7

The integration of a layout constraint language with an object-oriented user interface management system

K. Korfmacher, H.-W. Six and J. Voss
Praktische Informatik III, FernUniversität Hagen
D-58084 Hagen, Germany
klaus.korfmacher@henkel.dbp.de
{hw.six, josef.voss}@fernuni-hagen.de

Abstract

In this paper we present the COMPASS constraint language which integrates high-level layout specifications with the DIWA user interface management system. The general concept of the COMPASS system is the automatic adjustment of constraints according to their high-level specification such that a programming interface to explicitly add, remove or modify constraints is unnecessary. In particular, COMPASS provides the following features to support modular and declarative specifications of changing constraints: association of constraints with classes, a limited reach of constraints by imposing rules on the set of variables which may be used as source or target of a constraint, temporary and conditional constraints, and finally, the concept of a constraint schema which specifies a bundle of similar constraints. At runtime the number of constraints contained in such a bundle varies, i.e. constraints are automatically created or deleted, and each constraint can change its source variables. Declarativeness of constraint schema specifications is achieved by the use of high-level object expressions.

Keywords

Constraints, user interface layout, user interface management systems

1 INTRODUCTION

The usefulness of constraints as part of user interface development systems has been described many times. Constraints support layout computation of user interface objects (UIOs), consis-

L. J. Bass et al. (eds.), *Engineering for Human-Computer Interaction*

tency maintenance between UIOs and application objects, as well as consistency between related UIOs. The power of constraints lies in their declarative nature: the developer denotes conditions to be maintained by the constraint system and needs not care about when to repair constraint violations.

However, problems arise with large sets of constraints and with runtime changes. Large sets are hard to understand since the overall consequences of a variable change are potentially unlimited. In this situation a modular structure with well-defined interfaces is desirable, such that the propagation of value changes can be followed up. In addition, changes of constraints, i.e. their creation, deletion, and modification, are inevitable due to runtime changes in the UIO structure. Changes in the constraint set are also the result of shifting initiative: from the user's changing variables to application changes and back. A desirable solution to this problem would allow the developer to specify changing constraint sets just as declarative as the constraints themselves.

In this paper we address both issues, modular structure and high level specification of changing constraint sets. To this end, we describe the COMPASS constraint language and its integration into the DIWA user interface management system (Six, 1990). First of all, COMPASS as part of the DIWA system is restricted to layout computation. Dialog behavior and communication between UIOs are specified using DIWA's event handler language. Data exchanges and notifications between application objects and UIOs is carried out by an update/changed mechanism similar to that of Smalltalk's MVC (Krasner, 1988).

Layout computation however, turned out to be hard using conventional imperative programming techniques. In a hierarchical UIO structure layout computation typically deals with a complex mixture of bottom-up and top-down dependencies. For example, the size of a menu may depend on the number of its menu items and their sizes, or, the other way round, the position of a button may depend on the width of its surrounding form. To deal with such dependencies the original DIWA system contained a kind of simulated constraint system using backtracking. Based on these experiences we developed the COMPASS constraint system which is tightly tailored to the needs of layout computation for UIO hierarchies.

The general concept of the COMPASS system is the automatic adjustment of constraints according to their high-level specification such that a programming interface to explicitly add, remove or modify constraints is unnecessary. In particular, COMPASS provides the following features to support modular and declarative specifications of changing constraints:

- **Association of constraints with classes**: All constraints are part of a class definition, so every new instance is automatically equipped with the constraints of its class.
- **Limited reach of constraints**: By imposing rules on the set of variables which may be used as source or target of a constraint the coupling of constraints from different UIOs is reduced.
- **Temporary and conditional constraints**: The adjustment of constraints to changing situations is achieved by directly associating constraints with dialog states and condition variables.
- **Constraint schemata:** Changing constraint sets are furthermore supported by constraint schemata which specify bundles of similar constraints. At runtime the number of constraints contained in such a bundle varies, i.e. constraints are automatically created or deleted, and each constraint can change its source variables. Declarativeness of constraint schema specifications is achieved by the use of high-level object expressions.

2 RELATED WORK

Most of the work on constraints for user interfaces deals with solving strategies for elaborate constraint types, see e.g. (Sanella et al., 1993). However, there are some approaches to a higher level support. Closely related to our work are the systems RENDEZVOUS (Hill, 1993) and KALEIDOSCOPE (Freeman-Benson, 1990). Like COMPASS, both systems provide languages to associate constraint specifications with classes and some kind of automatic adjustment of constraints. The importance of such concepts has also been explained in (Vander Zanden et al., 1991).

