Integrating behavioural design into the virtual environment development process

James Stephen Willans

Submitted for the degree of Doctor of Philosophy

The University of York
Department of Computer Science
Human-Computer Interaction Group

November 2001

Abstract

A number of specifications formalisms have been developed (or applied) to support the abstract design of the behavioural component of the virtual environment interface. These formalisms subscribe to the philosophy that virtual environments should be viewed as hybrid systems which combine discrete and continuous behaviour. A significant deficiency in designing behaviour in this way is that the designs cannot be directly executed and explored in the same manner as an implementation. This limitation makes it difficult for a designer to evaluate the suitability of designs. The thesis presents the Marigold toolset which supports two approaches to evaluating behaviour described using the Flownet hybrid formalism.

The first approach involves refining the design to an implementation prototype where it can be explored with users. An emphasis within this approach is a usable means of integrating the behaviour with the components that form the direct interface to the users (the presentation) such as devices. This is achieved by the use of visual data flow networks. The second approach involves the analysis of Flownets so that characteristics of the design can be automatically checked. A consideration within this approach is a usable means of specifying the properties and understanding the results of the analysis.

A secondary focus of the thesis is a requirements specification approach for virtual environments. This is motivated by reports that one of the problems with the virtual environment development process is an accurate interpretation of the users requirements by the designers. The approach elicits requirements in a language familiar to the users, and translates these into a specification that can be used by a designer to construct designs. The Primrose tool has been developed to support this approach.

Contents

List of Figures

Acknowledgements

I am indebted to my supervisor Professor Michael Harrison whose support, advice and friendship has made my research so enjoyable. Although the work presented in the thesis is my own, a number of further individuals have enhanced my thoughts. Dr. Shamus Smith provided a sounding board for ideas that matured as a result of his feedback. The formal methods and graphics expertise of Dr. David Duke provided useful insights into the strengths and limitations of my ideas. Useful feedback has also been provided by Dr. Mieke Massink, Dr. José Campos, Dr. Darren Priddin, Professor Colin Runciman, Karsten Loer and I am particularly grateful for correspondences with Professor Robert Jacob (Tufts University). Jon Cook (University of Manchester) provided excellent support for the Maverik toolkit, and James Carter provided local support beyond the call of duty for my machines. Ben Challis and Shamus Smith were, and remain, excellent friends. Maria looked after me when I was not working, and continues to make an important difference. This thesis is dedicated to my parents and it is for their support, above all, which I am most grateful.

Declaration

Much of the work presented in this thesis has already been published elsewhere co-authored with Michael Harrison [Willans and Harrison 1999; Willans and Harrison 2000a; Willans and Harrison 2000b; Willans and Harrison 2001a; Willans and Harrison 2001b], Michael Harrison and Shamus Smith [Willans, Harrison, and Smith 2000; Willans, Smith, and Harrison 2001a; Willans, Smith, and Harrison 2001b] and David Duke and Shamus Smith [Smith, Duke, and Willans 2000]. Parts of chapter 6 are based on collaborative work with Shamus Smith and David Duke. In all other cases, I have presented only those aspects of the work which are directly attributable to me.

An article about the work presented in this thesis has appeared in Technology Research News [Patch 2001].

To my parents, Stephen and Mary

Chapter 1

Introduction

This thesis is concerned with the design of 3D virtual environment interfaces (sometimes called virtual reality interfaces). In recent years the use of 3D virtual environments has become more widespread, partly as a consequence of diminishing technology costs and partly due to the availability of development applications such as the Maverik toolkit [Hubbold, Dongbo, and Gibson 1996]. This class of interactive system is beginning to realise its potential in applications such as training [Higgett and Bhullar 1998; Hodges, Watson, Rothbaum, and Opdyke 1996], product prototyping [Thompson, Maxfield, and Dew 1999] and data visualisation [Sastry, Boyd, Fowler, and Sastry 1998] outside the context of specialised laboratories.

1.1 Virtual environments

The dominant form of computer interface continues to be the windows, icons, mice and pointer (WIMP) interface. A defining characteristic of this interface is that, regardless of application, the user interacts with consistent concepts such as menus and buttons via consistent interaction techniques. This enables the user to use previous knowledge of interaction to successfully interact with new applications. The consistent nature of WIMP interfaces also has a favourable impact on their development. A developer only needs to consider those aspects of the interface which are not reused since the reusable concepts are known to be adequate (and are provided in standard libraries). For instance, an interaction technique such as *drag and drop* does not need to be redesigned for each new application, its usability and functionality (and other concerns) are well established.

Virtual environment interfaces are commonly developed to simulate real world interfaces, or interfaces to support highly specialised tasks where there are novel concepts. Although there is a level of consistency in these types of applications, it is much finer grain than that of WIMP interfaces. As a result, the development

of virtual environment interfaces is a non-trivial process. A developer must design concepts in view of the requirements of each application and ensure that these designs are usable. Our concern is with the design of the software part of virtual environment interfaces. In the next section we examine the general approach to their design.

1.2 Virtual environment design

Two major components of a virtual environment interface are the visual world objects[1] that are rendered to a user and the behavioural rules that determine how the environment responds to user interaction.

The world objects of a virtual environment are usually designed using 3D modellers such as 3DStudio [Autodesk-corporation 1997] and AC3D [Colebourne 2001]. A screenshot of AC3D is shown in figure 1.1 displaying an office desk and chair world objects. Using these tools, world objects are designed by dragging and dropping visual primitives from a menu bar onto one of three views of the object being designed (front, side and plan). The visual primitives are perceived as in the real world including details of colour, texture and their spatial positioning. This makes it easy for a designer to make a transition between the requirements of the world objects (often described using photographs or drawings) and their realisation within a design. In addition, 3D modellers provide the facility to interact with the world objects (figure 1.1 lower right) by rotation and zoom, allowing the designer to evaluate the suitability of the designs of world objects by exploring how it will appear in the finished environment. Current research is shortcutting the transition between the requirements of world objects and their designs further. The approaches presented in [Zeleznik, Herndon, and Hughes 1996; Deering 1996] provide a means of translating rough sketches into concrete designs of world objects. The approach described in [Gibson and Howard 2000] demonstrates how photographs can be translated into concrete design of world objects with minimal human intervention.

In contrast, the design of the behaviour of a virtual environment is integrated into its implementation. The abstractions used can take one of two forms or something in-between. In the case of high-level implementation toolkits such as Alice [Pausch 1995], the behaviours are described using a language that has a (loose) correspondence to concepts in the requirements (the real world), for example to make a rabbit world object look at a helicopter world object: *bunny.pointat(helicopter)*. However there are a limited number of predefined high level abstractions (such as *pointat* that can be used. Alternatively, in the case of a low-level implementation toolkit such as Maverik [Hubbold, Dongbo, and Gibson 1996], the behaviours are described using geometric

[1] Hereafter referred to as world objects.

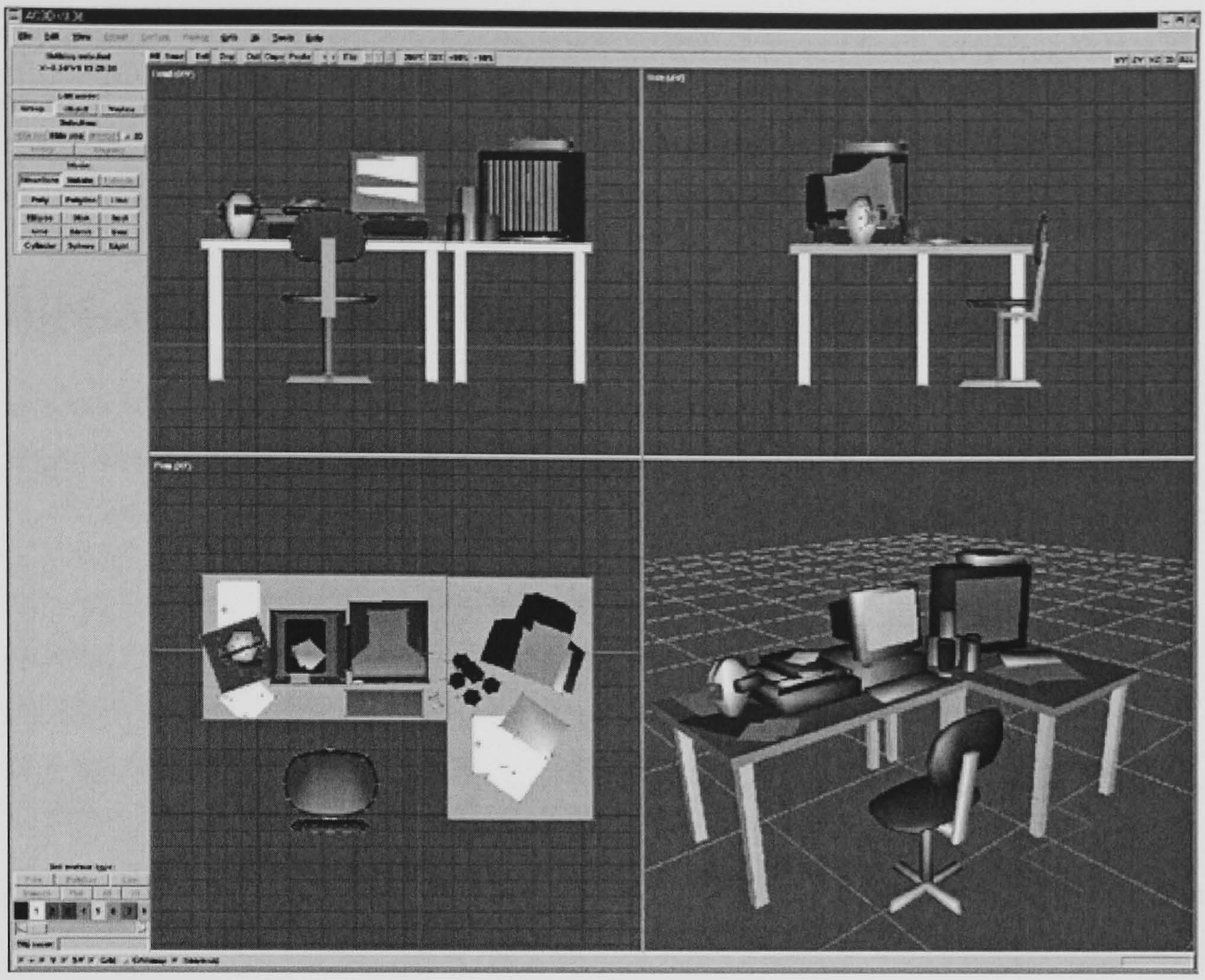

Figure 1.1: AC3D world object modeller

translations that bear little resemblance to how the requirements are expressed. Using languages akin to Alice, abstractions can be used which relate to the requirements, but their high-level nature limits what can be described. Languages like Maverik afford greater flexibility, but the behaviour must be designed using low-level abstractions which are difficult to relate to the requirements. This tradeoff between abstractions and flexibility is visualised in figure 1.2. Implementation abstractions make it difficult to achieve a description of the behaviour in a flexible manner using abstractions that correspond to the requirements.

This problem can be resolved by separating the design from the implementation. Here abstractions which are incomplete (non-executable) are used. Thus providing more flexibility and a better link with the requirements. A number of design specification formalisms have been developed (or applied) to support this (for example, [Jacob 1996; Smith, Duke, and Massink 1999]). Such approaches build upon similar techniques developed for more traditional interactive systems where, for instance, state-transition diagrams [Wasserman 1985], Petri-nets [van Bilion 1988] and Statecharts [van Zijl and Mitton 1991; Horrocks 1999] are used.

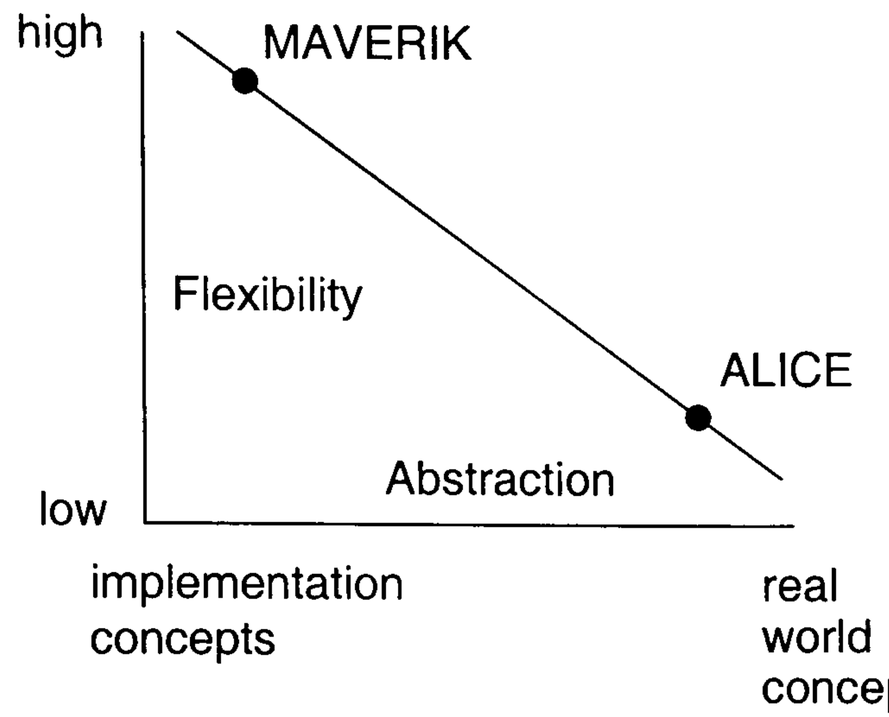

Figure 1.2: Implementation flexibility/abstraction tradeoff

Despite the strengths of using such specification formalisms, a significant weakness is they cannot be executed in the same manner as an implementation. This makes it difficult to evaluate whether a design is correct [Carr 1996]. It is generally considered that this deficiency is one of the main reasons behavioural specification formalisms have not been more widely adopted [Carr 1996; Morrey, Siddiqi, Hibberd, and Buckberry 1998]. The primary concern of this thesis is enabling the evaluation of design specifications of virtual environment behaviour.

1.3 Evaluating designs

Newman and Lamming separate usability evaluation into two approaches [Newman and Lamming 1995, p167]:

- Empirically - by building prototypes of the design.
- Analytically - by analysing design specifications.

This distinction is equally applicable and useful to software evaluation *per-se* [Berry and Wing 1985].

Prototyping designs

The prototyping of user interface designs is motivated by a need to involve the user within the design process. Often users have difficulty articulating their precise requirements for a system, but by interacting with a prototype the user can identify strengths and weaknesses of a design. Prototyping is a critical part of the engineering

of user interfaces [Sommerville 1996, p153], 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' [Myers 1989].

Analysing designs

Analysis is concerned with asking questions directly about a design. User interface designs are commonly analysed using informal techniques [Newman and Lamming 1995, p167], however a weakness of this type of analysis is its imprecision [Campos 2000, p25]. The use of formal methods has been explored [Abowd 1991; Harrison and Thimbleby 1990] to address this because their mathematical nature enables greater certainty that the analysis is correct (though the wrong thing could be specified). More recently the use of automatically analysing user interface designs has been explored [Paternó 1995; Campos 2000].

