Prototyping Pre-implementation Designs of Virtual Environment Behaviour

James S. Willans and Michael D. Harrison

Human-Computer Interaction Group
Department of Computer Science, University of York
Heslington, York, Y010 5DD, U.K.
{James.Willans,Michael.Harrison}@cs.york.ac.uk

Abstract. Virtual environments lack a standardised interface between the user and application, this makes it possible for the interface to be highly customised for the demands of individual applications. However, this requires a development process where the interface can be carefully designed to meet the requirements of an application. In practice, an *ad-hoc* development process is used which is heavily reliant on a developer's craft skills. A number of formalisms have been developed to address the problem of establishing the behavioural requirements by supporting its design prior to implementation. We have developed the Marigold toolset which provides a transition from one such formalism, Flownets, to a prototype-implementation. In this paper we demonstrate the use of the Marigold toolset for prototyping a small environment.

1 Introduction

One of the characteristics of virtual environments is the lack of a standard interface between the user and system. With virtual environments it is necessary to construct interfaces that support the specific requirements of individual applications [3]. Consider a flight simulator. The components that constitute its interface (including the devices, interaction techniques and objects rendered to the user) are all concerned with simulating the effect of flying the real aircraft. Another application, such as medical training, could not reuse the interface component of the airplane successfully, even though the application may share common goals such as training. This generic lack of standardisation is not surprising considering that virtual environments often seek to imitate the real world. For instance, compare interfaces for driving a car, flying a plane, opening a tin or opening a carton of milk. Each interface matches the requirements of its application and is quite different in terms of the information communicated and physical actions.

The lack of standardisation in virtual environment interfaces contrasts with the dominant style of WIMP (windows, icons, mice and pointers) interaction. WIMP interfaces, such as Microsoft Windows, reuse a consistent interface regardless of application. The devices (mouse and keyboard), interaction techniques (point and click) and components (buttons) are all standardised. This consistent style forms the basis of the success of WIMP applications, because users are aware of how to interact

M. Reed Little and L. Nigay (Eds.): EHCI 2001, LNCS 2254, pp. 91–108, 2001.
© Springer-Verlag Berlin Heidelberg 2001

with new applications because of their knowledge of previous applications. However, the success of virtual environment applications relies on the ability to recreate real world, or novel, interfaces particular to the needs of individual applications. An important side effect of the inconsistent nature of virtual environments is that the work required to build an application interface is vastly increased compared to that of a WIMP. A developer must consider carefully how the requirements of a particular application determine the design of the interface for that application.

There are two parts of a virtual environment application that form the human-computer interface. Firstly, the visual appearance and geometry of the renderings including representations of the user within the environment and, secondly, the behaviour of the interface, including the mapping of the user onto the environment via interaction techniques, and the behaviour of the environment itself in response to user interaction. The visual renderings are usually constructed in a 3D-modeller such as 3Dstudio [2]. These tools model the renderings as they will appear in the environment allowing easy verification of whether they meet the requirements or not (often these are compared to photographs [14]). By contrast, the behaviour of the interface is defined using program code within a virtual environment development application such as [5, 7] in a form that is difficult to analyse in terms of the initial requirements. Consequently, rather than dealing with abstract requirements such as the door must open half way when the user clicks the middle mouse button, the developer must deal with implementation oriented abstractions such as the low-level data generated from the input device(s) and complex mathematical transformations of 3D co-ordinate systems. Additionally, design decisions are embedded within the program code without any higher level documentation. This makes inspection and maintenance difficult. These issues are discussed in more detail in [26].

A strategy which has found some success addressing similar problems within software engineering and human-computer interaction is behavioural design (see [28, 11], for instance). Behavioural design is the process of constructing abstract representations of the behaviour of a system. The aim of this is to describe characteristics of the behaviour in a way that is independent of unnecessary implementation concerns. These descriptions are also used to discuss the design, perform reasoning (formal and informal) and inform the programmer of the precise requirements of the implementation. In addition, such designs facilitate the documentation and subsequent maintenance of the system. The motivation for incorporating this style of behavioural design into the development of virtual environments has recently been recognised and a number of formalisms which attempt at various levels of rigour to support this process have been developed (see [9, 12, 16, 23, 25, 31]).

Despite the clear advantages of pre-implementation behavioural design, it is unrealistic to expect this approach to determine the behaviour of a virtual environment interface completely, particularly in view of the diversity of possible designs. As noted by Myers `the only reliable way to generate quality interfaces is to test prototypes with users and modify the design based on their comments' [18], a view strongly supported in [30]. Thus, a more realistic situation is where behavioural design is closely integrated into a prototyping process such that designs can be tried out and tested with users and the designs refined to reflect their feedback. In order to address this goal we have developed the Marigold toolset [33, 34]. This toolset supports the rapid transition from pre-implementation designs of virtual environment interface behaviour (using the Flownet specification formalism [23, 25]) to fully

working implementation-prototypes. Marigold also provides a means of exploring different configuration of devices and behaviours [34] early in the design cycle. Currently the code-generation module of Marigold is for the Maverik toolkit [1], but the approach is independent of any specific environment. A number of rudimentary model-checking facilities are also provided [35]. In this paper we demonstrate how Marigold supports the prototyping of Flownet specifications by incrementally building a small kitchen environment.

The remainder of the paper is structured as follows. In section 2 we give an overview of the Flownet specification formalism and, in order to strengthen motivation for the use of this representations, we compare a specification to its equivalent program code. In sections 3, 4 and 5 we exemplify Marigold by building specifications and prototypes incrementally for a virtual kitchen. In section 6 we examine related work. Finally, in section 7 we summarise our conclusions.