KALEIDOSCOPE provides a mixture of declarative and imperative programming styles. It allows to temporarily add constraints, especially to support changing dialog behavior. Furthermore, it supports the simultaneous definition of constraints for array components, which can be seen as a limited form of a constraint schema. However, with regard to constraint modifications KALEIDOSCOPE is dominated by imperative concepts.

RENDEZVOUS provides source and target indirection, i.e. expressions to specify source and target variables of a constraint, such that the constraint is modified whenever these expressions change their values. However, this works only for simple expressions. More elaborate forms of indirection require the use of the programming interface. The system does not allow to specify constraint schemata yielding automatic creation or deletion of constraints.

3 BASIC DIWA CONCEPTS – UIO CLASSES AND ATTRIBUTES

DIWA and COMPASS are tightly coupled. So before going into details of the constraint language we have to introduce some basic DIWA concepts. The goal of the DIWA approach is twofold: on the one hand a model for a uniform software architecture of user interface objects is provided; on the other hand a high-level executable specification languages for UIO configuration and dialog behavior is provided.

The backbone of the architecture model is the hierarchical arrangement of UIOs. In many cases this hierarchy corresponds to geometric containment of the UIOs' screen rendering. In the following we will refer to the children of a UIO as **subobjects**. An important issue of the architecture is local responsibility of components: a UIO should work with minimal knowledge about other UIOs. Typically, a parent UIO is responsible only for its direct children. Another important issue is homogeneity of components which is a prerequisite to composability: according to the DIWA architecture every UIO is decomposed into one or more concurrently working event handlers, a presentation component and an application interface component. A discussion of the architecture is presented in (Six, 1992).

Within this paper presentation components are of special interest. They are responsible for the UIOs' screen rendering and layout computation. Presentation components are implemented by their own classes. Hence, these classes are the appropriate location for constraint specifications.

The building blocks of the DIWA specification language are UIO classes. The specification of a UIO class determines which components and subobjects an instance of the class will have.

The following example specifies a class Window as a subclass of class DialogObject. An instance of class Window will have three subobjects, its presentation component will be an instance of class WindowPresentation:

```
Window class [ DialogObject ]

    subobjects: [ InsideArea = DialogObject
                  Title = TextItem
                  CloseButton = CloseButton      ]

    presClass: WindowPresentation
    ...
end Window
```

To support dynamically changing UIO hierarchies the subobjects section of a UIO class can contain one or more **subobject sets**. Such a set is determined by a name and a base type for its elements (which is again a UIO class name). Consider for example a UIO class Tree with a subobject set NodeSet denoting the tree's nodes:

```
Tree class [ DialogObject ]
    subobjects: [ NodeSet = setOf Node ]
    ...
end Tree
```

Adding and removing set members is done by event handler actions, either explicitly by newInSet and deleteFromSet operations or implicitly by an updateSet operation which provides a hook for application methods to determine number and type of set members.

3.1 Attributes

Besides subobject and event handler definitions UIO classes contain lists of attribute descriptions. Attributes can be considered as a kind of exported instance variable of a UIO. Each attribute is specified by a name and a category, the latter determining the attribute's value type and usage. Actually these attribute categories form their own inheritance hierarchy. For example, an attribute of category BooleanDrawAttribute has values of type boolean with each value change automatically forcing a re-display of the UIO it belongs to. For another example, event handlers can express interest in attributes of category ActiveBooleanAttribute, so they are notified about value changes.

It is a common situation that attribute values associated with a UIO are primarily used by the UIO's parent. Consider for example an event handler setting marks to subobjects. It would be awkward to introduce such a marker attribute in all of the subobjects' UIO classes. To support this kind of attribute usage various categories of **subobject attributes** are provided. A subobject attribute can be considered as a dictionary which is located at a parent UIO and associates values with each of its subobjects. A subobject can read 'its' value of a parent's subobject attribute as if were its own attribute.