1.4 Thesis overview

The main contributions of this thesis are approaches to evaluating designs of virtual environment behaviour using prototyping and analysis. In order to support these approaches, the Marigold toolset has been constructed. Marigold supports the design of virtual environment behavioural using the Flownet formalism [Smith and Duke 1999b; Smith, Duke, and Massink 1999], and the refinement of designs to a prototype by 'plugging' the designs into a presentation (interaction devices and world objects). Analysis evaluation is enabled by Marigold's support for automatically checking properties of the Flownet designs.

A fundamental step prior to the design of any system is understanding the requirements that the design must satisfy. Without an adequate means of eliciting the requirements from the intended end-user and specifying these, the process of design becomes difficult and error-prone. A further contribution of this thesis is an approach to specifying virtual environment requirements in a manner that considers both the end-user and the developer. The Primrose tool has been developed to support this approach.

The thesis is structured as follows:

- Chapter 2 Background examines the current state of affairs with respect to two criteria. Firstly, the alternative specification formalisms for designing virtual environment behaviour. Secondly, approaches to evaluating such behavioural specifications.

- Chapter 3 Flownets details the existing Flownet formalism which we will utilise as a behavioural design specification formalism within this thesis.

- Chapter 4 Prototyping Flownets introduces the Marigold toolset and describes how it supports a transition from Flownet designs to implementation prototypes.

- Chapter 5 Analysing Flownets describes support within the Marigold toolset for the automatic analysis of Flownet designs.

- Chapter 6 Requirements Specification introduces an approach to eliciting and specifying virtual environment requirements. The Primrose tool is described which supports the application of this approach.

- Chapter 7 Case studies describes two case studies which apply the Marigold toolset to the design of virtual environments.

- Chapter 8 Conclusion reviews the contributions of this thesis and presents direction for future work.

Chapter 2

Background

The purpose of this chapter is twofold. Firstly, in section 2.1 we review design formalisms used to describe virtual environment behaviour to justify our use of Flownets. Secondly, in sections 2.2 and 2.3 we examine the extent to which current methods for prototyping and analysing such descriptions supports the evaluation of designs. This provides a context for the Marigold toolset.

2.1 Behavioural design formalisms

A number of formalisms have been explored for the specification of virtual environment behaviour at various levels of rigour. In [van Schooten, Donk, and Zwiers 1999; Smith and Duke 1999a] CSP (communicating sequential processes) [Hoare 1978] is used. The approach presented in [Kim, Kang, Kim, and Lee 1998] uses Statecharts [Harel 1987] to describe non-user driven behaviour (the user observes passively). These styles of specification abstract the behaviour into discrete, token style, steps. The user generates a token and the computer responds with a token determined by the state of the behaviour. For traditional interfaces such as those driven by menus and those based on WIMPs, these techniques work well because they are rich enough to reflect their command based nature [Jacob 1995].

When virtual environment behaviour is described using these techniques, it has been found that the descriptions lack a level of richness adequate to characterise the behaviour [Jacob 1995; Smith and Duke 1999b]. This is because the user's interaction with the environment is often continuous and the user perceives the rendering of the environment continually. This continuous behaviour should also be considered. Consequently, virtual environments may be considered more conveniently as hybrid systems and their behaviour modelled as a combination of discrete and continuous components [Jacob 1996; Smith, Duke, and Massink 1999; Wüthrich 1999]. Dix and Abowd also argue the need for this distinction in the wider context of interactive

systems, although they refer to this as status (continuous) and event (discrete) [Dix and Abowd 1996]. Three visual formalisms have been developed for (or applied to) the hybrid specification of virtual environment behaviour: HyNet, the Tufts formalism and Flownets.

For convenience we consider virtual environment behaviour as being of two types. Firstly, interaction techniques that map the user onto the environment to support navigation and the selection and manipulation of world objects. Secondly, world object behaviour defining how world objects respond to user interaction. To compare the three formalisms mentioned above, we shall use the interaction technique called mouse-based flying. Mouse based flying enables navigation on the x and z axis using the desktop mouse. The technique is initiated by pressing the middle mouse button. When the mouse cursor is moved away from the position of the mouse click, navigation through the environment begins. The user's speed and direction is directly proportional to the angle and distance between the current pointer position at the point the middle mouse button was pressed. Flying is deactivated by a second press of the middle mouse button. Variations of this technique are used in many desktop virtual environment packages such as the Virtual Production Planner [BBC/Colt International 1997] and VRML (virtual reality modelling language) [Carey and Bell 1997]

2.1.1 HyNet

HyNet (Hybrid High-Level Petri-Nets) [Wieting 1996] builds on Petri-nets [Petri 1962] using object-oriented concepts including inheritance and polymorphism. The application of HyNets to virtual environment interaction techniques is demonstrated in [Massink, Duke, and Smith 1999; Smith, Duke, and Massink 1999]. A HyNet specification is made up of a number of states and transitions which are related by arcs. Tokens are moved dynamically from state to state by the transitions.

The transitions can either be discrete or continuous, and are inscribed with a five part label which describes their firing capacity, activation condition, firing action, delay time and firing time:

- The firing capacity defines how often a transition can fire in parallel with itself.
- The activation condition is a boolean pre-condition for the transition firing to take place.
- The firing action for discrete transitions consists of executable expressions, for the continuous transitions it consists of a differential equation.

- The delay time (discrete transitions only) defines the time that must pass between firing and re-enabling the transition.
- The firing time (discrete transitions only) defines the length of time the execution of the transition must take.

There are a number of different types of arcs linking places and transitions. Inhibitor arcs prevent a transition firing (visually shown as an arc containing an open circle in its centre). Conversely, enabling arcs enable a transition to fire (shown as an arc ending in an open circle). The inhibitor or enabling arcs are activated if there is a token present in the place they are connected to. Finally, there are standard Petri-net arcs which are associated with a token type (weighting) defining which tokens can pass.

Two types of tokens flow around a HyNet specification. Either simple Petri-net tokens which mark the state of behaviour, or complex tokens which also mark the state but are instances of classes.

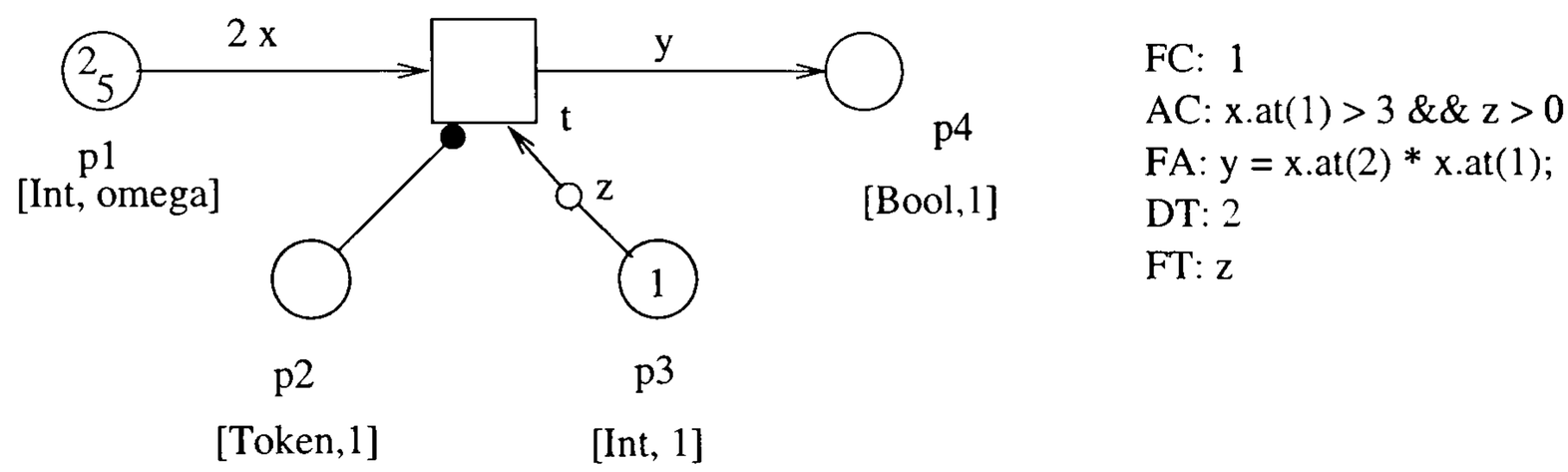

Figure 2.1: HyNet discrete transition (taken from [Massink, Duke, and Smith 1999])

In figure 2.1 a discrete transition is illustrated which is related to four places. Place p1 contains two tokens (2,5) and is related to the transition via a Petri-net regular arc. This transition specifies that two tokens can be carried and that these are assigned to x. Place p2 contains no tokens and is related to the transition via an inhibitor arc. Place p3 contains a single token (1) and is related to the transition via an enabling arc. In order for a transition to fire (regardless of whether it is discrete or continuous) it is necessary for every state targeting the transition via a regular arc and enabling arc to contain enough tokens to match the weighting of the arc, and for every state targeting the transition via an inhibitor to contain no tokens. This axiom is satisfied in figure 2.1. In addition the activation precondition (AC) for the transition must be satisfied. In this case, this means that the value of the token carried on the enabling arc (z) must be greater than 0, and that the value assigned to the first x (x.at(1)) must be greater than 3. This precondition can be satisfied by the following execution x.at(1) = 5, x.at(2) = 2 and z = 1.

When the marking of the net and the activation precondition has been satisfied, then the delay time (DT) of 2 clock ticks begins, when this expires the execution of the firing action occurs (FA). In this case the firing action specifies that the value of x.at(1) should be multiplied by x.at(2) and the result (10) placed in p4. The (FT) condition dictates that this should be completed in the number of time steps specified by z (1). This transition can only fire once in parallel with itself (FC).

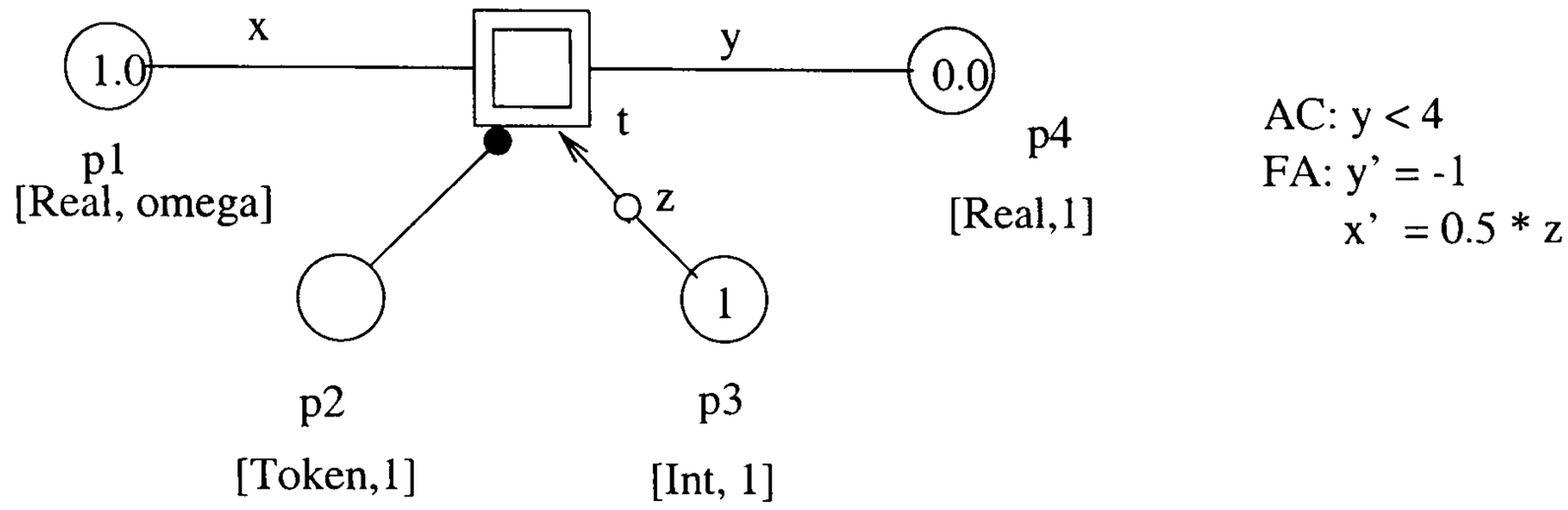

Figure 2.2: HyNet continuous transition (taken from [Massink, Duke, and Smith 1999])

To illustrate a continuous transition consider the example in figure 2.2. The job of a continuous transition is to continually change the value of objects in the adjacent places. It is graphically described by a double box. The transition enabling condition is the same as the discrete. In this example the precondition (AC) specifies that the value of y should be less than 4. This precondition is continually checked before each iteration of the transition execution, if the condition fails then execution is halted. In this example, the value of y is decremented by 1, and the value of x is increased by 0.5 * z every clock cycle. This behaviour will continue until the precondition fails or if a bounded (required) token is consumed by other behaviour in the net.

Shown in figure 2.3 is part of the mouse-based flying interaction technique specified using the HyNet formalism. In this specification, the action conditions (AC) are shown in the upper part of transitions and the firing actions (FA) are shown in the lower part of transitions. The top part of the HyNet specification describes the behaviour of the mouse. Initially there is a complex token in the *Mouse* state (denotated by [Mouse,1]) which records the position of the mouse and the state of its buttons in the {(x,y,vx,vy),0,0,0} data structure. The position part of this token is continually updated by the *move mouse* transition. When a mouse button press occurs, the *whenever you like* transition records the state of the button in this token and returns it to the *Mouse* state. The lower part of figure 2.3 describes how the state of the complex token in the *Mouse* state changes the state of the technique. A transition continually updates the complex token in the *Cursor* state which records the position