2 Flownets

A number of formalisms have been used for the description of virtual environment interface behaviour. In [22, 31], process algebra is used to describe interaction techniques, and in [24] the use of state based notations such as Statecharts [10] are investigated. However, more recently there has been a general opinion that virtual environment interface behaviour is better considered as a hybrid of discrete and continuous data-flow components [12, 36]. With this in mind, a number of further formalisms have been investigated [32, 36]. The formalism developed and presented in [13, 17] and Flownets presented in [23, 25] were both developed specifically for the description of virtual environment interaction techniques. This make it possible to abstract from low-level mathematical transformations and data-structures which may confuse a design. One of the major differences between the two notations are that Flownets use Petri-nets [20] to describe the discrete abstractions rather than state-transition diagrams. This enables the description of multi-modal (concurrent) interaction using Flownets. Moreover, although virtual environment behaviour may only consist of interaction techniques in simple environments (walk-throughs), more advanced environments contain world objects with their own behaviour. The user interacts with these world objects using appropriate interaction techniques. In [27] it is demonstrated how Flownets are able to model world objects in addition to interaction techniques. The relation between the user, interaction technique behaviour and world object behaviour is shown in figure 1.

In addition to using Petri-nets to describe the discrete elements, Flownets use constructs from a notation originally intended to model system dynamics [8] for the description of the continuous data flow and transformation processes. We will describe the formalism by way of an example of an interaction technique. The mouse based flying interaction technique enables the user to navigate through a virtual environment using the desktop mouse to control the direction and speed. Variations of this technique are used in many virtual environment packages (see for example VRML [4]). One variation works as follows. The technique is initiated by pressing of the middle mouse button and moving the mouse away from the clicked position. Once the mouse is a threshold distance away from the clicked position, the user's movement through the environment is directly proportional to the angle between the current

pointer position and the point where the middle mouse button was pressed. The distance between these two positions determines the speed. A second press of the middle mouse button deactivates flying.

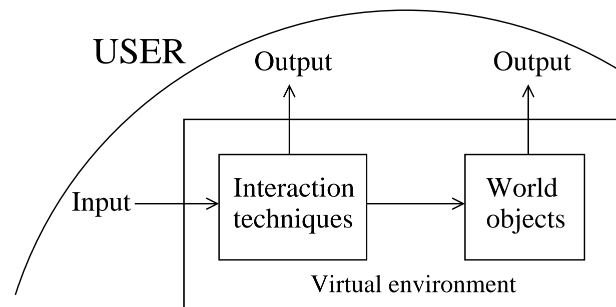

Fig. 1. The relation between the user the behaviour of virtual environments

The Flownet specification of the mouse based flying interaction technique is shown in figure 2. The technique has two plugs to the external environment: one input *mouse*, and one output *position*. When the middle mouse button is pressed the middle *m/butt sensor* is activated and the *start* transition fired (1). The *start* transition enables the flow control, which enables the transformer (*update origin*), which updates the value of *origin* held in a store with the current mouse position (2) (taken from the *mouse* plug). A token is then placed in the *idle* state. When the *out origin* sensor detects that the mouse has moved away from the *origin* position, transition (3) is fired which moves the token from the *idle* state to the *flying* state. A token in the *flying* state enables the corresponding flow control which enables the transformer (*update position*) to update the *position* data in the store using the current *mouse* position and the *origin* position (4). This is then output to the *position* plug. Whenever the *flying* state is enabled, the inhibitor connecting this state to the *start* transition implies that the *start* transition cannot be re-fired. When the *in origin* sensor detects that the *mouse* has moved back to the *origin* position, a transition is fired which returns the token from the *flying* state to the *idle* state closing the flow control and halting the transformation on *position*. Regardless of whether the technique is in the *idle* or the *flying* state, it can be exited by the *middle m/butt* sensor becoming true and firing either one of the two *exit* transitions (5 or 6).

The argument for using this formalism is that there is clarity about the implementation of requirements and about the characteristics of behaviour. This point can be illustrated by comparing the Flownet representation of the mouse based flying interaction technique in figure 2 with the equivalent implementation code in Appendix A written in Maverik. It is not important to understand the code, rather to appreciate how the behaviour of the technique is clearer in the Flownet representation of figure 2. This is because the behavioural structure (what happens when and why) of the technique is explicit in the Flownet but implicit in the code. Additionally, although the code contains more detail than the Flownet representation (low level data state and transformations, for instance), it is difficult to relate the abstractions to the requirements. If we treat the informal description given earlier in this section as the requirements, then it can be seen that the Flownet captures these concepts, for instance *idle*, *flying* and *middle mouse button*. More importantly, the Flownet

captures a precise relation between these concepts. For instance, that it is necessary to be in the state of *flying* in order to *update position,* and that *update position* requires positional data (*origin)* created during the transition to the *idle state*.

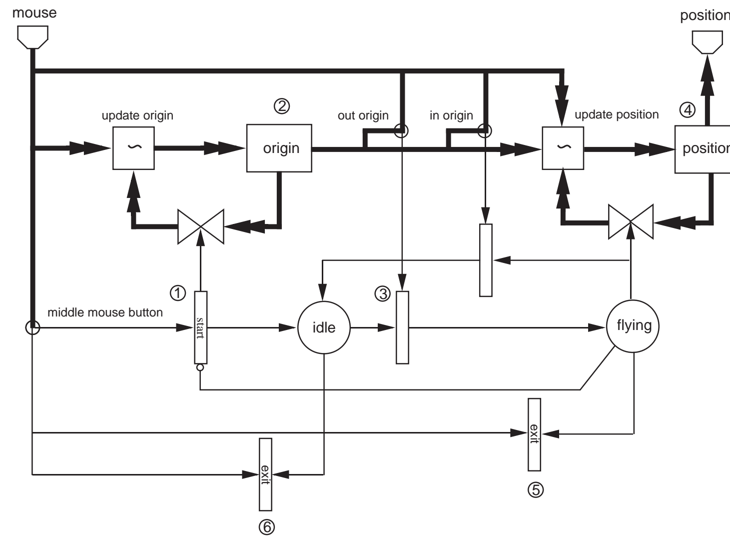

Fig. 2. Flownet specification of the mouse based flying interaction technique

3 Interaction Techniques

The Marigold toolset supports a design process which closely integrates virtual environment behaviour specification using the Flownet formalism, and the prototyping of that behaviour. In the next sections we exemplify this design process by building elements of a virtual kitchen. Within this section we provide a method for the user to navigate around the kitchen. For this we will use the mouse based flying technique introduced in the previous section. Prototyping a Flownet description of an interaction technique using Marigold is a two stage process. The first stage takes place in the hybrid specification builder (HSB) which supports the specification of a Flownet using direct manipulation (figure 3). At this point, a small amount of code is added to some of the nodes of the specification. This code describes the semantics of some of the Flownet components more precisely. There are three types of code that can be added:

Variable code. This is placed in the plugs of the specification. It describes what kind of information flows in and out of the plugs and, hence, around the specification. Illustrated in figure 4 (a) is the code added to the *mouse* plug. An integer variable represents the state of the mouse buttons and a vector represents the mouse position. Variable code is also used to define data which reside in the stores.