A special kind of subobject attribute, the category SubobLink, establishes relationships among the children of a UIO by associating with each subobject a set of its siblings. Extending the Tree example from above we can specify a SubobLink attribute Children to denote the parent-to-child relationship among the members oft the set NodeSet (However, note that with re-

gard to the UIO hierarchy all members of NodeSet are siblings):

```
Tree class [ DialogObject ]
        subobjects: [ NodeSet = setOf Node ]
        attributes: [ Selected = SubobBooleanAttribute
                      Children = SubobLink ... ]
        ...
end Tree
```

Attributes and subobject names form the basis of **object expressions** which provide a simple query facility to address UIOs, resp. their attributes. We illustrate this concept with an example: In the context of the Tree class from above the expression 'NodeSet (Selected, Children)' denotes the children of all selected tree nodes which is again a subset of NodeSet. In general, an object expression consists of a base subobject name which may denote an individual or a set, followed by a list of attribute names. The expression is evaluated with respect to a given UIO. Evaluation starts with the subobjects denoted by the base name. The evaluation of the attribute names consists of a sequence of filtering and navigation steps transforming the initial set into the resulting set. A boolean attribute has a filtering effect eliminating those subobjects whose attribute values are false. A link attribute yields a navigation step: each subobject is replaced by its related siblings.

Object expressions can have more than one level, thus allowing to address UIOs below the level of direct children. However, most of the time one-level and in some cases two-level expressions are used. This reflects our goal of local responsibility.

3.2 The use of attributes

Attributes and object expressions have been successfully used in the DIWA system for various purposes. An event handler uses object expressions to express interest in certain user actions, e.g. 'LeftDownIn (ItemSet (Enabled))', or may explicitly modify boolean attributes, e.g. 'setAttr (Mapped) to PropertyWin' to enforce the visibility of a UIO called PropertyWin, or send signals to other UIOs specified by an object expression. Application interface components can provide methods to compute attribute values according to application data. Presentation methods read attribute values to decide about a certain variant of the UIO's screen rendering, e.g. a push button's presentation checks the Selected attribute.

Attribute modifications depend on the attribute's category. Most important in the context of this paper is the use of attributes within constraints: in general all kinds of attributes can be either read, i.e. used as source variables, or modified, i.e. used as target variables. However, in our current implementation values of SubobLink attributes can only be used as source variables.

To support modularity, especially to reduce the coupling of constraints from different UIOs we impose rules on the access of attributes. These rules are either strong or weak:

- The strong rules: Since object expressions only address a UIO's direct and indirect descendents and its direct ancestor only attributes of these UIOs can be accessed. In addition, subobjects are not allowed to use their parent's SubobLink attributes.
- The weak rules: A UIO should only modify attributes specified in its own class. A UIO should not read attributes from the level of grandchildren or below.

4 CONCEPTS OF THE COMPASS LANGUAGE

4.1 Local layout specification

Our concept of layout constraints is influenced by the observation that in most cases layout computation can be handled locally, i.e. by the presentations of a UIO and its parent. The task of layout computation within the DIWA system can be described as follows: each UIO is equipped with its own coordinate system with the origin in the lower left-hand corner. With regard to this coordinate system the positions of the UIO's subobjects have to be computed. The lower left-hand corner of a subobject is taken as its position. We assume UIOs to have rectangular shapes, so their layout is determined by the four predefined attributes Left, Bottom, Width, Height (a UIO class inherits these attributes from the basic class DialogObject).

Locality of layout computation is supported by predefined attributes and constraints, determining a basic protocol. Besides the basic layout attributes Left, Bottom, etc. each UIO is equipped with corresponding subobject attributes Left_pro, Bottom_pro, Width_pro, and Height_pro[1] which are to be used by a UIO to determine proposal values for the layout of its subobjects. The subobjects themselves can accept these proposals, modify them, or compute completely new values. In any case, the subobjects' decision determines their actual layout.

4.2 Basic properties of the constraint language and solving system

The main purpose of this paper is the COMPASS language and its integration with the DIWA system. Therefore, we restrict the description of the underlying constraint system to those basic characteristics that directly interfere with the language:

- The COMPASS constraint system works with one-way constraints with all target variables being atomic. Hence, to compute a point two constraints have to be specified.
- Solving is explicitly initiated. On the one hand this prevents 'hyper-active' user interfaces. One the other hand unnecessary computations are prevented which is of special importance since constraint solving may create, delete or modify constraints.
- Cyclic dependencies can only be detected at runtime, i.e. during constraint solving. In this case a warning is given and the variables involved in the cycle are left unchanged.

