detail. The *close door* (“▢”) icon elides detail. Completely opening a component invokes an editor for the program code. Therefore, ClockWorks is a complete programming environment for Clock, allowing architectures and code development, and program execution.

The *interface* of a component can be shown by toggling the *interface* (“⇆”) icon. Components communicate via *input*, *request* and *update* messages. For example, the *Terminal* component is capable of responding to the *mouseButton* input (generated when a user clicks on a terminal picture), may make the *hoursFree* request (to find out how many hours a terminal is free), and may issue the *setCurrentTerminal* update (to specify that a terminal has been selected by the user). In general, the arrows on the left side of a component indicate the messages the component may receive, while the arrows on the right side indicate the messages the component may issue.

Messages are automatically routed up the tree to the nearest component capable of handling them. This routing mechanism provides a form of inheritance by delegation, where components inherit all the facilities of the components appearing above them in the tree. This communication mechanism means that components can only access the data of components appearing above them – components may not directly communicate with their children or siblings. Section 4 shows how a constraint mechanism built into the architecture language allows components to communicate indirectly, allowing fully general communication without the need for explicit communication links.

Components in Clock are implemented using a scripting language similar to the functional language Haskell [9]. The architecture language, however, does not depend on the implementation language used for components – an earlier version of Clock was based on Turing [8], a Pascal-like language.



```
mouseButtonUpdt "Down" = incrementCount.
mouseButtonUpdt _ = noUpdate.

view = NumText getCount.
```

Figure 6: Simple example of a constraint. When the user clicks on the displayed number, the count is incremented. The view is then automatically recomputed to reflect the changed value of *getCount*. The view is therefore *constrained* to display the current value of the counter. The code shown is the complete implementation of the *CountButton* component.

4 Communication via Constraints

Clock's architecture language imposes restrictive scoping rules, reducing the ways in which components can directly communicate. These restrictions have the positive effect of reducing the number of direct dependencies among components, in turn reducing the impact of architectural change. In order to allow components to indirectly communicate, Clock embeds a constraint mechanism into its architecture language. First, we shall introduce a simple example of these architectural constraints, and then show how the mechanism can be used to simplify the communication between the components of the terminal reservation system.

4.1 Simple Constraint Example

Figure 6 shows a simple program written in Clock. The output of the program is an integer number appearing on the screen. When the user clicks on the number, it is incremented.

The architecture for this program consists of a single component, *root* of class *CounterButton*. This component makes use of a *Counter* ADT taken from the Clock library. The *Counter* maintains an integer value, which can be incremented, decremented or queried. *CounterButton* uses the *incrementCount* update to increment the counter, and the *getCount* request to query the counter value. *CounterButton* also takes the *mouseButton* input, in order to respond to the user's mouse clicks.

The code for *CounterButton* is simple: the *mouseButtonUpdt* function specifies that when the user clicks on the number, the number is to be incremented:

```
mouseButtonUpdt "Down" = incrementCount.
mouseButtonUpdt _ = noUpdate.
```

(The second line of this function definition specifies that any mouse button update other than *"Down"* results in no update.) The *view* function specifies that the component's display view is to be the current value of the counter (as obtained via *getCount*), displayed as numeric text:

```
view = NumText getCount.
```

This view function is a *constraint* in that its value is always automatically updated when the counter's value changes (and hence the value of *getCount* changes.) This constraint has the effect that the displayed number is always the value of the counter, without the programmer having to explicitly update the display. As we shall see, constraints provide a mechanism for components to respond to changes in other components without explicit communication.