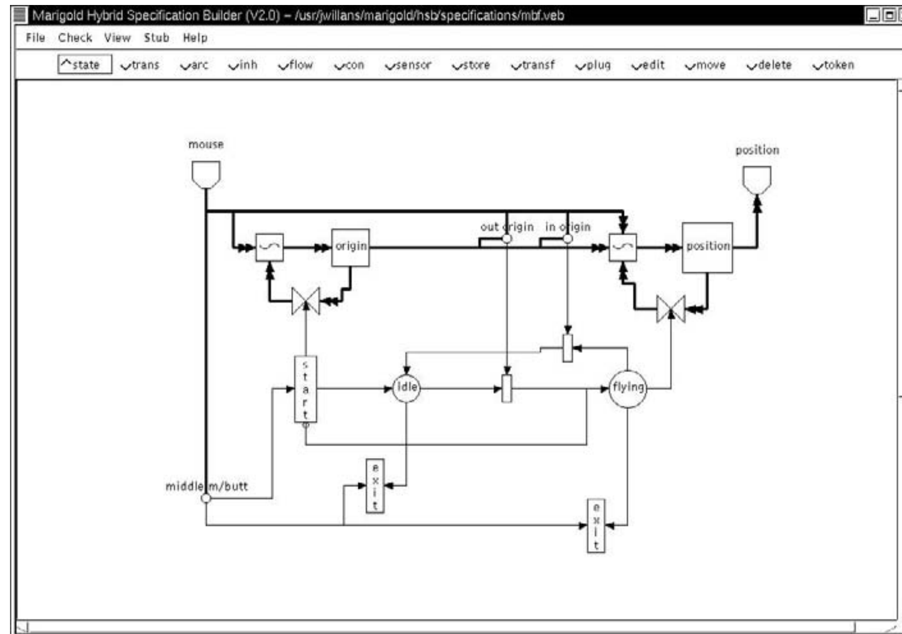

Fig. 3. The mouse based flying specification in Marigold HSB

Conditional code. This is placed in some transitions and all sensors. It describes the threshold state of the data for firing the component. Illustrated in figure 4 (b) is the code added to the *middle m/butt* sensor. As can be seen from figure 4 (b), the HSB informs the developer which data flow in and out of the node (i.e. which data they are able to access). The code specifies that when the middle mouse button is pressed, the sensor should fire.

Process code. This is placed in all transformers and denotes how the information flowing into the transformer is transformed when enabled. Illustrated in figure 4 (c) is the code added to the *position* transformer. This describes how *position* should be transformed using the current *mouse* position and the *origin* position.

Once the code has been added, it is necessary to generate a stub of the interaction technique. This is a description of the interaction technique which is independent of an environment. No commitment is made to the inputs and outputs of a technique.

The second stage of implementing a Flownet specification involves integrating the interaction technique stub, generated from the HSB, into an environment (input devices and output devices, for instance). This is performed using the prototype builder (PB). Illustrated in figure 5 is the mouse based flying interaction technique within the PB and connected into an environment. Each node has a set of variables. The variables for the mouse based flying interaction technique (*mbf*) are those that were placed in the plugs within the HSB. The relation between the environment elements are defined by joining these variables enabling the flow of data from one to another. Within the mouse based flying specification, we have linked a *desktop mouse*, as an input to the technique, and a *viewpoint*, as an output from the technique. From this specification, the code for a prototype environment can be generated and compiled. However, for the navigation to be perceived world objects must be present

within the environment. In the next section, we specify and introduce a world object for the user to observe as they navigate.

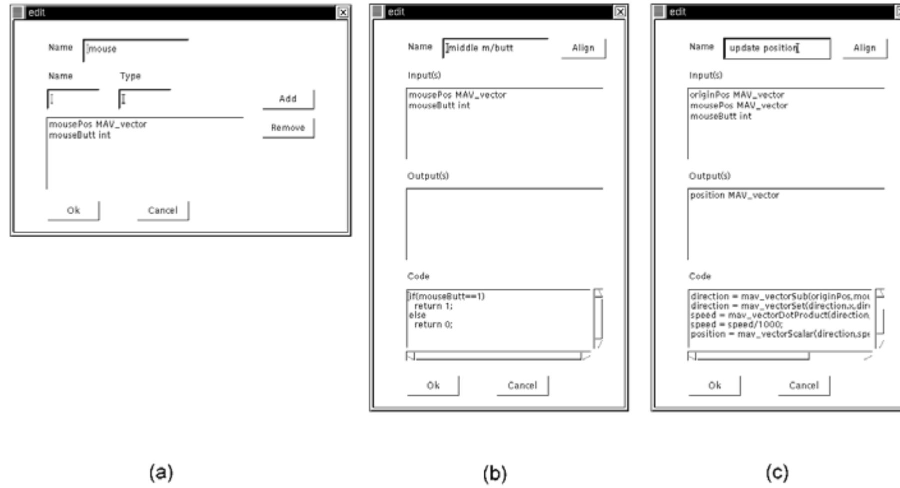

Fig. 4. a) Adding variables to the mouse input plug b) Adding conditional code to the middle mouse button sensor c) Adding process code to the position transformer