4.3 A simple example

For a first example we present a simple window layout. The UIO class Window introduces the required subobjects and attributes:

---

1. Actually, there are more than four basic layout attributes. For example, instead of Left and Width the attributes Left and Right can be used. Width is automatically adjusted in this case. According subobject attributes for proposal values are also provided.

```
Window class [ DialogObject ]

    attributes: [ OuterBorder = NumberAttribute
                  InnerBorder = NumberAttribute
                  DefaultWidth = NumberAttribute
                  DefaultHeight = NumberAttribute ]

    subobjects: [ InsideArea = DialogObject
                  Title = TextItem
                  CloseButton = CloseButton      ]

    presClass: WindowPresentation
    ...
end Window
```

Layout computation for this example works as follows: Based on a given window size close button and window title are placed side by side in the top left corner of the window. These two subobjects are assumed to determine their sizes by themselves. Therefore, there are no constraints to compute proposal values for their width and height. The remaining space is given to the InsideArea subobject:

```
WindowPresentation: presClass [ Presentation  ]

dialogClass: Window

status: all

    Width = DefaultWidth
    Height = DefaultHeight

    InnerBorder = 4
    OuterBorder = 5

    CloseButton:Left_pro = OuterBorder
    CloseButton:Bottom_pro = Height - CloseButton:Height - OuterBorder

    Title:Left_pro = OuterBorder + CloseButton:Width + InnerBorder
    Title:Bottom_pro = Height - Title:Height - OuterBorder

    InsideArea:Left_pro = OuterBorder
    InsideArea:Bottom_pro = OuterBorder
    InsideArea:Width_pro = Width - 2 * OuterBorder
    InsideArea:Height_pro = Height - max ( CloseButton:Height, Title:Height)
                                   - InnerBorder - 2 * OuterBorder

end WindowPresentation
```

The following figure illustrates the situation. It displays those dependencies providing proposal values for position and shape of the window's InsideArea subobject. The interface of the Win-

dow UIO to its subobjects is indicated by bulges labeled with the subobjects' names.

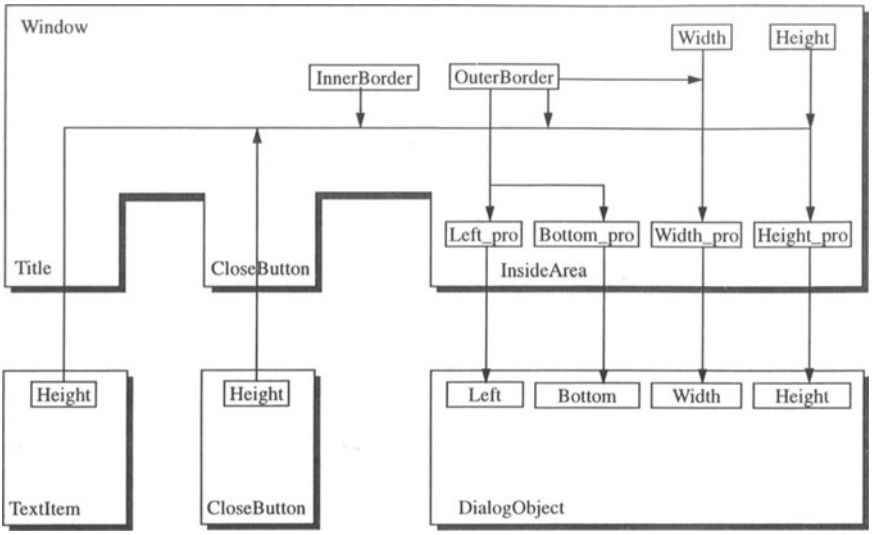

Figure 1 Attribute dependencies for the Window example.

The type of the InsideArea subobject is set to the most general UIO class: DialogObject. Subclasses of Window will replace this type by more specific UIO classes. However, in order to get proper layouts the new type of InsideArea should accept the proposal values unchanged.

4.4 Temporary constraints – associating constraints with dialog states

DIWA event handlers provide an explicit abstraction for dialog states. Therefore, COMPASS introduces temporary constraints that directly rely on these states. The following example specifies an abstract class Mover for 'self moving UIOs'. The event handler MoveHandler specifies the course of the action. For demonstration purpose it is kept simple: the start of the movement is rigidly tied to a left button click, rubber-band feedback is not provided but instead the UIO is moved with every mouse pointer movement. Thus the basic structure of the event handler is revealed: the sequence of the dialog states StartMoving and Moving and the explicit triggering of constraint solving (the statement 'solve') as reaction upon arriving events:

```
Mover class [ DialogObject ]

    attributes: [ OffsetX ... ]

    eventhandlers: [ MoveHandler ]
    MoveHandler: eventhandler
            StartMoving =    ( LeftDownIn (self) → solve → Moving )
            Moving =         ( MouseMove → solve → Moving
                             | LeftUp → solve → StartMoving )
    end MoveHandler

    presClass: MoverPresentation

end Mover
```