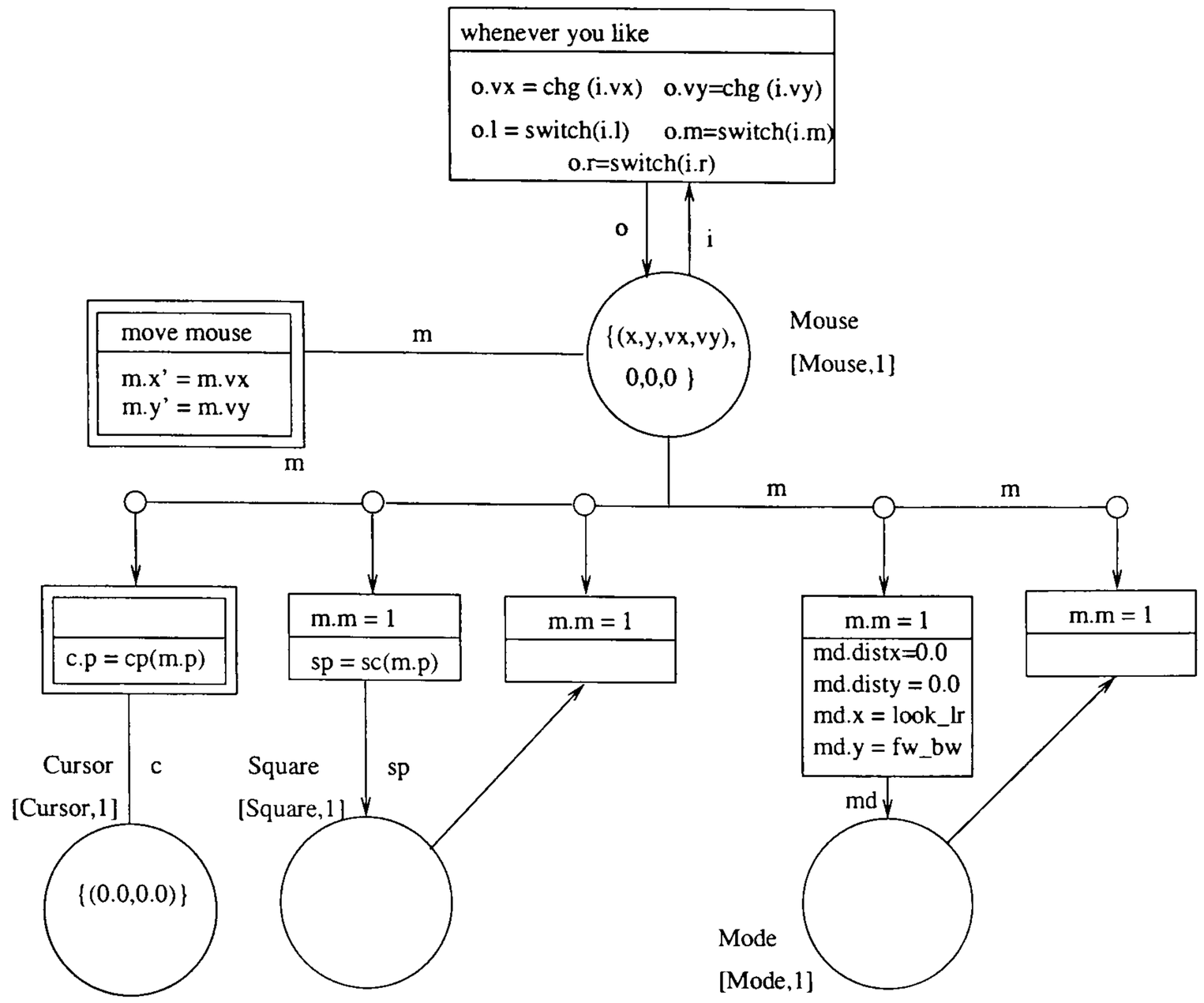

Figure 2.3: Partial HyNet specification of the mouse-based flying interaction technique (taken from [Massink, Duke, and Smith 1999])

of the mouse. This token is used in the expanded specification to render a cursor on the screen to reflect the position of the mouse. When the *Mouse* token records that a mouse button has been pressed ($m.m = 1$) a discrete transition places a (simple) token in the *Square* state, this is immediately removed by a further transition. A token in the *Square* is used in the expanded specification to render a square to the user to mark the origin of navigation. The *Mode* state is used in the expanded specification to record how the direction of navigation through the environment. A (simple) token is generated by a transition and placed in the *Mode* state when a mouse button has been pressed. This token is immediately consumed by a further transition. Figure 2.4 describes a complete HyNet specification of the mouse-based flying interaction technique taken from [Massink, Duke, and Smith 1999]. In addition to describing the behaviour of the mouse and the interaction technique itself, this design details the projection of the environment onto the screen (as indicated in the figure).

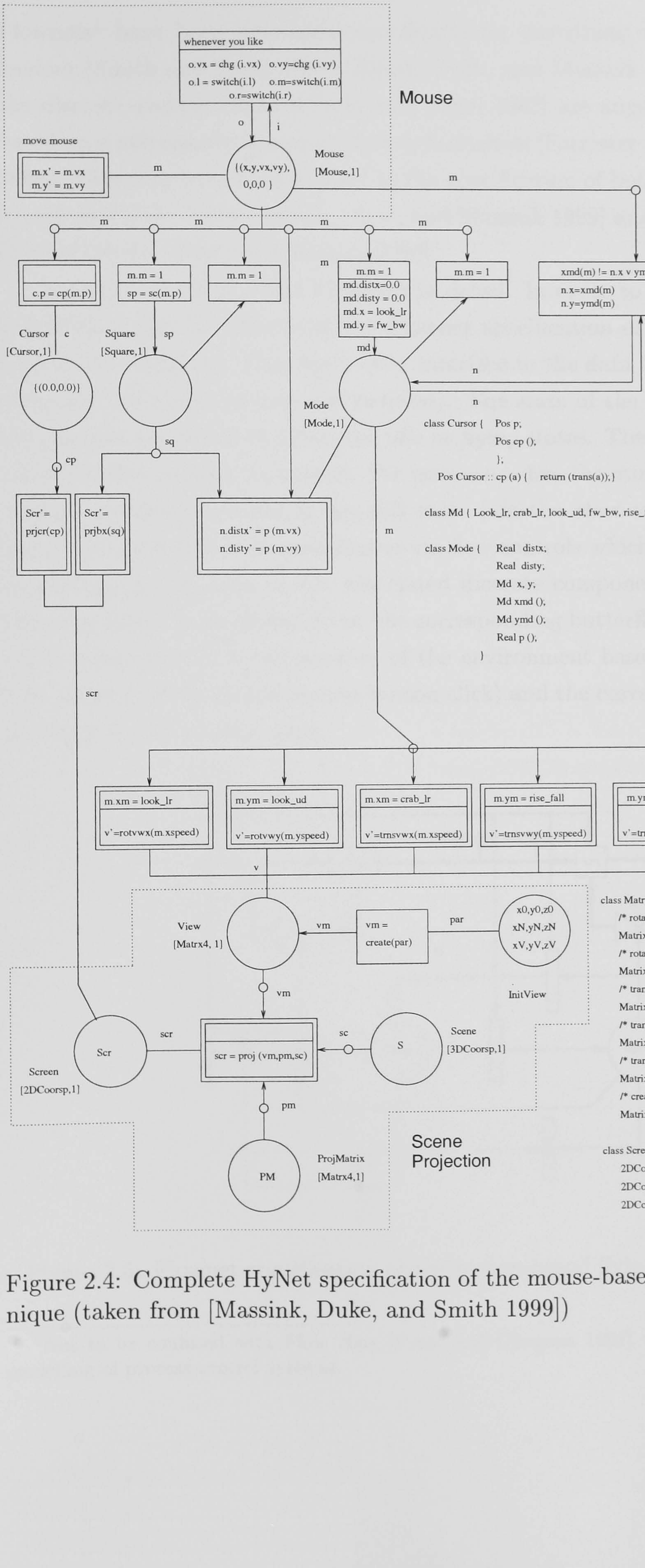

Figure 2.4: Complete HyNet specification of the mouse-based flying interaction technique (taken from [Massink, Duke, and Smith 1999])

2.1.2 Flownet

Flownets[1] have been developed specifically for describing virtual environment behaviour [Smith and Duke 1999b; Smith, Duke, and Massink 1999]. Within Flownets the discrete event/condition Petri-nets [Petri 1962] are augmented using constructs based on a systems dynamics modelling formalism [Forrester 1961] for the continuous detail. Flownets have been applied to the specification of both interaction techniques [Smith and Duke 1999b; Smith, Duke, and Massink 1999] and the behaviour of world objects [Smith, Duke, and Willans 2000].

In chapter 3 we describe Flownets in detail. In order to give an overview of the formalism figure 2.5 illustrates the Flownet specification of the mouse-based flying interaction technique. This has a clear interface to the data flowing in and out of the technique via plugs (*mouse* and *position*). The state of the technique is denoted by the presence of a token in either the *idle* or *flying* states. The continuous behaviour is related to the discrete via sensors. For instance, when the *middle mouse button* sensor triggers, a token is placed in the *idle* state (via the *start* transition). The discrete behaviour is related to the continuous via flow controls which are enabled or disabled depending on the state of the associated discrete component. For instance, when there is a token in the *flying* state, the corresponding butterfly flow control is enabled which transforms ( ) the position of the environment based on the origin position (the position of the middle mouse button click) and the current mouse position. This is output to the *position* plug.

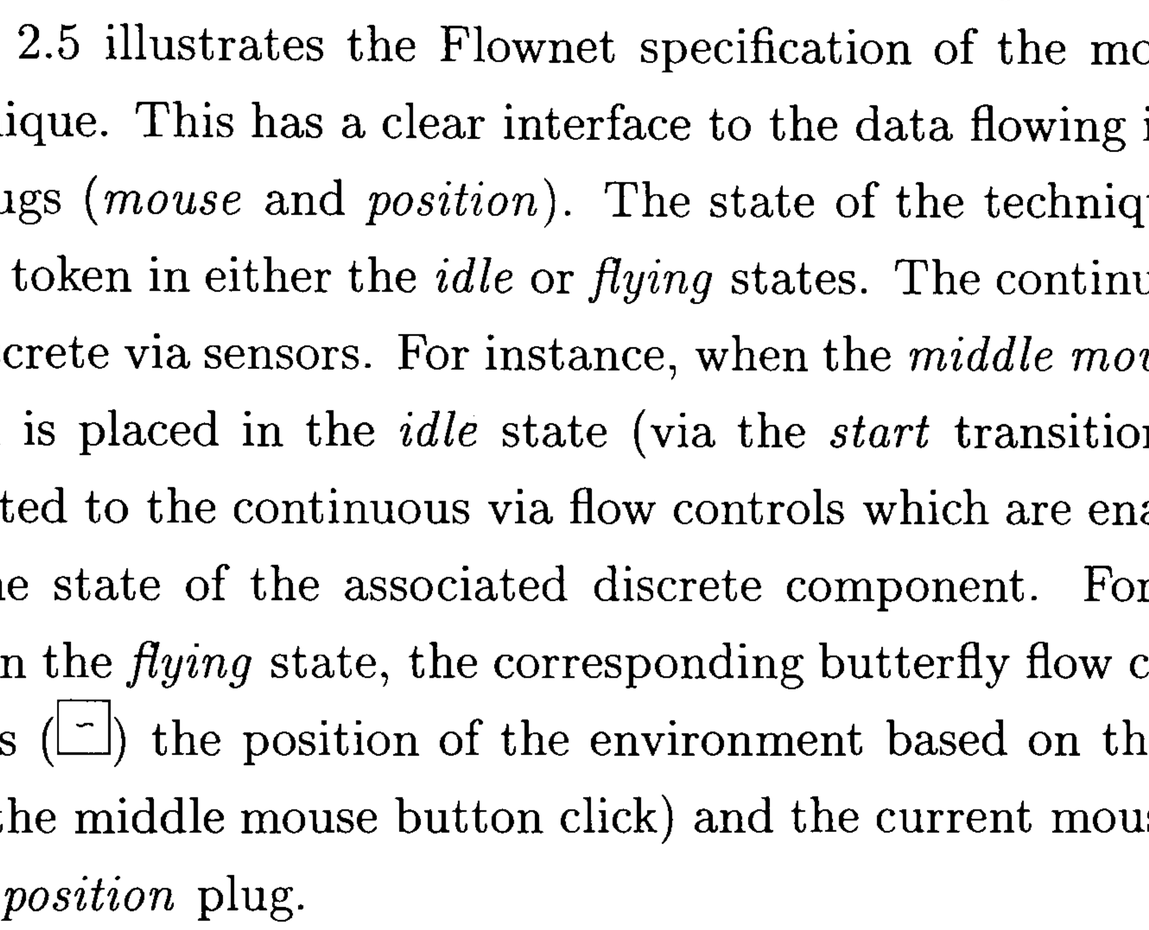

Figure 2.5: Flownet specification of the mouse-based flying interaction technique

[1]Not to be confused with Flow Nets [Flaus and Ollagnon 1997] which is used for the hybrid modelling of process control systems.

2.1.3 Tufts formalism

The formalism developed by Jacob et.al and presented in [Jacob 1996; Jacob, Deligiannidis, and Morrison 1999; Morrison and Jacob 1998] was also developed specifically for the specification of virtual environment behaviour at Tufts University. Within this the discrete components are described using state transition diagrams and the continuous components using links and variables. Unlike HyNets and Flownets, there is no diagrammatic relation between the two representations, this relation is achieved by the cross referencing of variables.

Figure 2.6 shows the specification of mouse-based flying using the Tufts formalism. The lower part shows the state transition diagram which specifies the three discrete states that the technique can be in (*inactive*, *idle* and *flying*). The upper part describes how data stored in variables (circles) is continually transformed by links (square boxes) when they are enabled. Links are enabled when the current discrete state matches their *identity* (this plays the same role as a Flownet flow control). For instance, when the user is in the discrete state of *flying* the link labelled *FLYING* is enabled. This allows information contained in the *mouse* and *originPos* variables to flow into the *position* variable.

The relation between the continuous and discrete part enables the firing of discrete transitions (and is the equivalent of a Flownet sensor). These are defined by functions on continuous variables. For instance, the function *MOUSE.pos(outorigin)* describes a threshold on the continuous *mouse* variable which detects when the position of the mouse has moved away from the origin position (the function itself, such as *pos(outorigin)*, is not explicitly captured in the formalism). When the threshold *MOUSE.pos(outorigin)* occurs, and the technique is in the *idle* state, the state of behaviour will change from *idle* to *flying*.