4.2 Constraints and the TRS

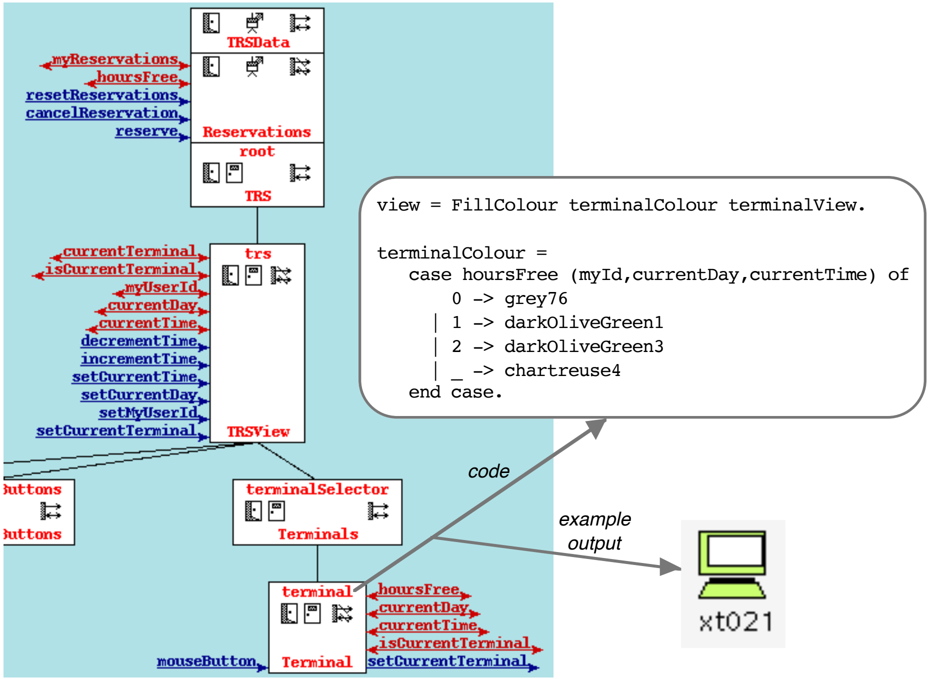
Figure 7 shows how the terminals in the terminal room are drawn. Here, the view function states that each terminal is to be drawn as a *terminalView*, filled with the appropriate *terminalColour*:

```
view = FillColour terminalColour terminalView.
```

The *terminalView* function (not shown here) implements the picture of a terminal. The *terminalColour* function specifies the colour of the terminal based on the number of hours for which it is free:

```
terminalColour =
  case
    hoursFree (myId,currentDay,currentTime)
  of
     0 -> grey76
   | 1 -> darkOliveGreen1
   | 2 -> darkOliveGreen3
   | _ -> chartreuse4
  end case.
```

Here, *hoursFree* is a request, taking the three parameters of the terminal id (*myId*), and the day and time the user has selected (*currentDay* and *currentTime*).



```
view = FillColour terminalColour terminalView.

terminalColour =
   case hoursFree (myId,currentDay,currentTime) of
       0 -> grey76
     | 1 -> darkOliveGreen1
     | 2 -> darkOliveGreen3
     | _ -> chartreuse4
   end case.
```

Figure 7: An example of the use of constraints in the terminal reservation system of figure 1. The colour of the terminals is constrained to depend on the current day and time, and the amount of time for which the terminal is currently free. Therefore, components that change the day or time, or make or cancel reservations do not have to directly communicate with the *Terminal* component.

As shown in figure 7, *hoursFree* is implemented in the *Reservations* ADT, and *currentDay* and *currentTime* are requests implemented in the *TRSView* component. This means that the terminal's colour is *constrained* to these three values of date, time and hours the terminal is free: if any of these values changes, the colour on the display is automatically updated.

Figure 8 shows how these constraints allow indirect communication between components. For example, if the user advances the time using the arrow buttons in the reservation buttons display, the *DayTimeSelector* component issues the update *incrementTime*, which is handled by the *TRSView* component. Incrementing the time changes the value of *currentTime*, therefore automatically triggering the recomputation of the colour of all terminals, and the redisplay of those terminals whose colour has changed.

This example shows how constraints reduce the direct dependencies among components. When the user advances the time, somehow the *DayTimeSelector* component must inform the *Terminal* component that the terminal colours may be out of date. By using a constraint, the need for this update is automatically inferred by the Clock run-time system. Therefore, the *DayTimeSelector* and *TRSView* components need have no direct knowledge of the existence of the *Terminal* component. This means that when the programmer replaces the terminal map functionality with the browser version, it is guaranteed that there will be no direct references to *Terminal* in any other component in the system.