The associated constraints are specified in the class MoverPresentation. In state StartMoving the current cursor location is stored in variables CursorStartX and CursorStartY. These values are used to keep track of how much the mouse pointer has been moved. In addition, the position change is not directly applied to Left and Bottom but instead modifies OffsetX and OffsetY. This preserves the parent's influence on the UIO's position:

```
MoverPresentation: presClass [ Presentation ]

    dialogClass: Mover

    status: initial
        OffsetX = 0
        OffsetY = 0

    status: all
        Bottom = Bottom_pro + OffsetY
        Left = Left_pro + OffsetX

    status: StartMoving
        CursorStartX = CursorX
        CursorStartY = CursorY

    status: Moving
        OffsetX = old [OffsetX] + (CursorX - CursorStartX)
        OffsetY = old [OffsetY] + (CursorY - CursorStartY)

end MoverPresentation
```

The predefined attributes CursorX and CursorY provide access to the current location of the mouse pointer expressed in the coordinate system of the UIO the constraint is attached to. Whenever solving is initiated CursorX and CursorY are assumed to have changed their values.

Since in our approach solving is explicitly triggered, the use of an attribute's old value makes sense. For example 'old [OffsetX]' denotes the value of OffsetX before the current solving has been started. Of course, an old value is not allowed as target variable.

The above example uses different kinds of status indications: besides dialog state names the symbols all and initial appear. There is one more status indication: default. In particular, the indications have the following meanings:

- Constraints with indication 'all' apply always.
- Constraints with indication 'initial' apply only at UIO creation time.

- Constraints with indication <name> apply when solving is triggered by an event handler in state <name>.
- Constraints with indication 'default' apply when there is no other constraint for the same target attribute.

The direct association of constraints to event handler state names is a slight short-cut. Since a UIO may have more than one event handler the name of a state might be ambiguous. We accept this ambiguity to keep things simple. However, a <name> status of a constraint does only match for the UIO from which the solving has been invoked. Therefore, the matching is kept local to the UIO.

Explicitly triggered solving as we have seen in the example is the regular case. Implicit solver invocation only happens at UIO creation time to provide an initial layout for the newly created UIO.

4.5 Conditional constraints

In addition to temporary constraints we introduce another means to control the effectiveness of constraints. In some situations it is convenient to deal with variants of layout computation. Consider for instance a Motif-style push button with a text label. Such a UIO can determine its width and height in two different ways. On the one hand it can compute its shape all by itself using the default font height, the label string to be displayed, and some space around the text used for border decoration. On the other hand it can adjust its size according to the values of Width_pro and Height_pro given by its parent. In this case the proposed width is accepted unchanged. The proposed height is taken as an upper bound, so eventually FontHeight is reduced appropriately.

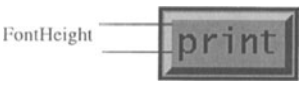

Figure 2 The layout of a labeled button.

Which variant applies is associated with a boolean attribute, the **condition variable** of the constraint. In our example this is the attribute SelfShaping. Its value may again depend on another constraint. For the SelfShaping variant we assume presentation methods textWidth and textHeight to compute the space needed to display the string Value:

```
when SelfShaping
        FontHeight =DefFontHeight
        Width =     textWidth (Value, FontHeight, FontName) + 2 * BorderAndShadow
        Height =    textHeight (Value, FontHeight, FontName) + 2 * BorderAndShadow
end when

when not SelfShaping
        Width =     Width_pro
        FontHeight =min (   DefFontHeight,
                            (Height_pro - 2 * BorderAndShadow) * 2 / 3 )
        TextHeight =FontHeight * 3 / 2
        Height =    TextHeight + 2 * BorderAndShadow
end when
```

The figure below shows attribute dependencies for both variants. It can be seen that the set of involved attributes as well as the directions of the dependencies vary.

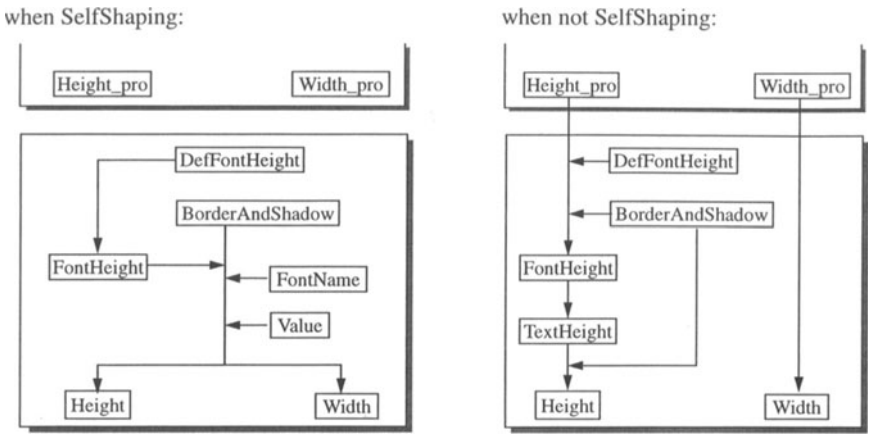

Figure 3 Attribute dependencies according to different values of a condition variable.