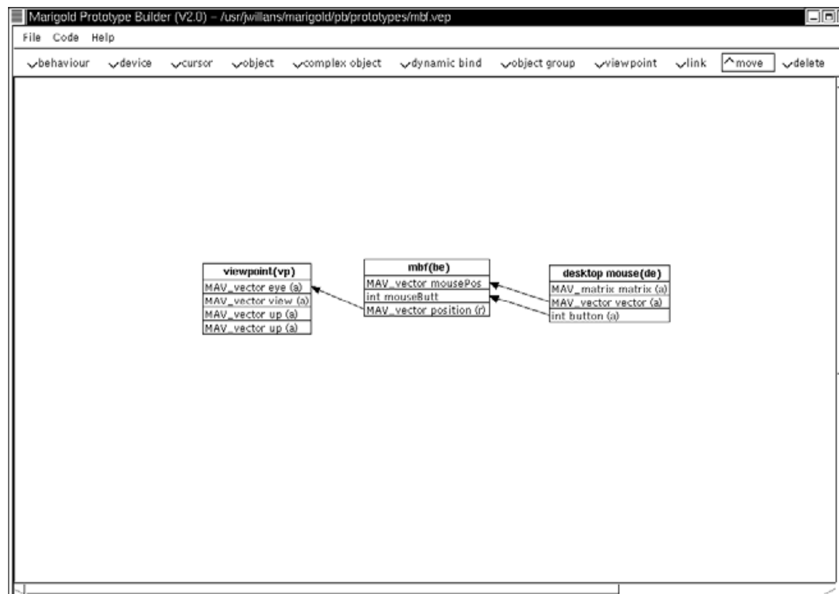

Fig. 5. PB specification for a virtual environment prototype using the mouse based flying interaction technique

4 World Objects

Often virtual environments are simply walk-through where the user navigates the environment observing static renderings of world objects. However, more complex environments reflect the real world where the user can interact with and observe world objects. In this section we expand the environment of the previous section to include a virtual gas hob (oven) world object with which the user can interact.

A world object specification is also constructed using the HSB and a stub generated as described in the previous section. Rather than using the PB to integrate the stub of the world object's behaviour into an environment directly, a third tool, the Complex Object Builder (COB), supports an intermediate refinement stage. Within the COB a link is made between a world object's behaviour stub (generated from the HSB) and the visual renderings constructed using a third party 3D-modelling tool such as 3DStudio [2]. Any additional information required from the external environment (the PB) is also made explicit using the COB. From the COB specification, another stub is generated and integrated into the complete environment (interaction techniques and devices, for instance) within the PB specification. This intermediate refinement stage simplifies the PB specification because a single node is an encapsulation of both a world object's behaviour and appearance. In addition, a world object's encapsulated behaviour/appearance may be reused. The COB supports this type of reuse by packaging these components together in a reusable node.

Figures 6 and 7 illustrate the discrete and continuous parts of the behaviour of the hob respectively. Although the HSB supports the construction of a Flownet specification using one view, it can be useful to split the continuous and discrete part of larger specifications to maintain clarity of presentation. As can be seen from these two figures, common constructs (sensors and flow controls) relate the two views. Code is added to some of the nodes constituting the specification in the manner as described for the mouse based flying interaction technique, and a stub of the behaviour is generated.

The next stage is to integrate this stub with the visual renderings using the COB (figure 8). The node labelled *hob* is the stub of the gas hob generated from the HSB. Representations of the renderings of the *gas switch*, *ignition* and *flame* are related to the behaviour because their states are changed according to the Flownet specification. However the *oven body* is not related because, although this is visually perceived, it has no behaviour. The *external link* node specifies that data is required which is not contained within the specification. Consequently, the position of the virtual hand must be linked into the gas hob within the PB. The relative positioning of the rendering representations are also set within the COB. From the COB, a stub of the specification is generated.

Our original PB specification is expanded to include the complex world object. Figure 9 shows the prototype specification that includes the original mouse based flying interaction technique (*mbf*) and a simple manipulation (*sman*) interaction technique. The *sman* technique, also defined using a Flownet, controls the position of a pointer world object by two mappings of the *keyboard* device. The *hob* world object stub can be seen on the left side of the specification. The one variable within this complex world object is linked to the pointer as required by the *external link* within the COB specification.

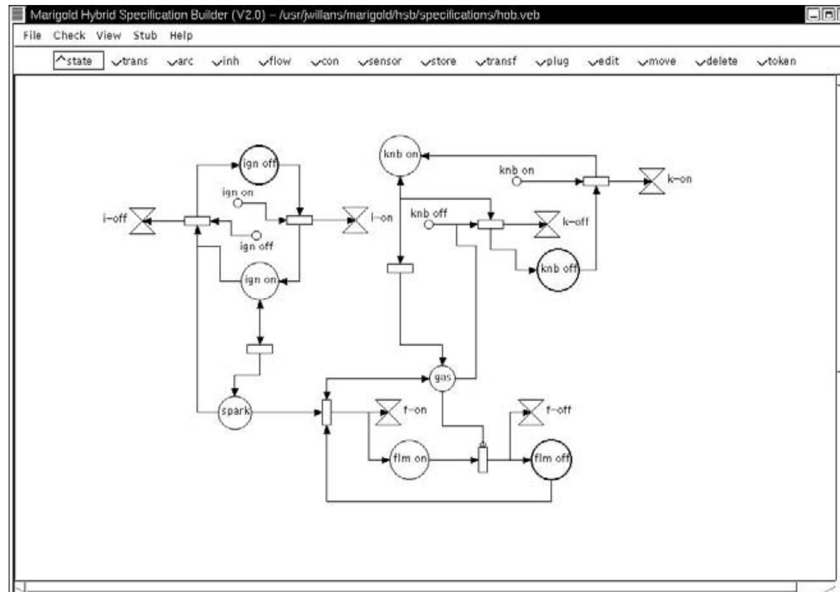

Fig. 6. The discrete part of the Flownet specification for the gas hob within the HSB