4.3 Modifying the Architecture

Figure 9 shows how the architecture can be modified to implement the browser version of the terminal reservation system. The *Terminals* component is replaced by a tree of components implementing the scrollable list of terminals. In this version, the component *TerminalLine* uses constraints to determine the appearance of each line in the browser. The complete view function from *TerminalLine* is:

```
view =
  let terminalLine = Text myId in
    if hoursFree (myId,currentDay,currentTime)
        < duration then
      TextColour inactiveColour terminalLine
    elsif isCurrentTerminal myId then
      inverted terminalLine
    else
      terminalLine
    end if
  end let.
```

The basic view of each line is simply the terminal id as a text string (“`Text myId`”). If the duration

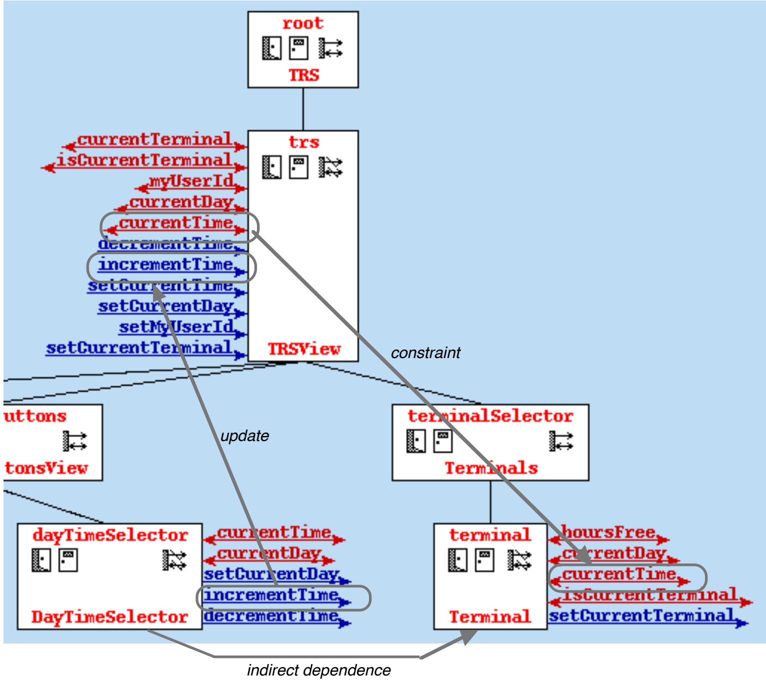


Figure 8: Example of how constraints reduce direct dependencies among components. Here the *DayTimeSelector* modifies the current time (via the *incrementTime* update.) The *Terminal* component depends on the *currentTime* request (also defined in the *TRSView* component) via a constraint. When the time is incremented, the system automatically infers that an update to the view of terminal may be required.

the user has selected is longer than the number of hours the terminal is available (“`hoursFree (myId, currentDay, currentTime) < duration`”), then the terminal id is greyed out. If the terminal is the currently selected one (“`isCurrentTerminal myId`”), it is drawn inverted (“`inverted terminalLine`”). Since this component is connected into the architecture via constraints, it is not necessary to modify the existing architecture components in order to implement its functionality.

Note that in order to make the *duration* request available to the *Terminal* component, the implementation of the request was moved up to the *TRSView* component. The next section will detail how this was accomplished.