4.6 Constraint schemata – the use of attributes and links

So far, we have presented constraints with single target variables. In this section we introduce the more general concept of a **constraint schema**. In its most simple form the same right-hand side is used to determine an attribute for all elements of a set. For example, the following schema sets the left hand sides of UIOs given by the expression 'ItemSet (Mapped)' to a constant:

```
ItemSet (Mapped):Left_pro = Delta
```

To achieve more elaborate schemata we have to make use of additional structures on subobject sets. The most basic additional structure to exploit is the natural ordering of a UIO's subobjects. Basically, this ordering is given by the sequence of the subobjects' creation. In addition it determines the visibility of overlapping siblings: younger subobjects are rendered in front of their older siblings. However, this sequence can be changed at runtime, especially to lift a subobject in front of its siblings.

The natural ordering is a total ordering on all subobjects of a UIO which is automatically induced on arbitrary subsets. Hence, the first and the last element of a subset can be determined as well as the successor and predecessor of a given set member. Thus, to vertically arrange the elements of 'ItemSet (Mapped)' according to their natural ordering we can specify the following schema:

```
with item from ItemSet (Mapped)
      item (first):Top_pro =          Height - Delta
      item (not first):Top_pro =      item (pred):Bottom - Delta
end with
```

The schema distinguishes two cases: the top of the first set member is tied to the UIO's height, the top of the other set members is tied to the bottom of their predecessor. The name item is used as a bound variable to refer to set elements.

The following examples make use of object expressions for both, target and source variables. We extend the Tree example from above denoting parts of a tree layout computation. For this purpose some auxiliary subobject attributes are needed: SubtreeLeft, SubtreeHeight, SubtreeWidth denote position and size of the subtree associated with a given tree node. HasLeftSibling is a SubobBooleanAttribute indicating whether a tree node has a left sibling.

```
with node from NodeSet (HasLeftSibling)
      node:SubtreeLeft =      node(LeftSibling):SubtreeLeft
                              + node(LeftSibling):SubtreeWidth
                              + Delta
end with
```

In addition to the previous example the following constraint schemata make use of aggregations. For example the maximum value an attribute takes for the elements of a certain set can be used as part of the right-hand side of a schema. The second schema counts the number of UIOs denoted by an object expression:

```
with node from NodeSet
      node:SubtreeHeight =    node:Height
                              + max ( node(Children):SubtreeHeight )
                              + Delta
end with

with node from NodeSet
      node:HasLeftSibling =   count (node (LeftSibling)) > 0
end with
```

Note that the last schema determines attribute values that control the target side of another schema.

4.7 Non-local layout specification

Of course, locality of layout computation has its limitations. As a typical example consider the following situation: The UIO class SpecialTree introduces an additional subobject named PropertyWin of type SpecialPopUpWindow. Furthermore, we assume that an instance of class SpecialTree lives inside a SpecialWindow UIO (being its InsideArea subobject). Now consider the task of positioning the property window with a fixed distance to the upper right-hand corner of the SpecialWindow such that moving the window keeps the property window at its side. The following figure sketches this situation, figure 6 below shows a screen dump for similar UIOs.

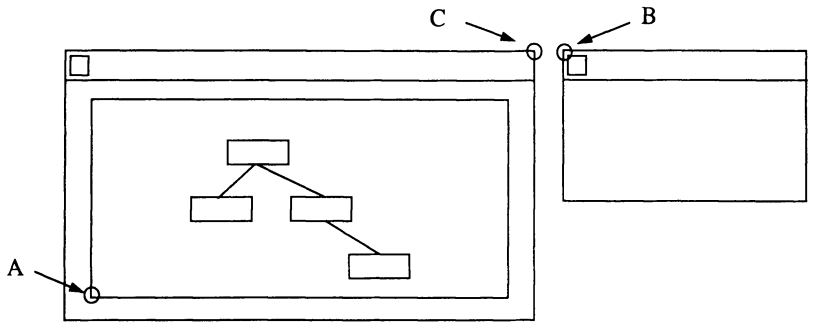

Figure 4 Non-local dependencies.