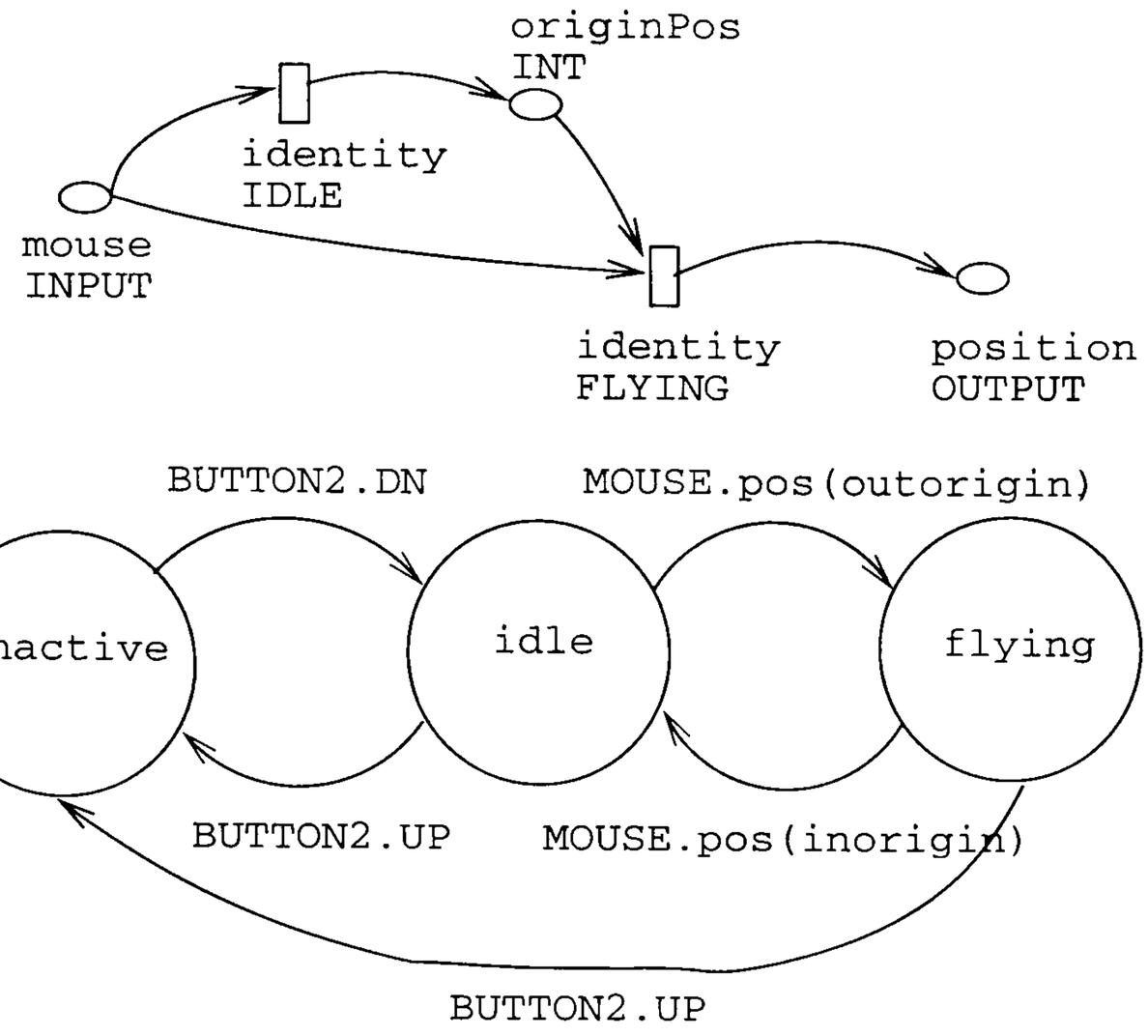

Figure 2.6: Specification of mouse-based flying using the Tufts formalism

2.1.4 Discussion

In the previous sections we have described three approaches to specifying virtual environment behaviour using hybrid formalisms. Although we have not described the formalisms exhaustively we have given enough detail to illustrate the main concepts and their use in modelling the mouse-based flying interaction technique. Of the three formalisms, HyNet is the most complex to understand and relate to the informal textual description of the interaction technique. However, this is not a fair comparison since this design also includes that of the mouse input device and the projection of the environment as output to the user. Even taking this into consideration, HyNet is still the most detailed specification because of its inclusion of precise details of the data flowing around the specification. While it remains a purely behavioural specification and does not include many details required for its implementation, concepts such as the concrete data description are similar to those used in an implementation rather those in the requirements. This kind of detail becomes more important as a design is refined towards a final implementation, however it is less critical in the earlier stages of design. As such, HyNet value is better placed as an implementation design formalism rather than one to be used for initial designs. This is an opinion also expressed in [Smith, Duke, and Massink 1999].

Flownets and the Tufts formalism are similar in many respects. They both make a clear distinction between those concepts which are continuous in nature and those that are discrete. They both use a standard notation to describe the discrete behaviour.

However, Flownets are advantageous because they use a concurrent formalism to describe the discrete component. The reasons for this will now be discussed.

Concurrency

In the case of interaction techniques the discrete part of the formalisms describes the state of the user and how their input should be interpreted (the mode of interaction). Often interaction techniques for virtual environments are multi-modal where the user is in multiple states concurrently. For instance, in the head-butt zoom interaction technique [Mine, Brooks Jr, and Sequin 1997] the users uses their hands to form a viewing window and the location of their head to zoom in and out of the window. The hands of the user can be in a number of states (form window, window formed and resize window) and the head of the user can be in a number of states (zoom out, static and zoom in) concurrently. These concurrent states are often dependent on each other, for instance the window must be formed in order to be able to zoom.

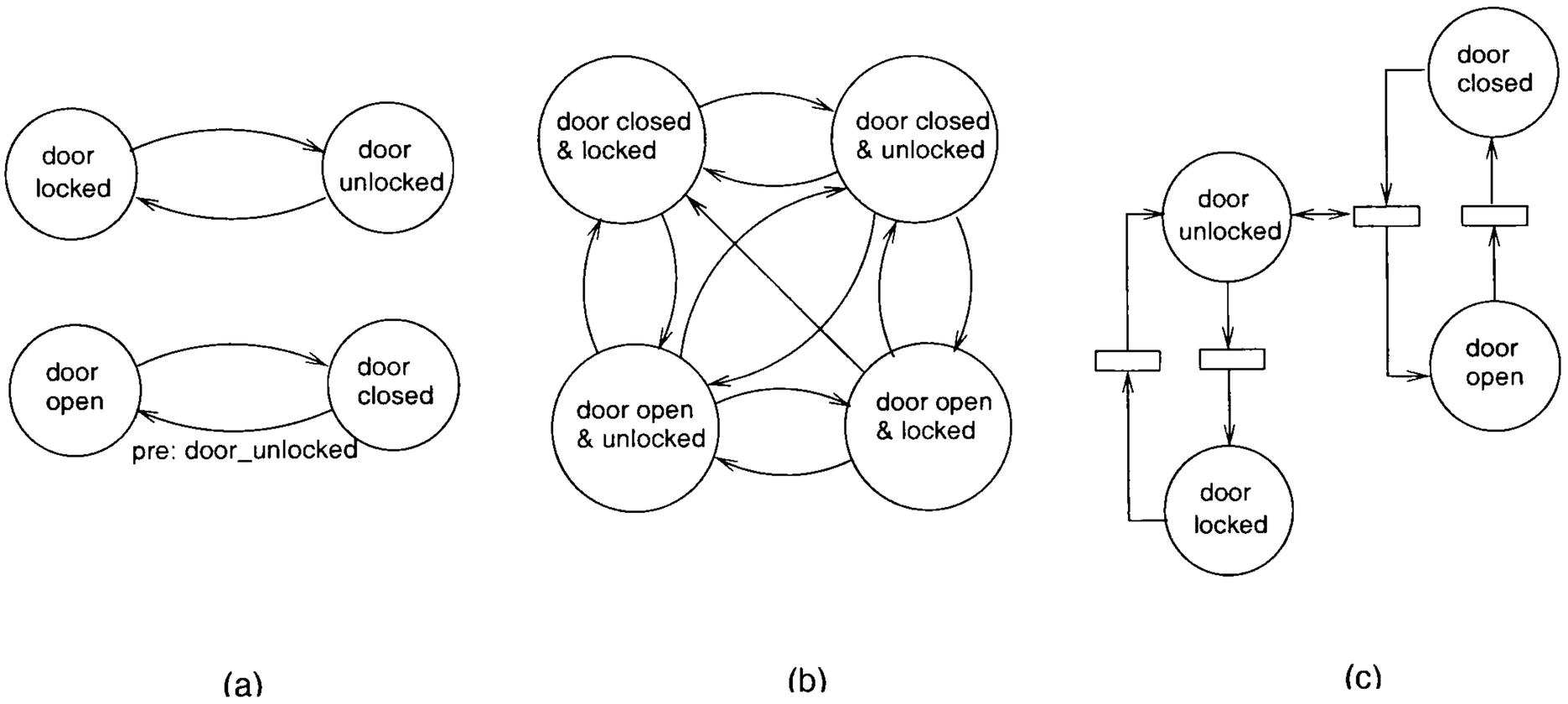

Figure 2.7: A comparison of using state transition diagrams with Petri-nets to describe dependencies between concurrent discrete state behaviour

For world objects the discrete part of the formalism describes the state of the objects and how input should be interpreted. World objects often have many components that behave independently but which have dependencies on one another, in a similar manner to interaction techniques. For instance a locking door may consist of the door itself and a lock. The door can be in a number of states (open and closed) and the lock can be in a number of states (locked and unlocked) concurrently. A dependency may exist describing that the door cannot be opened when locked.

The Tufts formalism uses state transition diagrams to describe the discrete components. Although state transition diagrams do not describe concurrent state behaviour, a number of techniques can be applied to bypass this limitation. One approach is to model the concurrent components as separate diagrams. Using this approach it is nec-

essary to describe dependencies between diagrams using pre-conditions (or guards) on the transitions which reference other diagrams. The pre-condition must be satisfied in order for the transition to take place. This approach succeeds with interaction techniques because these usually have a small (discrete) state space. However, world objects often have a much larger state space reflecting the complexity of real world objects. This makes it difficult to comprehend the resulting design. An alternative approach is to interleave behaviours. Again, for interaction techniques this can work (and is the approach adopted by the Tufts formalism in [Morrison and Jacob 1998]), however with the larger state space of world objects (not explored for the Tufts formalism) this results in an of states and transitions. An example of using these two techniques is illustrated in figure 2.7 (a) and (b) for a locking door world object. Essentially using state transition diagrams to model concurrent behaviour is going against the grain of the formalism. This limitation of state transition diagrams is also expressed in [Foley, van Dam, Feiner, and Hughes 1990, p458-459] in the context of modern interfaces generally.

The Flownet formalism uses Petri-nets to describe the discrete components. Petri-nets were designed to overcome the inability of sequential formalisms (state transition diagrams, for example) to describe concurrency. Consequently it can model with ease the concurrent state behaviour of interaction techniques and world objects. The locking door world object is described using a Petri-net in figure 2.7 (c).

2.1.5 Summary

In this section we have drawn a number of conclusions:

- Three hybrid formalisms have been developed for (or applied to) the specification of virtual environment behaviour: HyNet, Flownets and the Tufts formalism.
- The value of HyNet is as an implementation specification formalism rather than for the description of initial designs.
- Although Flownets and the Tufts formalism are similar, Flownets supports the specification of discrete concurrency which is important for the description of virtual environment behaviour.

2.2 Prototyping designs

In this section we examine efforts to translate design specifications of behaviour into an implementation of the behaviour. First, we examine approaches to achieving this with

specifications of traditional interfaces. Secondly, we examine approaches to achieving this specifically for behavioural specifications relating to virtual environments.

2.2.1 Traditional approaches

UIMS

The challenge addressed by user interface management systems (UIMS) is to separate the semantics of the user interface from the application. This allows the interface to be designed independent of application concerns. In addition, different interfaces can be designed for the same application (this is useful when there are multiple users with different concerns).

The most common interpretation of a UIMS is the Seeheim architecture [Pfaff 1985] shown in figure 2.8. This separates the user interface into three components. The presentation component which receives raw data from input devices and renders some interface (usually visually) to the user. The application interface component which communicates directly with the application. The dialogue component which manages the dialogue of interaction between the presentation and the application components. Efforts have largely focussed on describing the dialogue component using formalisms such as state transition diagrams [Denert 1977; Jacob 1986] and Statecharts [van Zijl and Mitton 1991; Wellner 1990; Lucena and Liesenberg 1994]. For the purposes of our discussion the dialogue component can be considered equivalent to what we call behavioural design specifications in section 2.1.

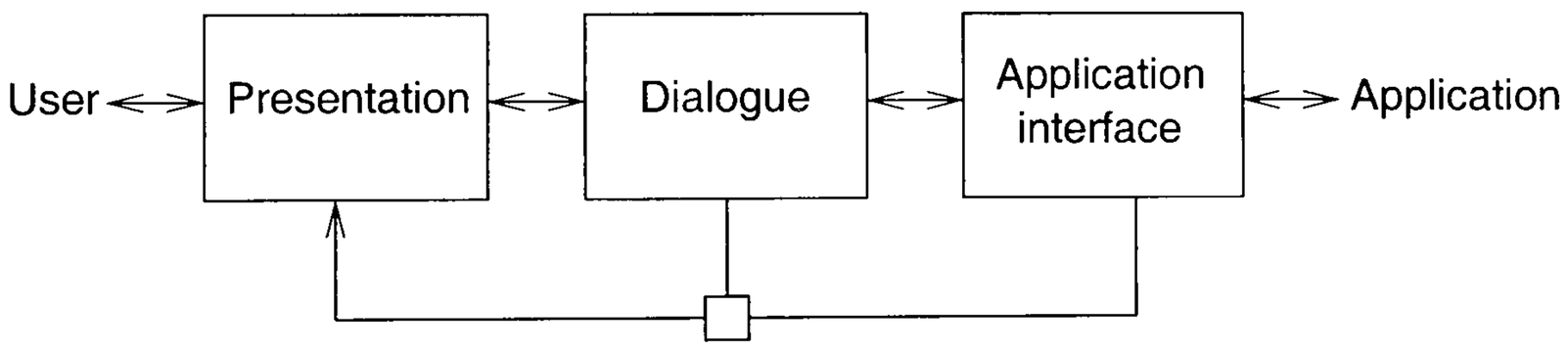

Figure 2.8: Seeheim UIMS Architecture [Pfaff 1985]

A review of the various realisations of UIMS is presented in [Beaudouin-Lafon 1994]. However, the general approach is consistent. We will illustrate this using the dialogue description shown in figure 2.9 (taken from [Olson 1992, p37]). This describes how the user can draw a line or a rectangle interactively. The dialogue receives (logical) events from the user, in figure 2.9 these can be *Line*, *Rectangle* and *MouseDown*. These events change the state of the dialogue according to its current state and also call actions, for example *P1:=MouseLoc*, *DrawLine* and *DrawRect*.

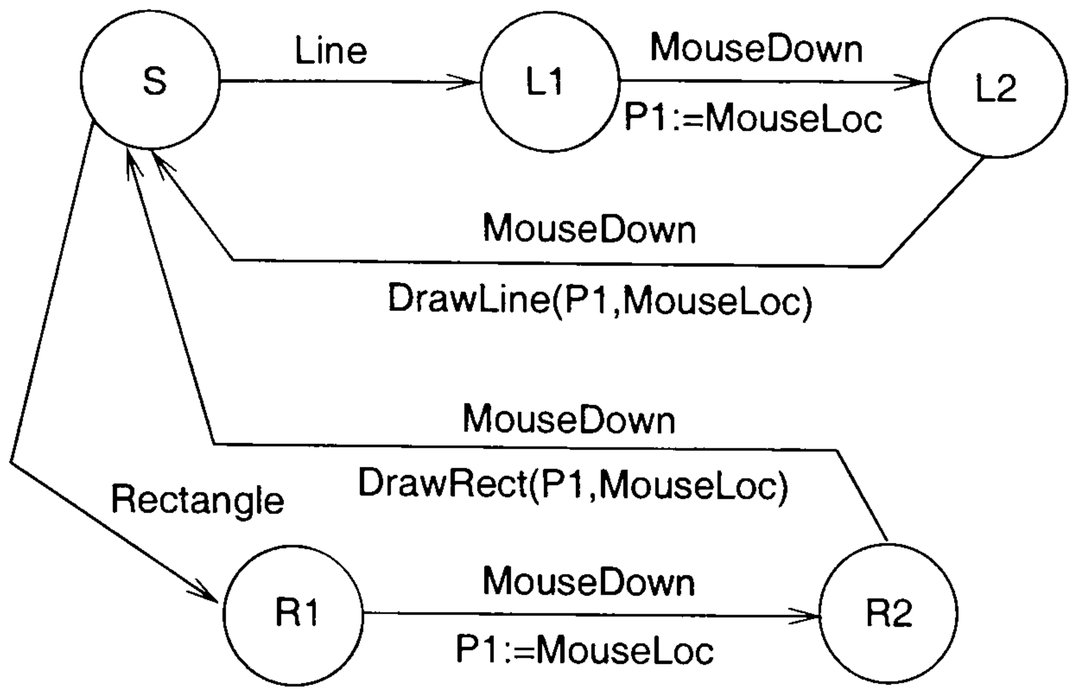

Figure 2.9: An example of a UIMS dialogue specification (taken from [Olson 1992])

The main application contains a loop which continually propagates events to the dialogue and checks for actions from the dialogue (figure 2.10, also taken from [Olson 1992, p39]).

```
CurrentState := S;
Repeat
{
   GetEvent(E);
   Select a transition T using CurrentState and E
   DoCommand(Action(T));
   CurrentState:=NextState(T);
}
```

Figure 2.10: An example of an application interface component within a UIMS (taken from [Olson 1992])

Statemate

Statemate [Harel, Lachover, Naaad, Pnueli, Politi, Sherman, Shtull-Trauring, and Trakhtenbrot 1990] is a tool which supports the prototyping of Statechart [Harel 1987] specifications. Within Statemate, transitions can be linked to events from widgets such as buttons, and variables can be linked to functions on display widgets. As the user interacts with the widgets they behave according to the Statechart specification. In this way, Statemate can be seen as a form of UIMS. However, unlike typical UIMS, the presentation of the prototype is not the actual intended presentation of the specification. For instance, the consequence of interaction with a behavioural design for an aircraft interface might be explored using the limited widgets supplied with Statemate (figure 2.11) rather than the devices of the real aircraft. This style of prototyping is more commonly known as specification animation and is also used in [Systä 1995] for evaluating formal specifications of user interfaces.