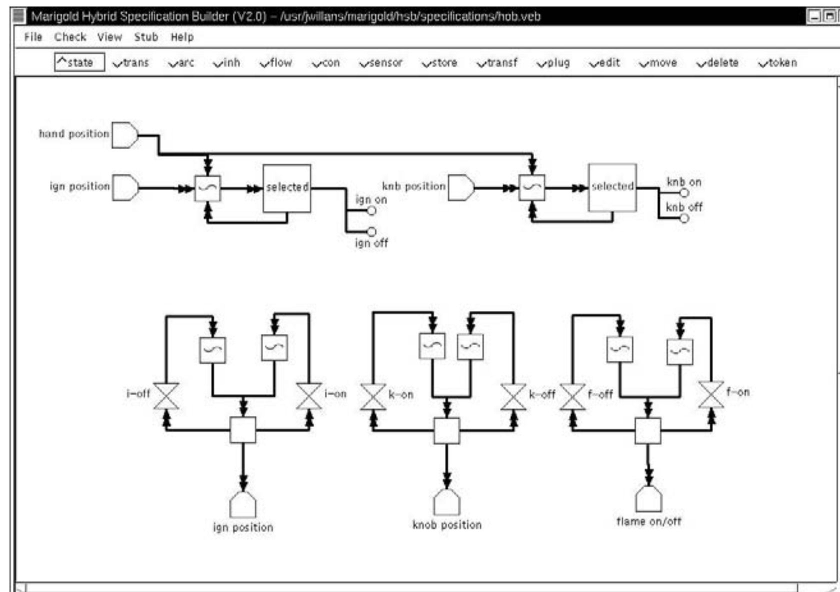

Fig. 7. The continuous part of the Flownet specification for the gas hob within the HSB

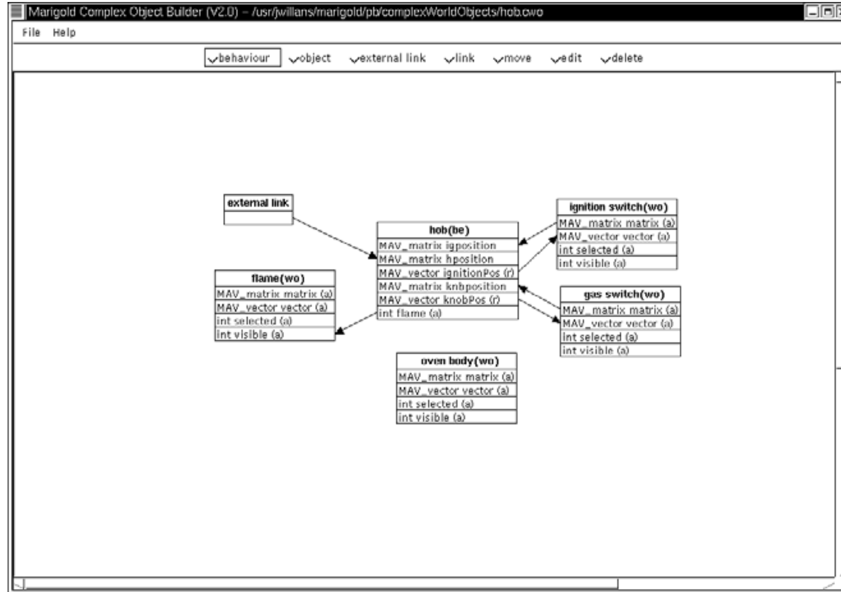

Fig. 8. The COB specification showing the integration of the Flownet stub of the gas hob into the external environment

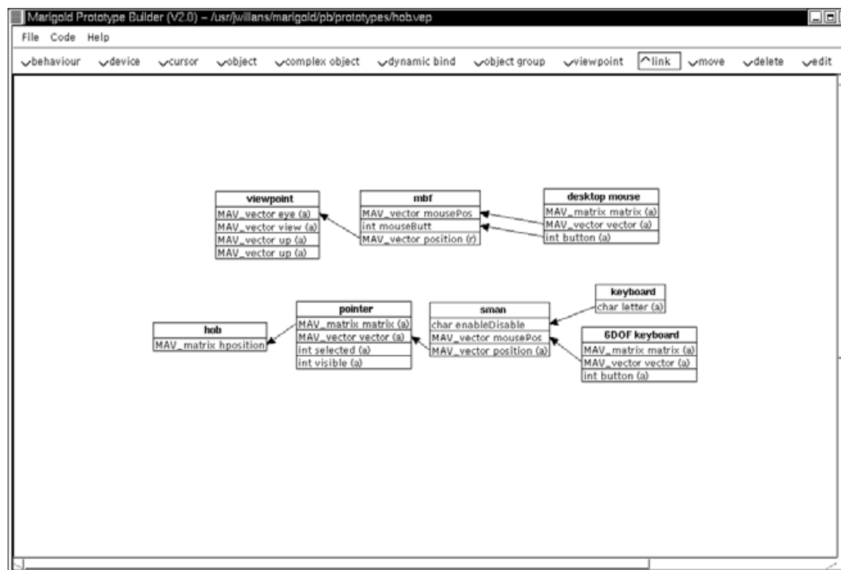

Fig. 9. The PB specification extended to include the integration of a complex world object and manipulation interaction technique

Fig. 10. The prototype (as generated from the PB specification) running

The prototype generated from this specification is illustrated in figure 10. It allows the user to navigate around the environment using the mouse and manipulate the virtual pointer using the keyboard. The virtual pointer can be used to interact with the hob which will behave according to its Flownet specification. The gas on/off switch is on the left side of the oven, and the push ignition on the right. Additionally, a pan is situated on top of the hob (this is part of the oven body rendering representation in figure 8).

5 Non-static Binding

Flownet specifications are not concerned with the environment external to the behaviour, they abstract from this by interfacing to plugs. The PB and the COB tools provide a means of binding an environment to the plugs of Flownets. The binding style described in the previous sections is static. However, a developer may wish to specify that a selection interaction technique can select one of a number of objects within the kitchen, without explicitly linking every object to the selection technique. Similarly, a developer may wish to state that any object placed within a kitchen drawer is affected by the opening and closing behaviour of the drawer because it is in the drawer. This type of non-static binding is supported by two additional constructs within the Marigold PB namely *world object group* and *dynamic bind*. In this section, we expand the specification we have developed in the previous two sections to include behaviours which make use of these constructs. The expanded PB specification is illustrated in figure 11 with a number of new nodes and links.

Firstly, as illustrated at the top of the specification of figure 11, the *world object group* construct provides a method of grouping world object renderings (*ball one* and *ball two*). In this example, the select interaction technique is using this group to determine which object is selected by the pointer object and to change the state of the ball objects selected variable (from false to true). The *sman* (simple manipulation) technique then controls the position of whichever ball is selected.