There are three levels of the UIO hierarchy involved. Point B has to be located in the coordinate system of SpecialTree, i.e. relative to point A, thereby taking into account the position of C. However, point C is outside the scope of SpecialTree. A straight forward solution would associate the following constraints with SpecialWindowPresentation:

```
InsideArea.PropertyWin:Left_pro = Width- InsideArea:Left + Delta
InsideArea.PropertyWin:Top_pro = Height - InsideArea:Bottom
```

However, this solution violates our weak rules on modifying attribute access: SpecialWindowPresentation interferes in the task of SpecialTreePresentation and it depends on the existence of a specific grandchild. Therefore, we introduce additional variables AnchorX and AnchorY. Their values are computed in SpecialWindowPresentation and 'forwarded' to SpecialTreePresentation where they are used to position the property window. In class SpecialWindowPresentation we add two constraints computing the position of B relative to A:

```
InsideArea:AnchorX = Width - InsideArea:Left + Delta
InsideArea:AnchorY = Height - InsideArea:Bottom
```

In class SpecialTreePresentation these values are used to position the property window:

```
PropertyWin:Left_pro = AnchorX
PropertyWin:Top_pro = AnchorY
```

Note that there is a change of coordinate systems. This task is most naturally located in the outer UIO: SpecialWindowPresentation. Once again, the following figure illustrates attribute dependencies:

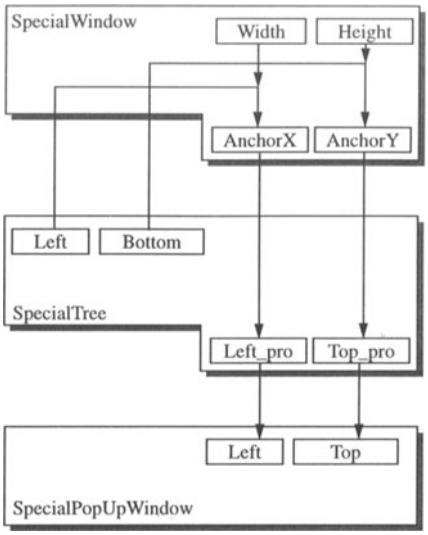

Figure 5 The 'forwarding' of attributes.

5 CONSTRAINT INHERITANCE

Inheritance as we know it from programming languages allows an inherited feature to be redefined by another identically named feature. This principle applies to UIO classes with regard to attributes, event handlers and subobjects. For constraint inheritance we have to take some care because the identifying name of a constraint schema is not quite obvious. Most desirable would be a semantic approach identifying a constraint by its target attributes. However, this is not feasible since inheritance is a compile time issue but with the use of object expressions the target attributes of a constraint schema cannot be determined at compile time. In addition, conditions and dialog states have to be taken into account. Thus, we have to take a syntactic approach: a constraint schema is identified by a combination of its target object expression, the target attribute name, the associated dialog state, and eventually by its associated condition variable name.

The use of subobject sets and constraint schemata support the specification of powerful reusable presentation classes. We have mentioned classes for moving UIOs, windows and trees. Other examples comprise classes for 'UIOs wrapped around their subobjects' or typical form layouts.

COMPASS allows an inherited constraint to be revoked just by stating the constraint with an empty right-hand side. For example, it is a common situation that an inherited constraint for dialog state 'all' has to be revoked in order to replace it by two or more other constraints with different associated states.

6 CONCLUSION

In this paper we have presented the COMPASS constraint language which integrates high-level layout specifications with the DIWA user interface management system. The language provides concepts for temporary and conditional constraints as well as constraint schemata for bundles of similar constraints. These concepts support the declarative specification of changing constraint sets. To our experience the expressive power of COMPASS allows to conveniently specify a wide range of layouts including rather technical situations like trees and application specific graphs and diagrams.

In general, the key to the expressiveness of constraint schemata lies in the expressive power of object expressions allowed for source and target attribute specification. In addition to the concepts presented in this paper we have implemented and tested other features. For example, the grouping of subobjects with identical values for a given attribute turned out to be very useful. Further experiences will certainly reveal more such concepts.

However, understanding complex constraint systems can still be a problem. Especially, the combination of (multiple) inheritance with conditional or temporary constraints makes attribute dependencies difficult to comprehend. To cope with this situation we have built a first prototype of a dependency visualization tool (figure 6). The focus of interest can be restricted to a certain UIO and one or more of its attributes. The tool displays all dependencies for these attributes similar to the illustrations we have used throughout this paper.