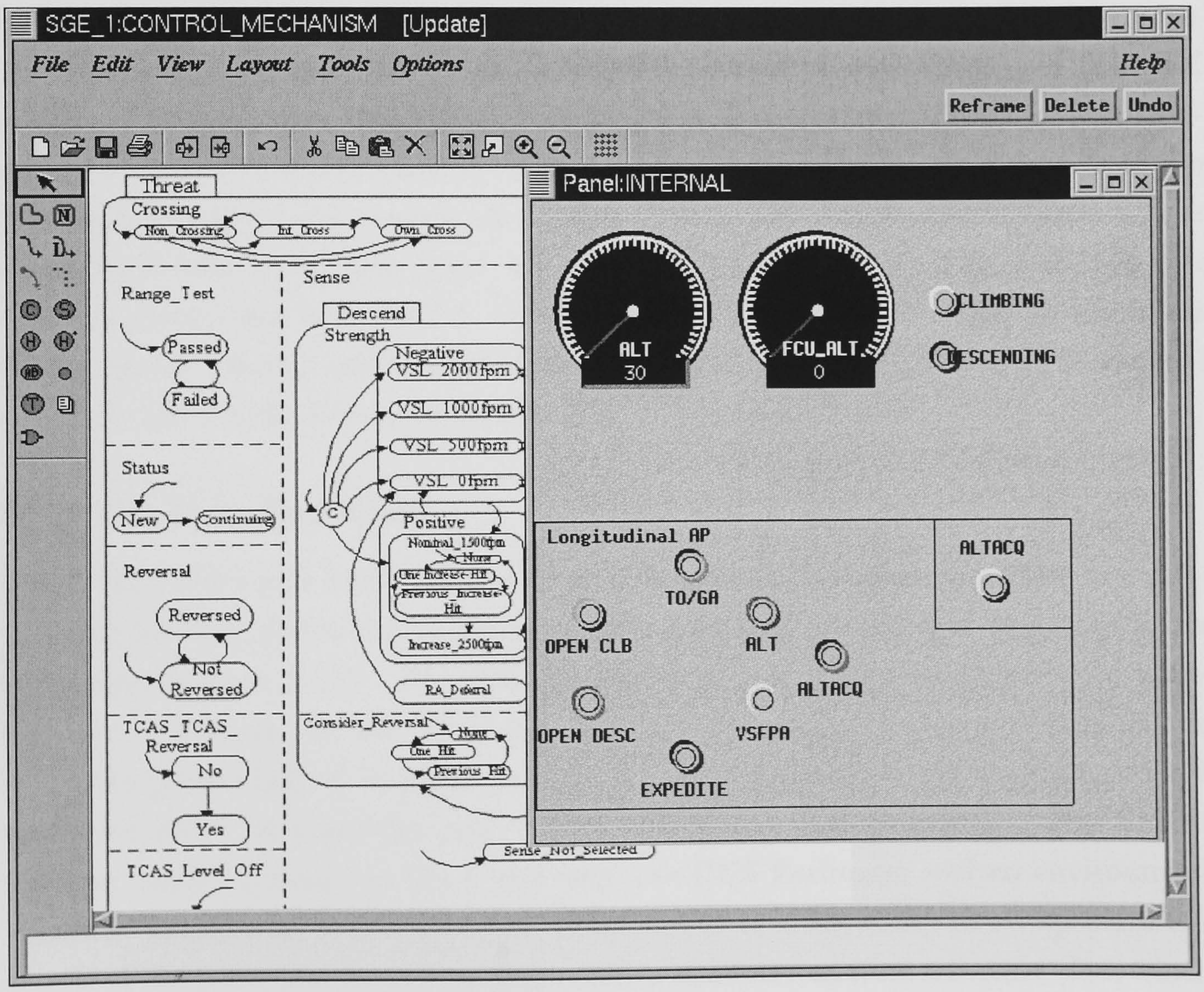

Figure 2.11: Animating a Statechart specification using the Statemate tool [Harel, Lachover, Naaad, Pnueli, Politi, Sherman, Shtull-Trauring, and Trakhtenbrot 1990]

2.2.2 Virtual environment approaches

The approach presented in [Kim, Kang, Kim, and Lee 1998] uses an existing tool for building real-time system models (ASADAL/PROTO), the authors claim the ability to generate virtual environment prototypes of the models, but conclude with a proposal of code generation as future work (there is no detail of how this might be achieved). However [van Schooten, Donk, and Zwiers 1999] and [Morrison and Jacob 1998; Jacob, Deligiannidis, and Morrison 1999] present concrete examples of the transition from behavioural designs to implementations.

From CSP

[van Schooten, Donk, and Zwiers 1999] describes how a behavioural specification described using concurrent sequential processes (CSP) [Hoare 1978] can be used to prototype a virtual environment based on an earlier approach for traditional interfaces [Alexander 1990]. This is achieved by two communicating processes. A C program implements the CSP engine and the presentation component is realised using a TCL/TK process. In the CSP specification some processes are allocated special virtual environment communication channels. These processes can then handle events from the presentation and/or call commands in the presentation. Illustrated in figure 2.12 is an example taken from [van Schooten, Donk, and Zwiers 1999]. This describes how the behavioural specification for *TableClosed* and *TableOpen* is linked to the input events *open()*, *close()* and *usertablerefresh*. Additionally there is an output channel *userclick* which calls a function to create a button labelled *click*.

From the Tufts formalism

PMIW [Morrison and Jacob 1998; Jacob, Deligiannidis, and Morrison 1999] is a software model to support the implementation of designs constructed using the Tufts formalism (section 2.1.3). The VRED[2] editor was developed to support the specification of designs. It was originally intended that this editor would support the automatic generation of implementations, but this is not currently the case and the transition must be manually achieved by coding the design directly. The PMIW software model is based in C++ and uses the IRIS Performer virtual environment libraries. In order to realise a design using PMIW it is necessary to encapsulate the details of the behaviour in a class. This class is instantiated from a further class which is also responsible for constructing the remainder of the environment and the run-time maintenance of the environment. We will illustrate the use of PMIW using the mouse-based flying specification in figure 2.6. The partial code for the behaviour class

[2]The meaning of the PMIW and VRED acronyms are unclear.

```
Table {
  type [window]
  input{
    usertableopen    { receive [open()] }
    usertableclose   { receive [close()] }
    usertablerefresh { receive [showtext("Table filled")] }
  }
  output{ userclick {
    init [createbutton("click")]
  }
} = TableClosed

TableClosed =
      ( opentable -> usertableopen -> TableOpen )
  [] ( closetable -> TableClosed )

TableOpen =
      ( opentable -> usertablerefresh -> TableOpen )
  [] ( closetable -> usertableclose -> TableClosed )
  [] ( userclick -> textout2_perftime -> TableOpen )
```

Figure 2.12: CSP description of channels linking the behaviour to an implementation (taken from [van Schooten, Donk, and Zwiers 1999])

for mouse-based flying is shown in figure 2.13 (the complete code is approximately three times its length).

In the behaviour class, the variables are defined and links are instantiated. This is illustrated for the variable to record the origin position (line 5), the link to update the origin position (line 12) and the link to update the position of the environment (line 16). Each link has its own class which contains a constructor and an evaluation function. The constructor adds input and output variables to lists (lines 50-56), and the evaluate function describes how the variables in these lists are transformed (lines 62-64). The discrete part of the design is described using a special language (originally developed for [Jacob 1986]) embedded in the C++ code, which is parsed and translated prior to compilation. This is illustrated in figure 2.13 for the three discrete states of mouse-based flying (lines 23-37). The discrete behaviour is linked to the continuous variables (lines 26,27) in order for the variables to be enabled as a result of discrete interaction.

A behavioural class is instantiated in a second class. The second class is responsible for the construction of the environment including initialising world objects and adding these to data structures, and the initialisation of devices and ensuring that these are polled at runtime.

```
1  ************************************************************
2  * instantiate continuous variables
3  ************************************************************
4
5  originPos = new Variable<pfVec3>(VariableBaseAll::INT);
6
7  ************************************************************
8  * instantiate continuous links
9  ************************************************************
10
11 extern MouseCursor *mouse;
12 Link *identity = new LinkRecordOrigin(mouse, originPos);
13 identity->SetName("LinkRecordorigin");
14
15 extern PfWindow *pfwindow;
16 Link *identity = new LinkFlying(originPos, mouse, pfwindow);
17 identity->SetName("LinkFlying");
18
19 ************************************************************
20 * state transition event handlers (parsed before compile)
21 ************************************************************
22
23 bool mbf::IhIo(Token token) {
24 <std>
25 mbf->end
26
27 inactive:        BUTTON2.DN -> idle
28
29 Enable: recordOrigin
30 idle:            BUTTON2.UP -> inactive
31                  Cond: originEvent->outOrigin(); -> outOforigin
32
33 Enable: flying:
34 outOforigin     BUTTON2.UP -> inactive
35                  Cond: originEvent->inOrigin(); -> recordOrigin
36 ;
37 </std>
38 }
39 ************************************************************
40 * link discrete events to continuous links
41 ************************************************************
42
43 recordOrigin = new Condition(LinkRecordOrigin);
44 flying = new Condition(LinkFlying);
45
46 ************************************************************
47 * constructors for links
48 ************************************************************
49
50 mbf::LinkRecordOrigin::LinkRecordOrigin(Variable<pfVec3> *src,
51                                         Variable<pfVec3> *dest)
52         assert(src != NULL);
53         assert(dest != NULL);
54         ins->Add(src);
55         outs->Add(dest);
56 }
57
58 ************************************************************
59 * transformation functions for links
60 ************************************************************
61
62 void mbf::LinkRecordOrigin::Evaluate() {
63         dest->SetI(src->getI);
64 }
```

Figure 2.13: Partial listing for the mouse-based flying interaction technique specified using PMIW

2.2.3 Discussion

An important part of the prototyping approaches previously discussed is the linking of the behaviour (or the dialogue) to the presentation which defines the concrete interface to the user. The presentation is made up of a number of concepts. Input devices receive interaction from the user, and the system renders some state via output devices. Usually the rendering of this state takes place visually. For virtual environments the visual rendering of the environment is described using world objects constructed using a 3D modeller. This taxonomy is illustrated in figure 2.14.

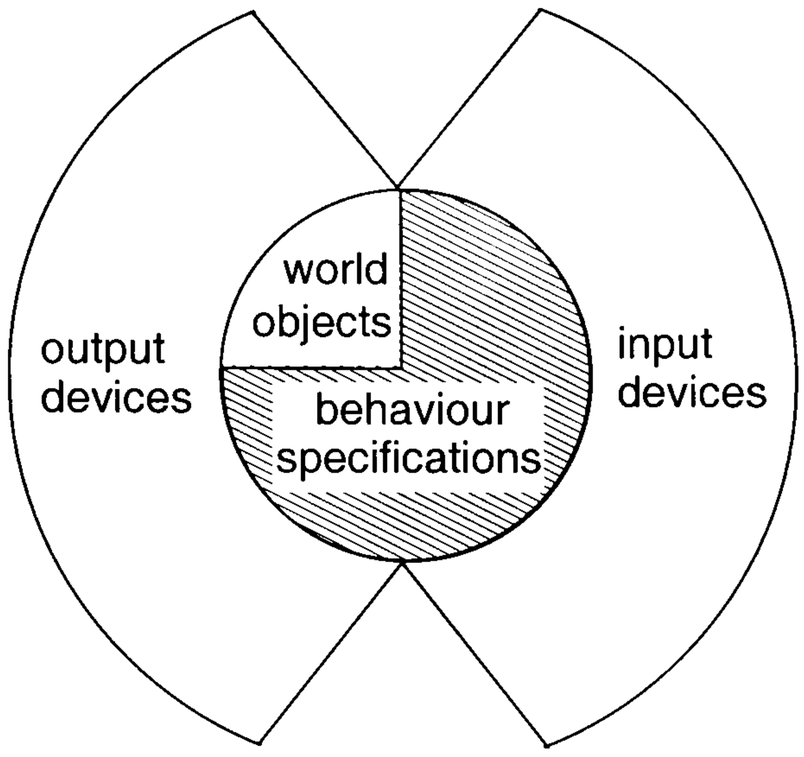

Figure 2.14: The presentation concepts for a virtual environments and their relation to the behaviour

The UIMS approach provides a means of implementing specifications by linking the specifications to events and actions provided by the presentation. The focus of the effort has been on the exploration of languages for describing the dialogue (behavioural) component. The definition of the presentation component is usually described in an underlying toolkit. This toolkit describes the events and actions that the presentation supports, and the binding between these and the concrete realisation of the presentation concepts. In order to change the bindings it is necessary to alter the toolkit program code definition. However, rarely does this need to happen in interfaces such as WIMPs because of their consistent device interface (mouse and keyboard) and rendering to the user (windows and widgets).

Similarly, the Statemate approach links specifications to a consistent presentation of widgets and devices via events and actions. The important difference is that the presentation of the prototype (animation) is not (usually) the intended final presentation. The input devices and appearance of display may all be different in the final implemented design. In that context, the Statemate approach supports the designer

in evaluating the result of interaction. What can not be evaluated is the suitability of the behavioural design in the context of the intended presentation.

The presentation of virtual environments is not consistent between applications. Limited Statemate style insight can be gained by using consistent presentation concepts that are crude approximations to the intended final context. This is the approach used to prototype (animate) CSP specifications where TCL/TK widgets are used. However, in order to gain a deeper understanding of the designs it is also important to be able to explore the presentation of the behaviour. For instance, to evaluate the suitability of a navigation interaction technique design in its intended context; or to adapt an interaction technique design in order to account for characteristics of input devices (to introduce constraints, for instance). Furthermore, in a prototyping context it is desirable to be able to explore *alternative* presentations. For example, to experiment with different devices and interaction technique combinations.