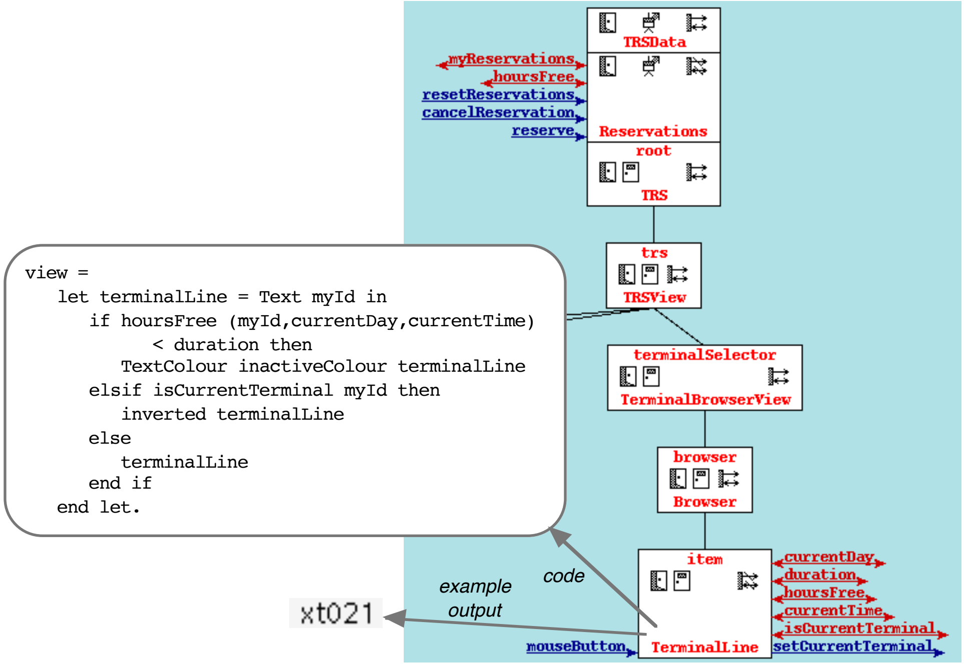
5 Restructuring of Architectures

While constraints significantly simplify the modification of architectures, not all modifications are as simple as plug-replacing components. To support more complex evolution, the architecture language must support easy restructuring of architectures. In Clock, we have found two techniques to be crucial in aiding restructuring: communication by delegation, and easy grouping and ungrouping of components.

5.1 Delegation for Automatic Routing

As architectures evolve, it is common to move ADT’s, split or combine components, and otherwise change the locations of where updates and requests are handled. In typical architecture descriptions, communication is bound to specific components. I.e., it is necessary to specify exactly where a method call is directed. This form of explicit targeting makes code less robust to change, since as components evolve, the method may no longer be handled by the same component. In Clock, requests and updates are automatically routed to the nearest component above the issuer of the message that is capable of handling it. This mechanism is a form of inheritance *by delegation*. This automatic routing of messages means that as the implementations of messages evolve, the code that uses them does not have to be modified. In the *Duration* example above, the only operation required to move the location of the *Duration* ADT is to drag it up the tree – none of the code using the messages defined in the ADT needs to be modified.

The use of delegation to implement routing is only possible due to Clock’s restricted visibility rules, which are in turn made possible by the use of constraints in the architecture language. If components were allowed to communicate arbitrarily, then some form of explicit routing would be required.



```
view =
   let terminalLine = Text myId in
      if hoursFree (myId,currentDay,currentTime)
            < duration then
         TextColour inactiveColour terminalLine
      elsif isCurrentTerminal myId then
         inverted terminalLine
      else
         terminalLine
      end if
   end let.
```

Figure 9: The modified architecture supporting the modified terminal reservation system of figure 3.

5.2 Hierarchical Restructuring

As with most architecture languages, Clock allows components to be grouped together to form higher-level components. As was shown in section 3, this grouping mechanism is used extensively in our example architectures.

Many tools provide easy support for creating groups, but not necessarily for modifying them. As architecture structures evolve, however, it must be possible for group structures to easily evolve with them.

A typical use of group restructuring is shown in figure 10. This example shows how groups can be created to help create a simpler program structure, and even to provide reusable components for later use. When the programmer created the scrollable list of terminal lines, the browser was composed of a set of terminal lines, and a scroll bar. The programmer noticed that the browser and the scroll bar logically belonged together, as a scrolling browser. To group the browser view and the scroll bar, the programmer first selects them, and then groups them. The resulting *Browser* component is a browser over arbitrary display objects (in this case, terminal id's). By performing the grouping operation, the programmer has cleanly separated the browsing capability from the particular details of a terminal id browser. By replacing the *TerminalLine* component, this browser can be used as a browser for any other textual or graphical data.

This form of grouping cannot necessarily be performed in advance. Since programmers don't know in advance exactly what the functionality of the program will be, it is not possible for them to know what grouping abstractions will be appropriate.

Grouping of this form can be used to implement a visual form of architecture *pattern* [1]. The *Browser* component is a pattern for how to implement a browser. When instantiated in a program, the programmer must provide some component to be the items over which the browser operates. Such patterns from the library can be treated as black boxes, where the component is simply instantiated and the necessary children components filled in; otherwise, the component can be ungrouped and customized to the particular application. In practice, Clock programmers commonly use both forms of reuse of architecture groups.