Secondly, the *dynamic bind* construct allows conditions such as *in the drawer* to be described so that when an object (rendering) satisfies the statement it binds to the behaviour. In figure 11 such a construct (*db*) can be seen labelled *in drawer* and linked to the position of the *drawer* complex object. When a dynamic bind is inserted into a PB specification, the tool asks the user to specify a visual rendering and set the position of this visual rendering. For the *in drawer* dynamic bind a rendering was constructed to represent the space inside the virtual drawer (this is not displayed in the environment). This rendering is positioned so that it is initially inside the drawer. A number of options can also be set on a *dynamic bind* construct. In this example, it is specified that when an object is fully within the space defined by the rendering (in the drawer) it will bind dynamically to the behaviour of the *drawer* and that the *dynamic bind* rendering itself should also bind to the behaviour. Consequently, when objects, such as *ball one* and/or *ball two* are placed within the *drawer*, they bind to the *drawer's* behaviour and open and close with the drawer. In addition, although the bind rendering cannot be observed in the environment, this always remains inside the virtual *drawer*.

The *dynamic bind* construct also has the potential to specify physical laws within a virtual environment. For instance, a bind could be constructed the size of the environment which is linked to a Flownet specification imposing a gravity behaviour. Consequently, when objects are within the environment, they are linked to this behaviour. Or, a bind could be placed within a swimming pool, such that when an object enters the water, it is linked to the gravity of water. To enable this, it would be necessary for Marigold to support the generation of collision detection.

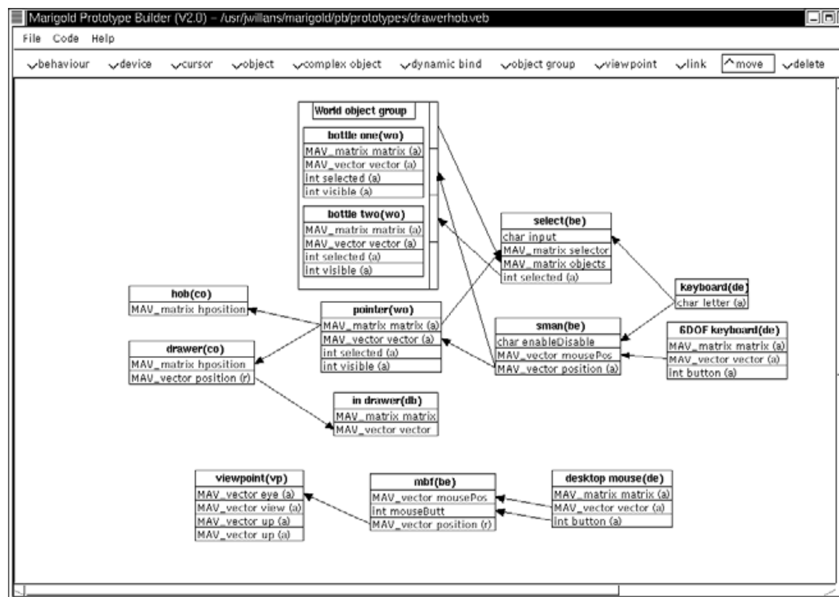

Fig. 11. Expanded PB specification with an additional drawer complex world object, selection interaction technique and dynamic binding constructs

The small kitchen example now supports navigation, control of the virtual hand, behaviour of the gas hob, behaviour of the drawer and the selection and manipulation of the two balls. These balls are initially positioned within the drawer and therefore bind to the drawer's opening and closing behaviour. The virtual hand can be used to remove either one or both of the balls from the drawer, whereupon they lose their binding to the drawer's behaviour. A screenshot of this environment is shown in figure 12.

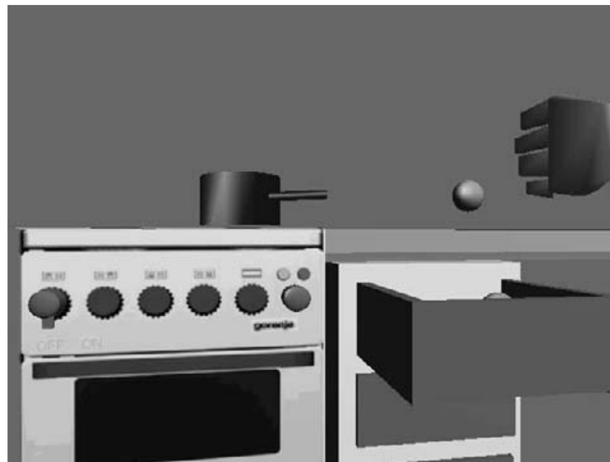

Fig. 12. Final prototype with a drawer, hob and three interaction techniques

6 Related Work

The distinguishing feature of Marigold is the initial consideration of the virtual environment interface behaviour independent of any specific implementation using Flownets. The approach presented in [13, 17] links higher level abstractions to implementation components, such that when these abstractions are rearranged, the change is propagated into the underlying implementation. This is in the style of user interface management systems (UIMS) which has been researched widely for other styles of interface. However, such an approach can be seen to be implementation driven, rather than requirements driven, since an implementation abstraction must exist in order to satisfy a specification abstraction. We have illustrated that using Marigold the developer is building implementation components that meet the requirements of the specification.

Another approach which has influenced the design of the PB and COB components of Marigold is the data-flow style of specification. In [21] modules are connected together (in a manner similar to the PB) to specify virtual environments. These modules are linked to an underlying implementation which changes according to the configuration of the specification. This style of specification has been extended so that it can be achieved while the developer is immersed in the environment [29]. Like the UIMS approach, the semantics of these modules are static and consequently limit

what can be specified . In Marigold, although static components are also used within the PB and COB, these components (devices and rendering representation) are input and output to the behaviour and not the behaviours themselves. The behaviour components are developed specifically for an applications requirements using the HSB.

An approach to designing complete virtual environments is introduced in [15] which contains a component for describing the behaviour of the system using Statecharts. This is an adaptation of an existing real-time system design approach. A number of contrasts can be drawn with Marigold. The behaviour of the systems described are walk-through. It would be difficult using this approach to describe some of the complex user-driven behaviour captured using Flownets, for example. Additionally, Marigold offers a faster and, thus, tighter integration between specification and prototype because of the environment integration method supported by the PB and COB.