The Tufts approach is important because it demonstrates that the abstractions used in hybrid specifications of virtual environment behaviour can also be used in the implementation of designs. This approach is built on top of a virtual environment library (Performer) which enables the intended presentation of the virtual environment to be implemented with the behaviour. Despite the strengths of the Tufts approach, a number of shortcomings can be identified:

- The approach does not support Flownets. We have argued (section 2.1.4) that the concurrent nature of the discrete component of Flownets is important for the design specification of virtual environment behaviour.

- The transition between design and implementation must take place manually. Consequently semantics must be specified twice, this is time consuming and may result in translation errors.

- In making the transition, the designer is burdened with low-level implementation issues. For instance, the management of data structures to hold world objects and the polling of devices. These issues are important for a final implementation where performance and scalability must also be considered. However, in a prototyping context it is desirable to abstract from these.

2.3 Analysing designs

An important focus of interactive system research has been the formal analysis of design specifications. Within this domain, a number of techniques have been developed which support the checking of desirable requirements within a design specification. For instance, within a formal design specification of an air-traffic control system, a

desirable requirement might be that two planes cannot occupy the same air space simultaneously. This can be formalised and proved to exist. There are also a number of generically desirable requirements which can be analysed in a similar manner. For example, *undoability*: the ability for the user to undo any action they perform. The application of this style of analysis can be seen in [Abowd 1991; Paternó 1995].

The power of design analysis is that it is exhaustive with a high degree of certainty that the requirements hold within the design, this is particularly the case when it is automatically applied. This can be achieved by using model checking [Clarke Jr., Grumberg, and Peled 1999] as presented in [Campos 2000]. The application of automation to design analysis often has the added advantage of being rapid. The potential to (automatically) apply analysis techniques to the domain of virtual environments is discussed in [Massink, Duke, and Smith 1999], although no application is illustrated.

2.4 Conclusion

In this chapter we have reviewed past work concerning the specification of virtual environment behaviour, and the explorations of such designs using prototypes and specification analysis. The discussions of this work has motivated the aims of this thesis:

- We want to provide a translation from Flownet design specifications of virtual environment behaviour to a prototype.
  - The approach should support the exploration of presentations.
  - The approach should hide implementation concerns from the designer.
- We want to explore the analysis of design specifications.

Chapter 3

Flownets

As described in the previous chapter, Flownets [Smith, Duke, and Massink 1999; Smith and Duke 1999b] are an appropriate formalism for the design of virtual environment behaviour. In this chapter we further describe the formalism and discuss a semantics. Although our discussion is informal, a formal semantics for Flownets is given in appendix A using the Z specification language.

3.1 Discrete components

The discrete part of a Flownet is described using a condition-event Petri-net. Condition-event Petri-nets are based on original Petri-nets introduced in Petri's thesis [Petri 1962]. Flownets also use inhibitor arcs [Hack 1975] within the Petri-net component of the specification.

3.1.1 Basics

Shown in figure 3.1 is a simple Petri condition-event net example to illustrate the main concepts. A condition-event net is made up of places (p1,p2,p3) and transitions (t1,t2) related by arcs (we sometimes refer to these as discrete arcs to distinguish between these and continuous arcs which we introduce in section 3.2.1). The initial state (sometimes called the marking) of the net is defined by allocating tokens to places, these are represented by black dots (p3). Condition-event nets contain simple unstructured tokens, that is they represent conditions only rather than the more complex data tokens of high-level Petri-nets (see [Reisig 1982]). The behaviour of the net is defined by the movement of tokens around places via transitions. This is determined by the enabling and firing rules of transitions.

The enabling and firing rules of condition-event nets are illustrated by the two diagrams shown in figure 3.2. In order for a transition (t1 or t2) to be enabled, every

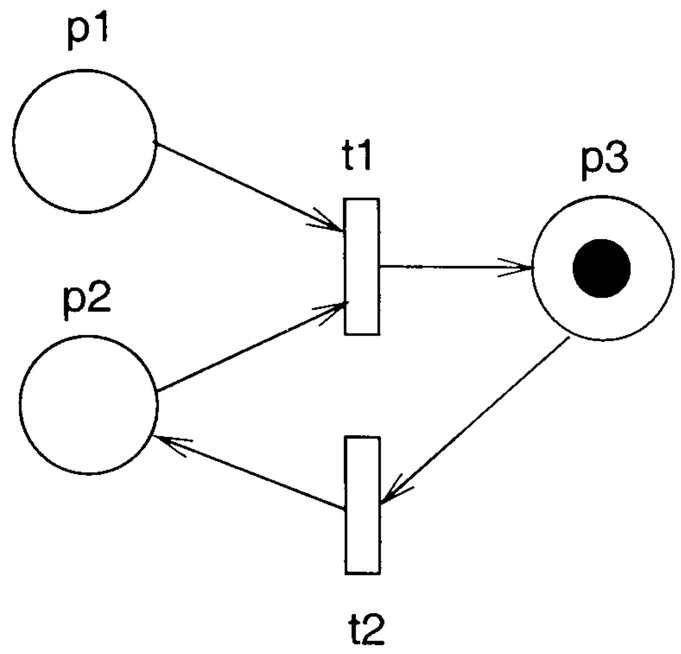

Figure 3.1: A simple condition-event Petri-net

place targeting the transition via an arc (p1 and p2, or p3) must contain a token. An enabled transition may fire whereupon a token is deposited in every place connected by an arc originating from the transition. In the case of figure 3.2 (a) a token would be placed in p3. In the case of figure 3.2 (b) a token would be placed in p5 and p6.

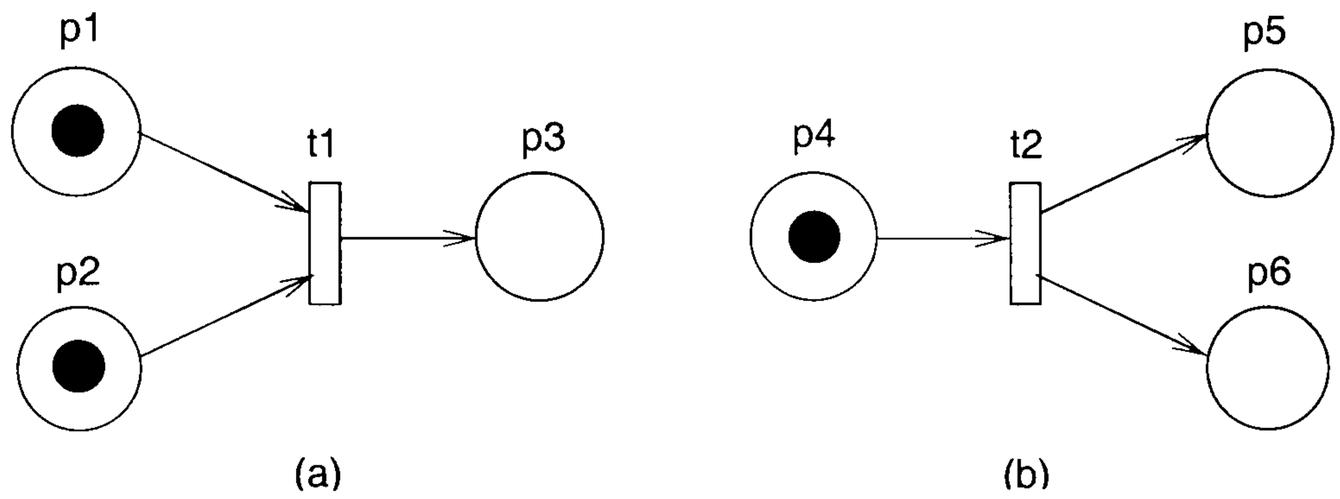

Figure 3.2: Examples to illustrate the firing rules of condition-event Petri-nets

With Petri-nets it is not possible to determine which transitions should be enabled when two or more can be enabled concurrently. While this non-determinism is maintained in Flownets to a great extent, it is necessary to introduce a mechanism by which priority is sometimes given to transitions. This is discussed in section 3.3.

3.1.2 Inhibitor arc

Flownets use the added construct of an inhibitor [Hack 1975] within the condition-event nets. An inhibitor arc specifies that the connected place must *not* contain a token in order for the transition to be enabled. An inhibitor can be used in combination with regular arcs as illustrated in figure 3.3. Within this, transition t1 will fire when there is a token in place p1 and there is *no* token in p2.

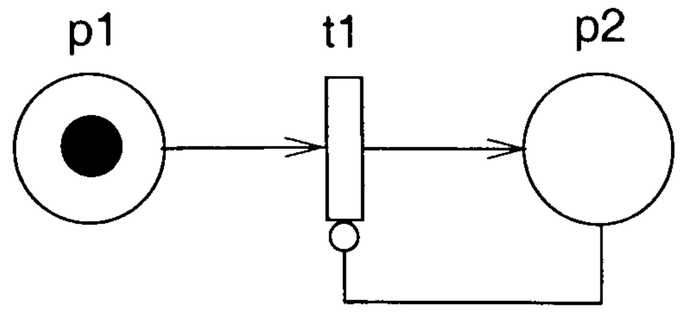

Figure 3.3: An example of an inhibitor arc

3.2 Continuous components

The continuous part of Flownets is based on a notation for modelling system dynamics presented in [Forrester 1961]. Within this, entities are modelled as continuous quantities interconnected in loops of information feedback and circular causality. An example of this notation can be seen in figure 3.4 taken from [Forrester 1961, p333]. The information source *allocation to development and design* is used as a condition for the continuous flowing of *ideas* through to *designs.*

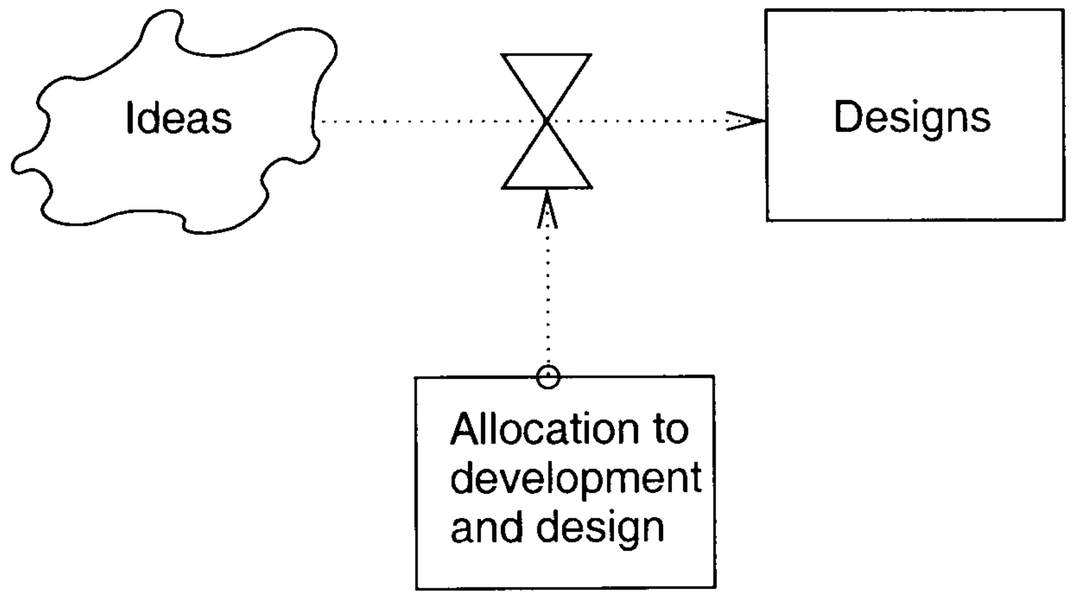

Figure 3.4: An example of systems dynamics modelling notation

3.2.1 Data input/output

A Flownet contains data links to and from the external environment called plugs. Plugs are labelled to describe the data to which they are linked. Continuous arcs originate from and terminate in plugs, and are visually depicted by thick lines ending in double headed arrows. Illustrated in figure 3.5 is an input and output plug linked to continuous arcs. Sometimes within a specification it is necessary to duplicate plugs for conciseness of specification. When this is the case, identical labels denote that two plugs are synonymous.

A plug input can also originate discrete data, for instance the clicking of a device button. As such, these can be associated directly with a discrete arc into the condition-event net as illustrated in figure 3.6.

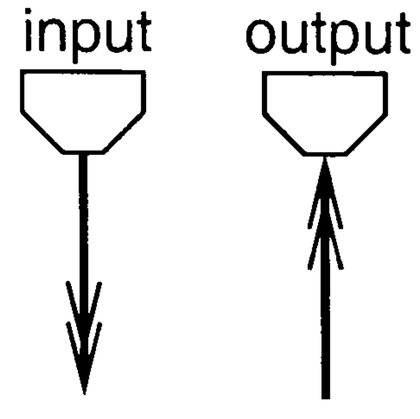

Figure 3.5: Flownet plugs linked to continuous arcs

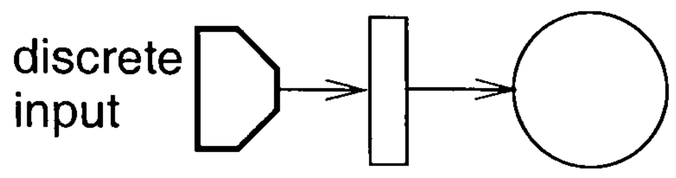

Figure 3.6: A plug link directly into the condition-event net

3.2.2 Continuous to discrete

Continuous data flow is related to the discrete condition-event net via sensors which represent some boolean condition on the data flow. A sensor can be targeted by one or more continuous arcs and can originate one or more discrete arc. Sensors are informally labelled to define a boolean threshold describing their firing condition. Illustrated in figure 3.7 are two examples of the use of a sensor to map continuous behaviour onto a discrete net. In figure 3.7 (a), transition t1 is enabled when *input* $= x$. In figure 3.7 (b), transition t2 is enabled when *input* $= x$ and there is a token in p2.