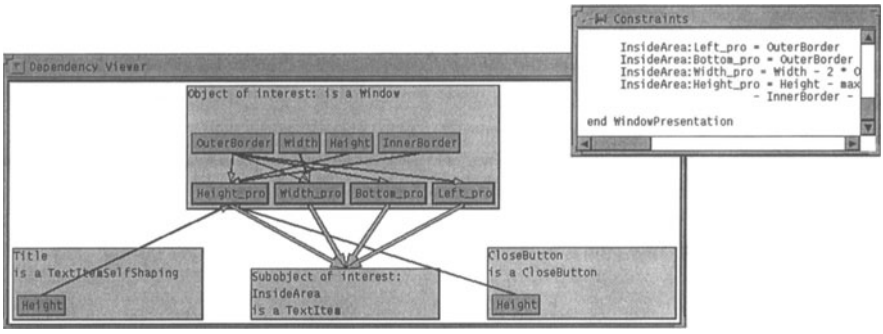

Figure 6 Prototype of a dependency visualization tool.

7 REFERENCES

Freeman-Benson, B. (1990) Kaleidoscope: Mixing Objects, Constraints, and Imperative Programing. Proceedings *OOPSLA/ECOOP '90*, ACM, 77–88.

Hill, R.D. (1993) The Rendezvous Constraint Maintenance System. Proceedings of the *ACM Symposium on User Interface Software and Technology*, ACM, 225–234.

Krasner, G. and Pope, S. (1988) A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, Aug./Sept. 1988, 26–49.

Olson, D.R. (1992) User Interface Management Systems: Models and Algorithms. Morgan Kaufman.

Sanella, M., Maloney, J., Freeman-Benson, B., and Borning, A. (1993): Multi-way versus One-way Constraints in User Interfaces: Experiences with the DeltaBlue Algorithm. *Software – Practice and Experience*, 23(5), 529–566.

Six, H.-W. and Voss, J. (1990) DIWA – A Hierarchical Object-Oriented Model for Dialog Design. Proceedings of the *IFIP Working Conference on Engineering for Human-Computer Interaction*, North-Holland, 383–402.

Six, H.-W. and Voss, J (1992) A Software Engineering Perspective to the Design of a User Interface Framework. Proceedings of the *International Computer Software and Applications Conference*, IEEE Computer Society Press, 128–134.

Vander Zanden, B., Myers, B., Giuse, D., and Szekely, P. (1991) The Importance of Pointer Variables in Constraint Models. Proceedings of the *ACM Symposium on User Interface Software and Technology*, ACM, 155–164.

8 BIOGRAPHY

Hans-Werner Six is a full professor in computer science at FernUniversität Hagen since 1985. He received a Ph. D. from University Karlsruhe in 1978. His current research interests include Geographical Information Systems, Software Engineering, and graphical user interfaces.

Josef Voss is an assistant lecturer at FernUniversität Hagen. He received a Ph.D. in computer science in 1990. His research interests include user interface tools and user interfaces in requirements engineering.

Klaus Korfmacher is an application programmer in chemical industry. He has been a part-time student at FernUniversität Hagen and received a master degree in computer science in 1993.

Discussion

Pedro Szekely: Have you analyzed the computational complexity of your constraint solving scheme?

Josef Voss: We haven't done that yet. Experience shows that the system takes some time to initialize but that once initialization has been completed, the constraints are solved quickly.

Pedro Szekely: Do you have constraints that allow the creation of the widget tree?

Josef Voss: The system does not explicitly support it but there are features in DIWA that should allow the achievement of similar effects.

Jim Larson: I've developed languages that no one uses. How do you know that anyone will use Compass.

Josef Voss: There are currently two users of Compass (the two developers) and they are both happy with the system.

Jim Larson: Why should I use Compass. What capabilities does it have that I could not achieve using normal programming languages.

Josef Voss: The point is not capabilities but comfort of specification.

Jim Larson: How do you measure comfort of specification?

Pedro Szekely: I saw some nice features of Compass that would make some things easier to specify.

Josef Voss: In the original DIWA there were some specifications that were very difficult. In Compass, they would have been much easier.

Michel Beaudouin-Lafon: Have you addressed or do you plan to address the issue of verification of a specification written in Compass.

Josef Voss: Most of the interesting activities of Compass happen at run time. We use one way constraints and test for cycles at run time. Only trivial verification can be done at compile time so I am not optimistic about the possibility of verifying the specification at compile time.