The interactive cooperating object formalism (ICO) is a design approach for the development of interactive systems [19]. ICO combines the power of object-oriented structure and Petri-nets for the internal behaviour of objects. A tool is being developed which supports their implementation [6]. The abstractions made within the PB, and the style of PB specification itself, enable a developer to rapidly integrate a specification into an environment. These abstractions are not readily available within ICO and would need to be written using code. In addition, the Flownet formalism itself seems to be a suitably rich formalism for describing virtual environment behaviour compared to Petri-nets alone. Flownets continuous data-flow constructs capture a strong mapping between the user interaction (devices), their state, and the presentation of this state.

7 Conclusion

In this paper we have discussed a need for the pre-implementation design of virtual environment interface behaviour and have reviewed the approaches developed to address this concern. We have also motivated a need to integrate this form of design with the building of prototypes so that developers and users can explore the designs at an implementation level. The Marigold toolset has been developed as an approach to providing a rapid transition between such designs and prototypes. We have described this toolset by exemplifying the incremental design and prototyping of a small kitchen prototype. Such a prototype may be used at two levels. Firstly, it is important to determine whether the environment is usable. For instance, in the example, does the behaviour of the environment enable the user to navigate to the drawer, open the drawer, transfer both balls to the pan on the hob from the drawer, and switch the hob on? At another level, does the environment fulfill the broader requirements? If the small kitchen was to aid in training chefs, then the behaviour of the environment is clearly inadequate. At the end of a session the chef maybe able to complete a task within the virtual environment, but this bears no correspondence to the behaviour of the real environment.

It is anticipated that once the developer and users have arrived at a design that satisfies the requirements, through an iterative process of design and prototyping, one of two options may be taken. Firstly, a programmer may construct the final

implementation directly from the specifications taking into account issue of performance (paramount to larger environments). Secondly, as suggested in [25], the Flownet specifications may be refined to a more detailed specifications (maybe using HyNets as presented in [25]), from which a programmer would implement the behaviour of the environment.

Acknowledgements

We are grateful to Shamus Smith for his comments on this paper.

References

1. *Maverik programmer's guide for version 5.1*. Advanced Interface Group, Department of Computer Science, University of Manchester, 1999.
2. Autodesk-corporation. 3DStudio. 111 McInnis Parkway, San Rafael, California, 94903, USA.
3. Steve Bryson. *Approaches to the Successful Design and Implementation of VR Applications*. London Academic Press, 1995.
4. Rik Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Developers Press, 1997.
5. Superscape Corporation. Superscape, 1999. 3945 Freedom Circle, Suite 1050, Santa Clara, CA 95054, USA.
6. Remi Bastide David Navarre, Philippe Palanque and Ousmane Sy. Structuring interactive systems specifications for executability and prototypeability. In Philippe Palanque and Fabio Paterno, editors, *Design, Specification and Verification of Interactive Systems '00*, pages 97-119. Lecture notes in Computer Science 1946, 2001.
7. Pierre duPont. Building complex virtual worlds without programming. In Remco C. Veltkamp, editor*, Eurographics'95 STAR report*, pages 61-70. Eurographics, 1995.
8. J. W. Forrester. *Industrial Dynamics*. MIT Press, 1961.
9. Mark Green. The design of narrative virtual environments. In *Design, Specification and Verification of Interactive Systems'95*, pages 279-293, 1995.
10. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231-274, 1987.
11. Michael Harrison and Harold Thimbleby, editors*. Formal Methods in Human-Computer Interaction*. Cambridge University Press, 1990.
12. Robert J. K. Jacob. Specifying non-WIMP interfaces. *In CHI'95 Workshop on the Formal Specification of User Interfaces Position Papers*, 1995.
13. Robert J. K. Jacob. A visual language for non-WIMP user interfaces. In *Proceedings IEEE Symposium on Visual Languages*, pages 231-238. IEEE Computer Science Press, 1996.
14. Kulwinder Kaur, Neil Maiden, and Alistair Sutcliffe. Design practice and usability problems with virtual environments. In *Proceedings of Virtual Reality World '96,1996*.
15. G. Jounghyun Kim, Kyo Chul Kang, Hyejung Kim, and Jiyoung Lee. Software engineering of virtual worlds. In *ACM Virtual Reality Systems and Technology Conference (VRST'98)*, pages 131-138, 1998.
16. Mieke Massink, David Duke, and Shamus Smith. Towards hybrid interface specification for virtual environments. In *Design, Specification and Verification of Interactive Systems '99*, pages 30-51. Springer, 1999.
17. S. A. Morrison and R. J. K. Jacob. A specification paradigm for design and implementation of non-WIMP human-computer interaction. In *ACM CHI'98 Human*