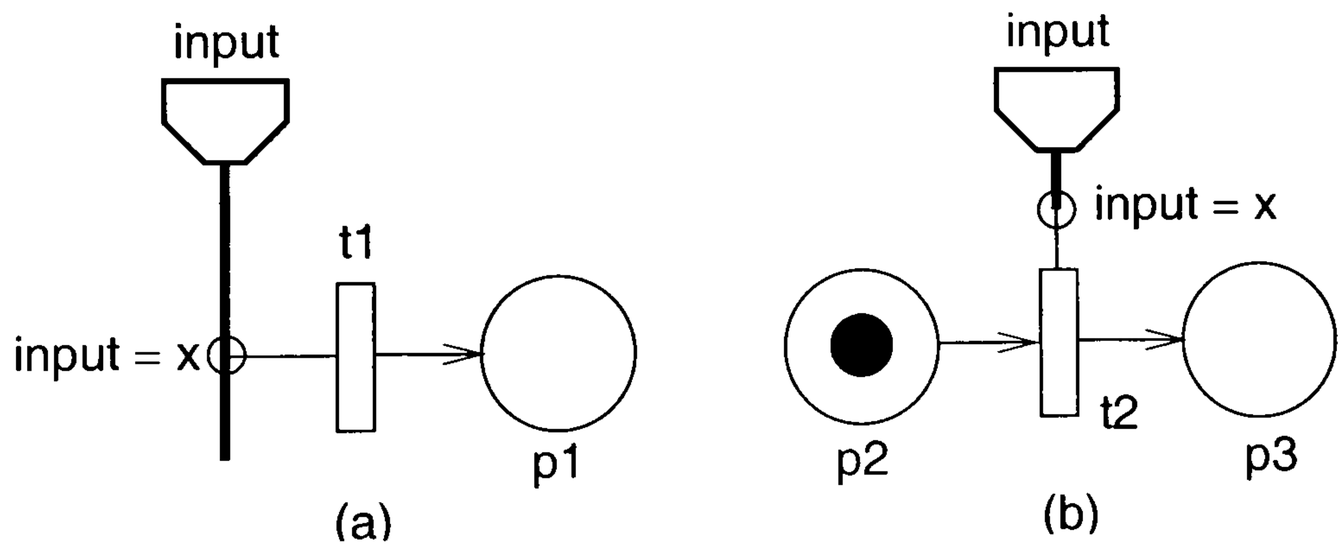

Figure 3.7: Example of sensors relating continuous and discrete behaviour

3.2.3 Discrete to continuous

The discrete condition-event net enables the flow of continuous data via flow controls. A flow control is targeted by arcs from places and/or transitions. It becomes enabled when one or more targeting transition or targeting place is enabled. A flow control is

also targeted by a continuous arc, and continuous arcs originate from a flow control. When a flow control is enabled data can pass continually from the input continuous arc to the output continuous arc. Illustrated in figure 3.8 are examples of the use of flow controls. In 3.8 (a), the flow control allows the flow of continuous data whenever there is a token in p1. In 3.8 (b), the flow control allows the flow of continuous data for each firing of transition t1.

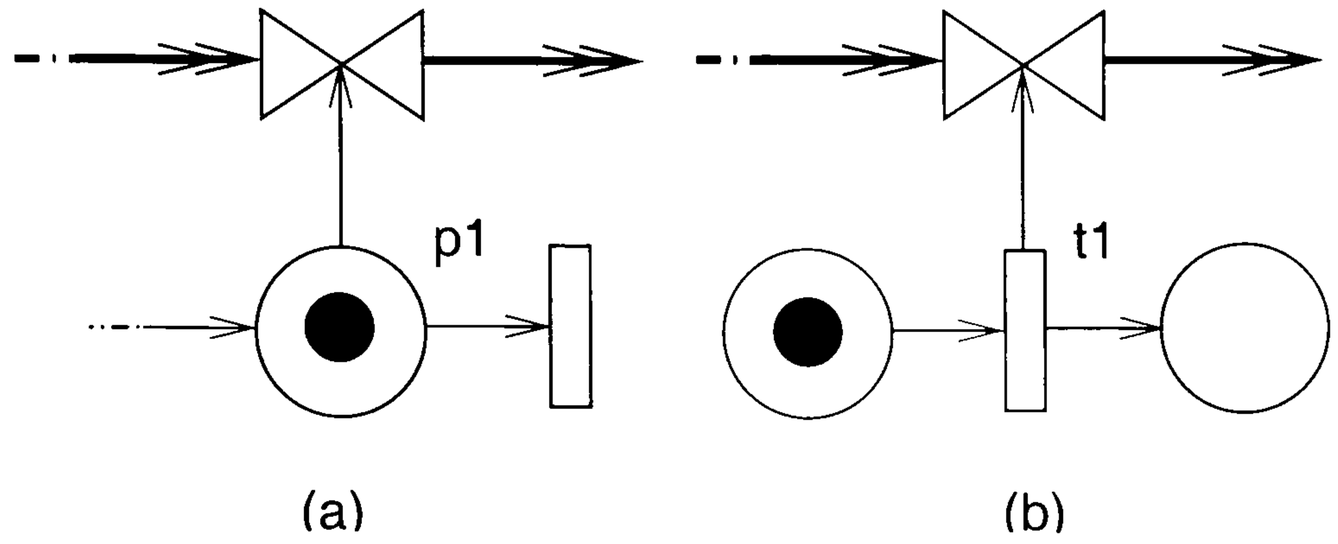

Figure 3.8: Example of flow controls relating discrete and continuous behaviour

3.2.4 Transforming and storing data

Continuous data is transformed by transformers. These are targeted by continuous arcs and are labelled to describe the transformation that they perform. There must also be a continuous arc originating that carries the resulting transformed data, this arc usually targets a store. A store is a repository of data which resides within a Flownet. Data is also communicated from stores via continuous arcs.

It is common to find transformers and stores used in the configuration illustrated in figure 3.9 where data *a* is continually transformed (via some transformation function) while the flow control (f) is enabled.

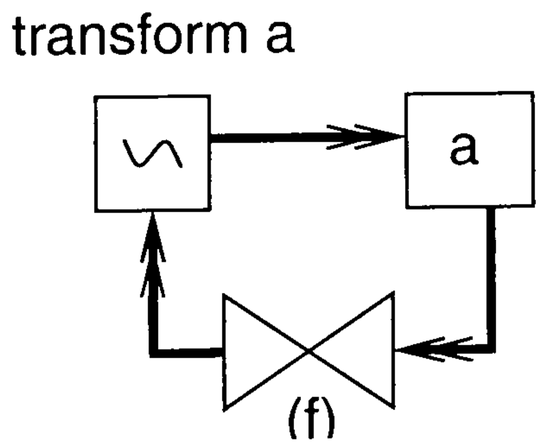

Figure 3.9: A common transformer store configuration

3.3 Dynamic behaviour

The execution of a Flownet is achieved through a number of operations which evaluate components and change their state based on the result of the evaluation. This division of operations is based on component groups. Evaluation of transitions is achieved by two operations. The first operation for interaction transitions, that is those transitions whose firing is influenced by the interaction of the user. This means that transitions targeted by either a sensor and/or a plug. Secondly, an operation for non-interaction transitions. These are transitions whose firing is solely dependent on the distribution of tokens in the Petri-net.

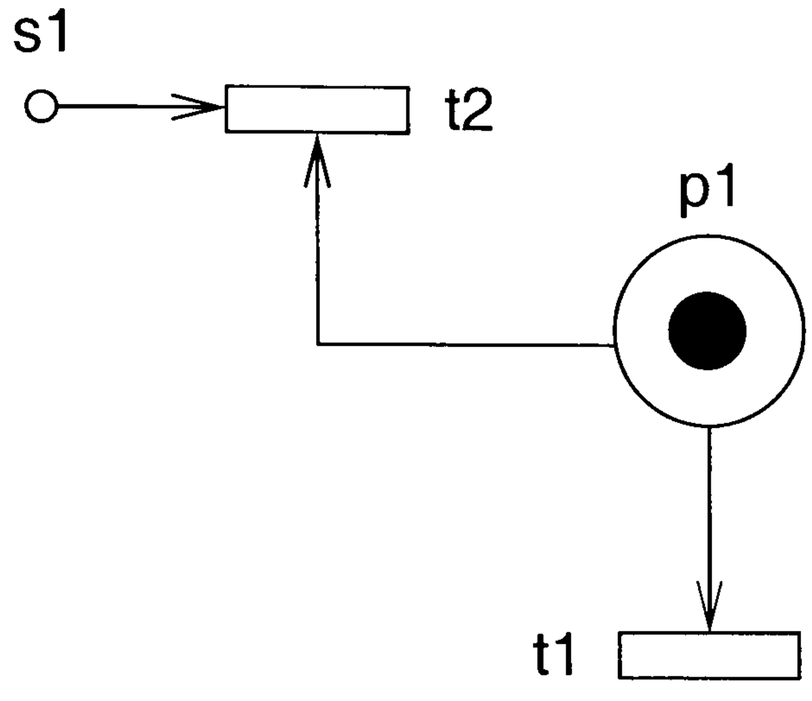

Figure 3.10: An example net to demonstrate the potential conflict of transition behaviour

The separation of transitions into two groups is required because an interaction transition must have firing priority over a non-interaction transition. The motivation for this can be illustrated with the example shown in figure 3.10. If place p1 is enabled then potentially either transition t1 can fire or, if s1 is also enabled, t2. If there is a non-deterministic firing priority of transitions, the possibility arises where t1 is given firing priority over t2 and, because t1 will always fire, t2 can never fire. The division of transition evaluation enables the evaluation of interaction transitions prior to evaluation of non-interaction transitions. Consequently when this semantic is applied to figure 3.10, transitions t2 will fire when place p1 is enabled and sensor s1 is enabled, otherwise transition t1 will fire.

As such, the execution of a Flownet can be considered as six sequential steps which are continually repeated:

- Step 1 - evaluate sensors: if the threshold condition of a sensor is met, then the sensor becomes true, if not the sensor becomes false.

- Step 2 - evaluate interaction transitions: an interaction transition becomes enabled when all places, sensors and boolean plugs targeting it by an arc are true, and all places targeting it by an inhibitor are false. An interaction transition then fires, upon which all the targeting places are disabled and the transition becomes true. Otherwise the transition becomes false.

- Step 3 - evaluate non-interaction transition: a transition becomes enabled when all places targeting it by an arc are true, and all places targeting it by an inhibitor are false. A transition then fires, upon which all the targeting places are disabled and the transition becomes true. Otherwise the transition becomes false.

- Step 4 - evaluate places: if there exists a transition targeting a place which is true, then the place is enabled.

- Step 5 - evaluate flow controls: if there exists a transition or place targeting a flow control which is true, then the flow control becomes true. If not, the flow control becomes false.

- Step 6 - evaluate transformer: if there exists a flow control targeting a transformer which is true, then the transformer becomes true, otherwise it becomes false.

This cycle of execution is illustrated in figure 3.11.

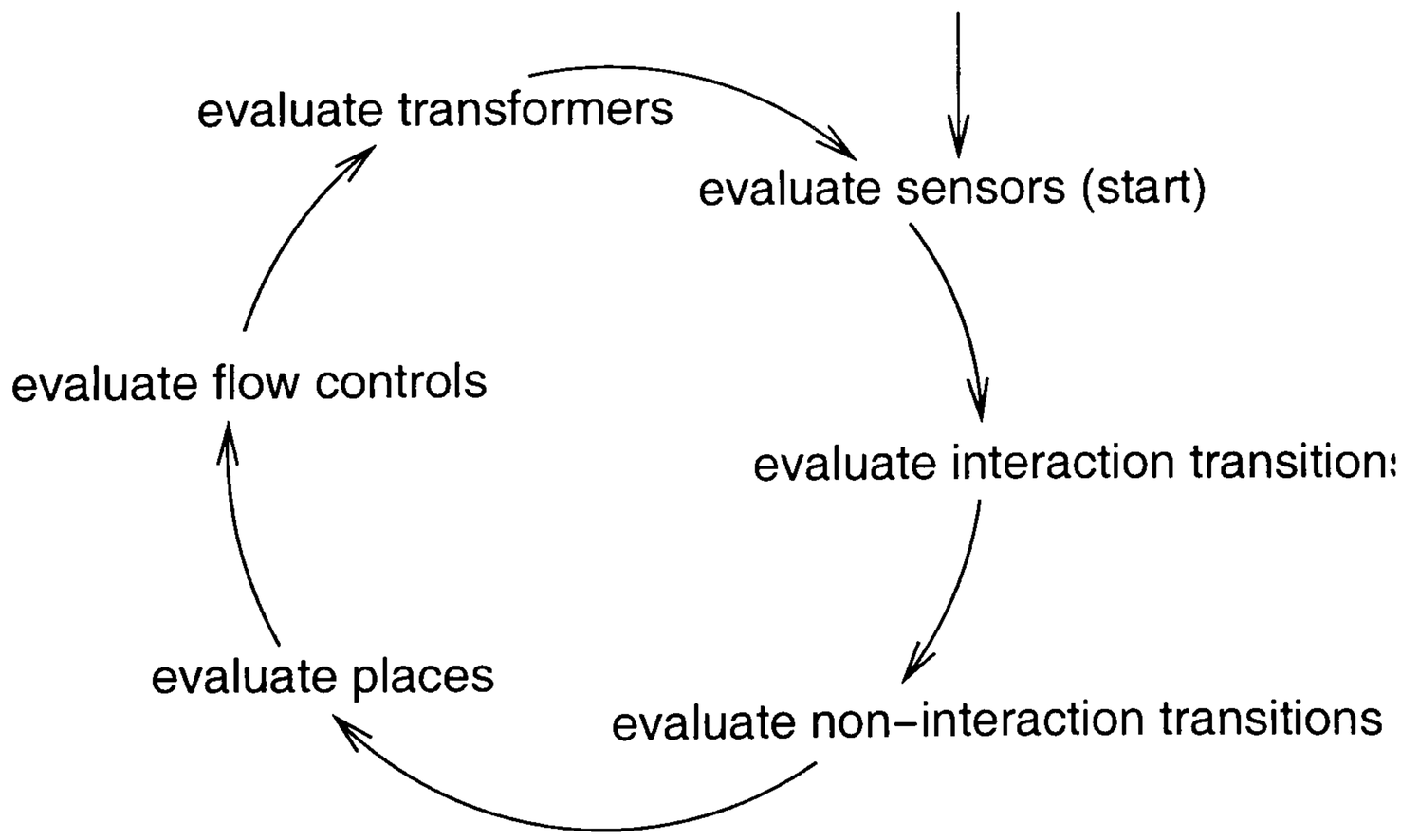

Figure 3.11: The execution cycle of a Flownet

3.4 Examples

In this section we exemplify Flownets. In section 3.4.1 we return to the mouse-based flying interaction technique used to explore the alternative formalisms in chapter 2. In section 3.4.2 we illustrate the use of the formalism for describing the behaviour of world objects using a door example.

3.4.1 Mouse based flying

As described in chapter 2, the mouse-based flying interaction technique enables flying through a virtual environment on the x and z axis using the desktop mouse. The technique is initiated by pressing the middle mouse button. When the mouse cursor is moved away from the clicked position, navigation through the environment begins. The user's speed and direction is directly proportional to the angle and distance between the current pointer position and the point at which the middle mouse button was pressed. Flying is deactivated by a second press of the middle mouse button.