6 Related Work

The design of the Clock architecture language and the supporting ClockWorks environment owes much to earlier work in software architecture languages and environments. This section reviews this earlier work, based on the desired properties of software architecture languages that we identified earlier.

The notion of using constraints to reduce the direct dependencies among components is not new to Clock. The first version of the idea was seen in the Smalltalk MVC model for user interface development [10]. In MVC, callbacks are used to automatically trigger view

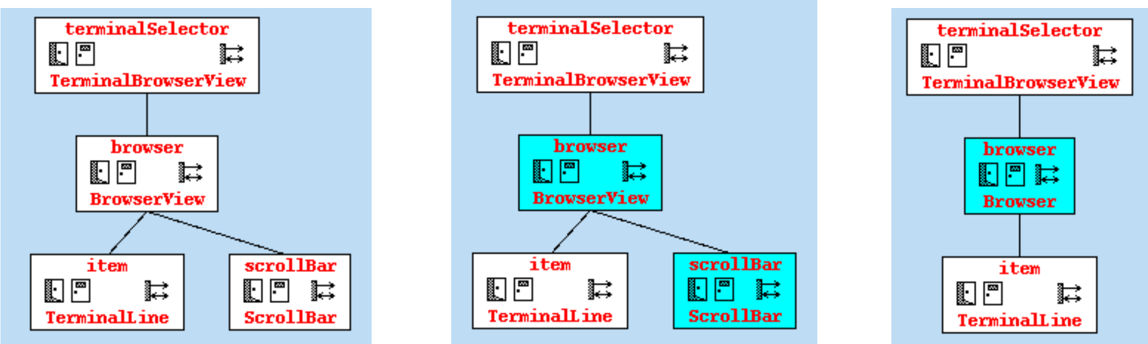


Figure 10: The *BrowserView* and *ScrollBar* components are grouped to create a new *Browser* component.

recomputation following modification in the state of an underlying *model*. The idea of using constraints as programming language constructs has been extensively developed in the Garnet [15] and RendezVous [7] systems. Garlan and Scott have also demonstrated how a constraint-like mechanism can be introduced into the module systems of traditional languages such as Ada [4]. Clock differs from these systems in that the constraint mechanism is tightly bound into the architecture language, not the underlying programming language. While these earlier approaches permit components to communicate in arbitrary ways, Clock's scoping rules imply what communication may be done directly, and what communication should be performed using constraints. These scoping rules lead to the ease of modification of Clock programs.

Other environments build knowledge of scoping into architecture development. One interesting approach is the *Star* system [11] which generates architecture editing and checking environments from the high-level specifications of scoping rules. The PegaSys environment [12] exploits hierarchical architectural structure to support the formal synthesis of architectures.

Clock's ability to perform automatic routing of messages made possible by the architecture language's restrictive scoping rules: if components are allowed to communicate arbitrarily, there is no automatic way of determining which component should handle a given message. Systems such as Garnet [14] and RendezVous [7] use the alternative mechanism of *pointer variables* [20] to allow indirect references to components. This mechanism allows components to communicate without explicit reference to the target component. Programmers must, however, explicitly maintain the values of indirection variables, meaning that the routing is not completely automatic.

Most tools for implementing architecture languages are not designed to support easy ungrouping and regrouping of architectural components. One notable exception is the *Darwin* system [16], which provides grouping and ungrouping mechanisms very similar to those of Clock. Darwin's approach is, however, strictly hierarchical, and does not allow the parameterization of component groups seen in the *Browser* example of figure 10. Advantages of Darwin over Clock are that multiple architectures can be edited concurrently and easily combined, and that both automatic and user-directed architecture layout are provided.

Numerous authors have pointed out that there are many styles of software architecture [2, 18] , and that architecture languages must support this heterogeneity [19]. Clock's architecture language, however, supports only a single style. To address this problem, Clock architectures can be treated as primitive units for composition using other architectural techniques. There is no reason, for example, why a complete Clock architecture could not be combined in a pipe and filter pattern with other architecture components. Another interesting topic for future research would be to see how the techniques from Clock's architecture language could be integrated into more traditional architecture languages.