Factors in Computing Systems Conference, pages 357{358. Addison-Wesley/ACM Press, 1998.
18. Brad A. Myers. User-interface tools: Introduction and survey. *IEEE Software*, 6(1):15-23, 1989.
19. Philippe A. Palanque, Remi Bastide, Louis Dourte, and Christophe Silbertin-Blane. Design of user-driven interfaces using petri nets and objects. In *Proceedings of CAISE'93 (Conference on advance information system engineering), Lecture Notes in Computer Science*, volume 685, 1993.
20. C. A. Petri. Kommunikation mit automaten. Schriften des iim nr. 2, Institut fur Instrumentelle Mathematic, 1962. English translation: Technical Report RADC-TR-65-377, Griffiths Air Base, New York, Vol. 1, Suppl. 1, 1966.
21. William R. Sherman. Integrating virtual environments into the data flow paradigm. In *4th Eurographics workshop on ViSC*, 1993.
22. Shamus Smith and David Duke. Using CSP to specify interaction in virtual environments. Technical Report YCS 321, University of York - Department of Computer Science, 1999.
23. Shamus Smith and David Duke. Virtual environments as hybrid systems. In *Eurographics UK 17th Annual Conference*, pages 113-128. Eurographics, 1999.
24. Shamus Smith, David Duke, Tim Marsh, Michael Harrison, and Peter Wright. Modelling interaction in virtual environments. In UK-VRSIG'98, 1998.
25. Shamus Smith, David Duke, and Mieke Massink. The hybrid world of virtual environments. *Computer Graphics Forum*, 18(3):C297-C307, 1999.
26. Shamus P. Smith and David J. Duke. Binding virtual environments to toolkit capabilities. *Computer Graphics Forum*, 19(3):C81-C89, 2000.
27. Shamus P. Smith, David J. Duke, and James S. Willans. Designing world objects for usable virtual environments. In *Workshop on design, specification and verification of interactive systems*, 2000.
28. Ian Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.
29. Anthony J. Steed. *Defining Interaction within Immersive Virtual Environments*. PhD thesis, Queen Mary and Westfield College, UK, 1996.
30. Kari Systa. *A Specification Method for Interactive Systems*. PhD thesis, Tampere University of Technology, 1995.
31. Boris van Schooten, Olaf Donk, and Job Zwiers. Modelling interaction in virtual environments using process algerbra. In *12th Workshop on Language technology: Interaction in virtual worlds*, pages 195-212, 1999.
32. Ralf Wieting. Hybrid high-level nets. In J. M. Charnes, D. J. Morrice, and D. T. Brunner, editors, *Proceedings of the 1996 Winter Simulation Conference, pages 848-855. ACM Press*, 1996.
33. James S. Willans and Michael D. Harrison. A toolset supported approach for designing and testing virtual environment interaction techniques. *Accepted for publication in the International Journal of Human-Computer Studies*, 1999.
34. James S. Willans and Michael D. Harrison. A `plug and play' approach to testing virtual environment interaction techniques. In *6th Eurographics Workshop on Virtual Environments*, pages 33-42. SpringerVerlag, 2000.
35. James S. Willans and Michael D. Harrison. Verifying the behaviour of virtual environment world objects. In *Workshop on design, specification and verification of interactive systems*, pages 65-77. Lecture notes in computer science 1946, 2000.
36. Charles Albert Wuthrich. An analysis and a model of 3D interaction methods and devices for virtual reality. In *Design, Specification and Verification of Interactive Systems '99*, pages 18-29. Springer, 1999.

Discussion

J. Höhle: Can you model independent behaviour instead of reactive behaviour - such as a time based behaviour?
J.Willans: You certainly can add timing constraints in the system but it would be hard to ensure their satisfaction. Satisfaction would depend upon implementation details.

N. Graham: How hybrid is your specification? You seem to be modelling dynamic behaviour within discrete framework. It does not seem to involve truly continuous inputs, such as dealing with the effect of momentum when lifting an object, or dealing with sound. Is this natural in flownets?
J.Willans: We do not have that kind of example yet. Flownets is hybrid in representation at the design level. There are other lower level representations that capture continutiy in more detail (e.g. HyNet).

Appendix A

```
#include "maverik.h"
#include "mav_tdm.h"

MAV_vector new_offset;
MAV_vector origin_pos;
int mouseClick;

/* RELATES TO FLOWNET SPECIFICATION    */
/* MIDDLE MOUSE BUTTON + UPDATE ORIGIN */

int mouseButtonPress(MAV_object * o,
MAV_TDMEvent * ev)
{
   int origin_x, origin_y;
   int xx, yy;
   if (ev->tracker == 0 && eb->button ==1) {
      mav_mouseGet(mav_win_all, &origin_x,
               &origin_y, &xx, &yy);
      origin_pos.x = origin_x;
      origin_pos.y = origin_y;
      origin_pos.z = 0;
      mouseClick = !mouseClick;
   }
}

/* RELATES TO FLOWNET SPECIFICATION */
/* UPDATE POSITION                  */
void updateViewpoint(void)
{
   mav_win_current->vp->eye =
     mav_vectorAdd(mav_win_current->vp->eye,
               new_offset);
}

/* RELATES TO PETRI-NET PART OF */
/* FLOWNET SPECIFICATION        */

void interaction(void)
{
   MAV_vector current_mouse;
   MAV_vector direction;
   float speed;
   int curr_mouse_x, curr_mouse_y, xx, yy;
   if (mouseClick) {
      mav_mouseGet(mav_win_all, &curr_mouse_x,
               &curr_mouse_y, &xx, &yy);
      current_mouse.x = curr_mouse_x;
      current_mouse.y = curr_mouse_y;
      current_mouse.z = 0;
      if (outOriginSq(current_mouse)) {
         direction = mav_vectorSub
           (origin_pos,
            current_mouse);
         direction = mav_vectorSet
           (direction.x,
            direction.z,
            direction.y);
         speed = mav_vectorDotProduct
           (direction, direction);
         speed = speed / 1000;
         new_offset = mav_vectorScalar
           (direction, speed);
         origin_pos = mav_vectorAdd
           (new_offset, origin_pos);
         updateViewpoint();
      }
   }
}
```

```
/* RELATES TO FLOWNET SPECIFICATION */
/* OUT OF ORIGIN                    */

int outOriginSq(MAV_vector current_mouse)
{
   MAV_vector temp;
   float distance;
   temp = mav_vectorSub(current_mouse,
origin_pos);
   distance = mav_vectorDotProduct(temp, temp);
   if (distance > 5) {
     return 1;
   }
   return 0;
}

int main(int argc, char *argv[])
{
   MAV_SMS *sms;
   MAV_composite comp;

   mav_initialise();
   mav_TDMModuleInit();
   sms = mav_SMSNew
      (mav_SMSClass_objList, mav_objListNew());
   mav_compositeReadAC3D
      ("desk.ac", &comp,MAV_ID_MATRIX);
   comp.matrix = mav_matrixSet(0, 0, 4, 0, 0, 0);
   mav_SMSObjectAdd
      (sms,  mav_objectNew(mav_class_composite,
&comp));
   mav_callbackTDMSet
      (mav_win_all,          mav_class_world,
mouseButtonPress);
   mav_frameFn0Add(interaction);
   mouseClick = 0;
   while (1) {
      mav_eventsCheck();
      mav_frameBegin();
      mav_SMSDisplay(mav_win_all, sms);
      mav_frameEnd(); }
```