7 Conclusion

This paper has argued that software architectures cannot be treated as static entities, unchanging over the life of a program. As the requirements of programs evolve, so will the architecture appropriate for implementing them. We have seen that this kind of evolution is particularly prevalent in interactive systems.

We have proposed that software architecture languages should have three properties to better help them support evolution: communication patterns should be restricted, communication should be automatically routed, and it should be easy to modify the hierarchical structuring of the program. These properties were demonstrated in the Clock architecture language. Clock shows how the integration of constraints into an architecture language reduces the direct dependencies among components, reducing the impact of architectural modifications. A visual programming environment supports easy hierarchical restructuring of Clock architectures.

Ongoing work with Clock is aimed at integrating support for distributed multi-media into the architecture language, and investigating how Clock architectures can be mapped to different forms of hardware architectures, including fully replicated architectures with ATM-based communication.

Acknowledgements

Clock and *ClockWorks* were developed by the authors, Catherine Morton, Roy Nejabi and Gekun Song. This work was carried out at the York Laboratory for Computer Systems Research, and was partially supported by the Natural Sciences and Engineering Research Council, the Information Technology Research Centre, and the Royal Norwegian Research Council. This paper benefited from helpful comments from Judy Brown.

References

[1] Kent Beck and Ralph Johnson. Patterns generate architectures. In *Proceedings of ECOOP '94*, pages 139–149, 1994.

[2] Thomas R. Dean and James R. Cordy. A syntactic theory of software architecture. *IEEE Transactions on Software Engineering*, 21(4):302–313, April 1995.

[3] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1995.

[4] David Garlan and Curtis Scott. Adding implicit invocation to traditional programming languages. In *Proceedings of the 15th International Conference on Software Engineering*, pages 447–455. IEEE Computer Society Press, April 1993.

[5] T.C. Nicholas Graham. *Declarative Development of Interactive Systems*, volume 243 of *Berichte der GMD*. R. Oldenbourg Verlag, July 1995.

[6] T.C. Nicholas Graham, Catherine A. Morton, and Tore Urnes. Clockworks: Visual programming of component-based software architectures. *Journal of Visual Languages and Computing*, July 1996. (To appear).

[7] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. The Rendezvous language and architecture for constructing multiuser applications. *ACM Transactions on Computer-Human Interaction*, 1(2):81–125, June 1994.

[8] Richard C. Holt and James R. Cordy. The Turing programming language. *Communications of the ACM*, 31(12):1410–1423, December 1988.

[9] Paul Hudak and Philip Wadler. Report on the functional programming language Haskell (v1.1). Technical Report YALEU/DCS/RR777, Yale University, August 1991.

[10] Glen E. Krasner and Stephen T. Pope. A cookbook for the using the Model-View-Controller interface paradigm. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[11] Spiros Mancoridis and Richard C. Holt. Extending programming environments to support architectural design. In *Proceedings of CASE'95*, pages 110–119. IEEE Computer Society Press, July 1995.

[12] Mark Moriconi and Dwight F. Hare. Pegasys: A system for graphical explanation of program designs. In *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 148–160, July 1985.

[13] Brad A. Myers. Separating application code from toolkits: Eliminating the spaghetti code of callbacks. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, pages 211–220. ACM Press, November 1991.

[14] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[15] Brad A. Myers, Dario A. Guise, and Brad Vander Zanden. Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. In *Proceedings of OOPSLA'92*, pages 184–200. ACM Press, October 1992.

[16] Keng Ng and Jeff Kramer. Automated support for distributed software design. In *Proceedings of CASE'95*, pages 381–390. IEEE Computer Society Press, July 1995.

[17] Jakob Nielsen. *Usability Engineering*. AP Professional, Cambridge, MA, 1993.

[18] Mary Shaw, Robert DeLine, Daniel V. Kline, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[19] Mary Shaw and David Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, School of Computer Science. Carnegie Mellon University, December 1994.

[20] Bradley T. Vander Zanden, Brad A. Myers, Dario Giuse, and Pedro Szekely. The importance of pointer variables in constraint models. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, pages 155–164. ACM Press, November 1991.

[21] M. Wein, S.A. MacKay, D.A. Stewart, C.-A. Gauthier, and W.M. Gentleman. Evolution is essential for software tool development. In *Proceedings of CASE'95*, pages 196–205. IEEE Computer Society Press, July 1995.