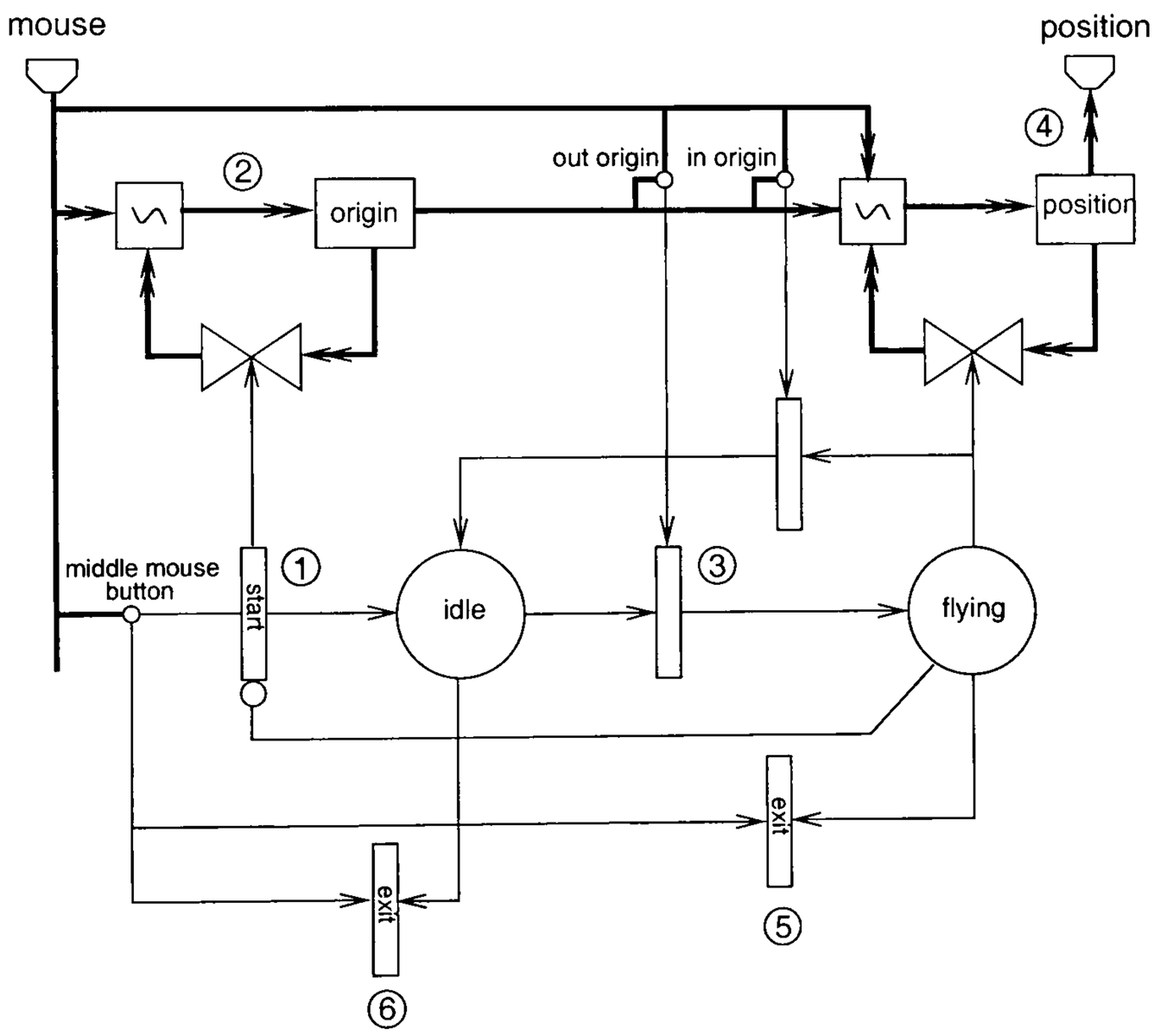

Figure 3.12: Flownet for the mouse-based flying interaction technique

The Flownet for this technique is shown in figure 3.12. This has one input: *mouse* and one output: *position*. When the middle mouse button is pressed, the Flownet *middle mouse button* sensor is activated and the *start* transition (1) is fired. The *start* transition enables the continuous flow which updates *origin* with the current mouse position (2). 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 triggered and the token is moved from the *idle* to the *flying* state. A token in the *flying* state enables the continuous flow which calculates the translation on *position* (4) using the current mouse position and the *origin*. This is then continuously supplied to the output plug. Whenever the *flying* state is enabled, the inhibitor determines that the *start* transition cannot be re-fired. When the *in origin* sensor detects that the mouse has moved back into the *origin* position, the token in the *flying* state is returned to the *idle* state closing the flow control and halting the transformation of *position*. Regardless of whether the technique is in the *idle* or *flying* state, it can be exited by the *middle mouse button* sensor becoming true and firing one of the exit transitions (5 or 6).

3.4.2 Door

In this section we demonstrate how Flownets can be used to specify world object behaviour by specifying a door. This door is initially in the closed state and begins to open when a device button is pressed. When the door is fully opened it can again be closed with a second press of the button.

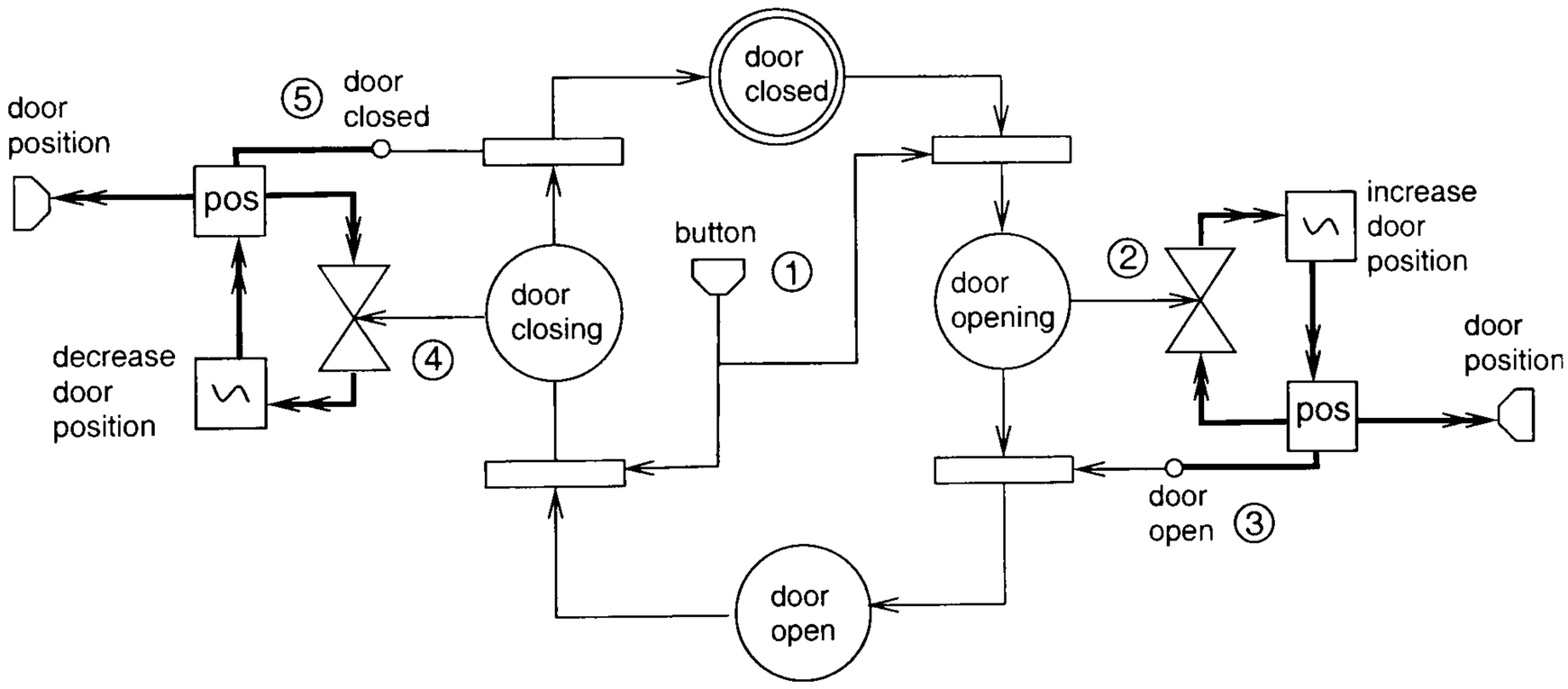

Figure 3.13: Flownet for the behaviour of a door world object

The Flownet for this behaviour is shown in figure 3.13. This has one input from the external environment: *button*, and one output to the external environment: *door position* (although this is duplicated for clarity of presentation). Initially the door is in the *door closed* state, a token is represented by the inner circle[1]. When a discrete button press is detected from the *button* plug (1), the associated transition moves the token from the *door closed* to the *door opening* state. This enables the flow control

[1]the token is represented in this way (rather than as a black dot) so that both the text and the token can be displayed within a place.

(2) which enables a continuous opening transformation of the *door position* which is output to the external environment through the *door position* plug. When the *door open* sensor (3) detects that the door is fully open, the token is then moved from the *door opening* state and placed in the door *door open* state. When the door is in the *door open* state, a second press of the button moves the token to the *door closing* state. This state enables the corresponding flow control which enables the transformation of the *door position* which is output to the external environment. When the *door closed* sensor detects that the door is fully closed, the token is moved from the *door closing* state back to the initial state of *door closed*.

3.5 Conclusion

In this chapter we have informally explored the various constructs within the Flownet formalism. A more thorough treatment of a semantics for Flownets is give in appendix A using the Z formalism.

Chapter 4

Prototyping Flownets

In this chapter we introduce the Marigold toolset. This toolset supports the translation from Flownet designs of virtual environment behaviour to an implementation prototype. Marigold is designed to be independent of any specific implementation. For the proof of concept described in this thesis we use the Maverik [Hubbold, Dongbo, and Gibson 1996] toolkit. Many toolkits would be suitable as an implementation target, however Maverik was particularly desirable for a number of reasons. Firstly, its open source nature allowed us to examine details of its implementation. Secondly, Maverik is widely used and has proved itself in small and large applications.

4.1 Introduction

In chapter 2 we argued that a prototyping approach for virtual environment behavioural specifications should hide low-level implementation concerns from the designer. This need to provide higher-level abstractions for building virtual environments has been addressed in the past in the approaches presented in [Sherman 1993; VPL Research 1991].

In these approaches the virtual environment is specified as a visual data flow network. The data flow networks are similar to those used in the hybrid formalisms discussed in chapter 2, but rather than specifying abstract concepts they are specifying implementation concepts. The nodes describe presentation components such as devices which are origins and targets of data, and functions which manipulate this data. The nodes are linked via transitions to define how the environment should behave. Consequently, the data flowing in through the input nodes (devices or world objects) is transformed by the behaviours and then output through the output nodes. A screenshot of "body electric" [VPL Research 1991] is shown in figure 4.1. In terms of behavioural design, these approaches suffer from the issue of trade off between flexibility and abstraction discussed in the introductory chapter since they are di-

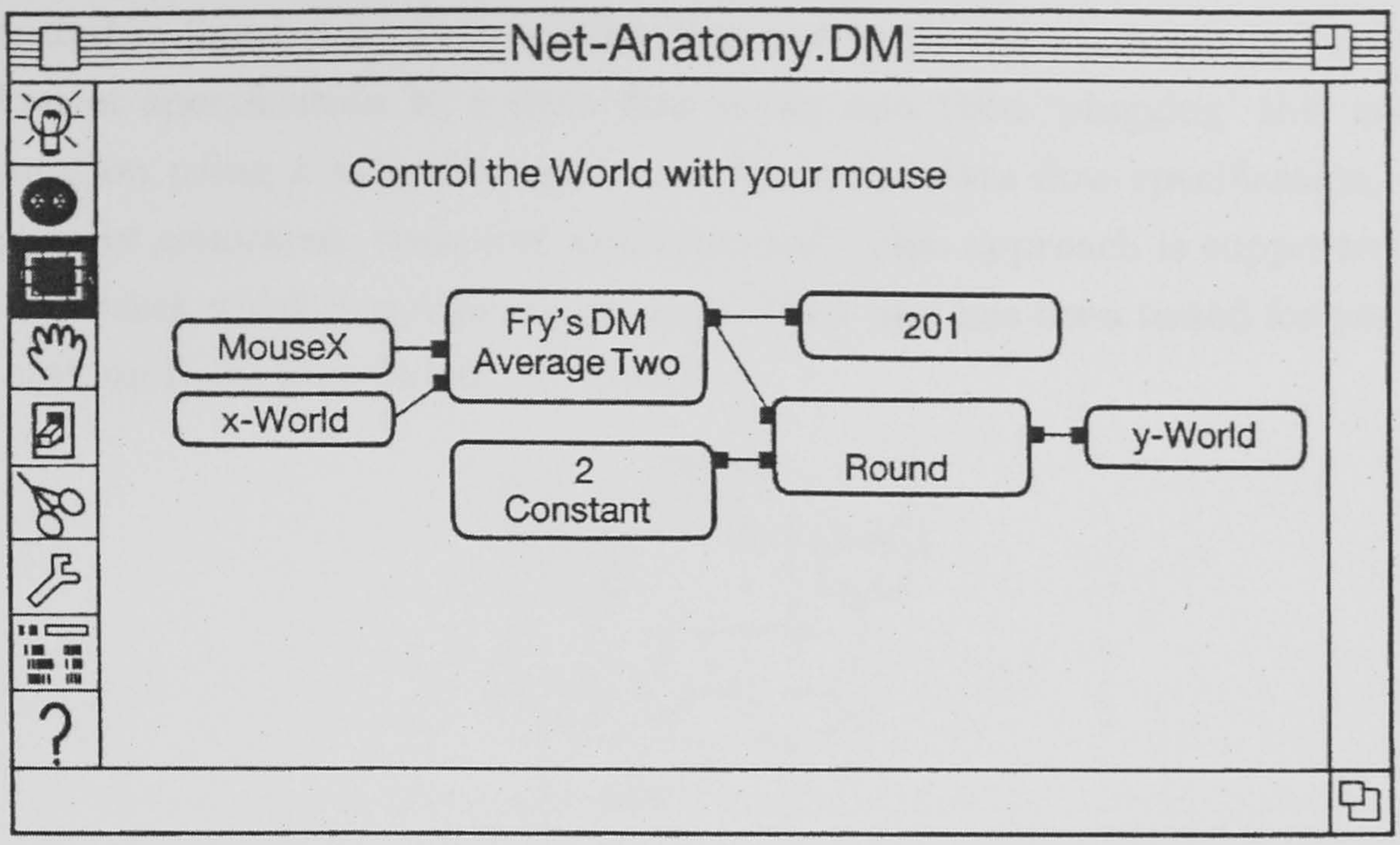

Figure 4.1: An example of a "body electric" data flow specification (taken from [Kalawsky 1993, p218])

rectly associated with an implementation. However, the method of introducing the presentation and relating this to the behaviour is advantageous for two reasons:

- the nodes encapsulate underlying implementation detail, therefore this detail is hidden from a designer.

- there is a loose coupling between the nodes that describe the presentation and the behaviour nodes. This allows presentation components to be substituted with ease by inserting new nodes and linking these to the behaviour using transitions.

This data flow style of specifying virtual environment implementation is motivated by earlier work exploring alternative methods for the specification of data visualisation algorithms. Of particular significance is the application visualisation system (AVS) [Upson, Jr., Kamins, Laidlaw, Schlegel, Vroom, Gurwitz, and van Dam 1989] where complex algorithms are encapsulated into nodes which are subsequently linked to input and output data sources. This approach has been extensively applied to the specification of traditional interfaces in order to yield the benefits outlined above (see for instance [Smith 1990; Ingalls, Wallance, Chow, Ludolph, and Doyle 1988; Esteban, Chatty, and Palanque 1995]). In the virtual environment context, this style of specification has also been used to specify behaviour while immersed in the virtual environment [Steed 1996].

The challenge addressed in this chapter is to provide a means of prototyping Flownets using data flow networks. An overview of the approach to achieving this

is illustrated in figure 4.2. This involves supporting the (semi-automatic) refinement of a Flownet specification to a data flow node, and then 'plugging' this node into a presentation using a data flow network. From the data flow specification, code is automatically generated, compiled and executed. This approach is supported by the Marigold toolset which is written using Java/AWT and has been tested for portability across unix and windows based platforms.

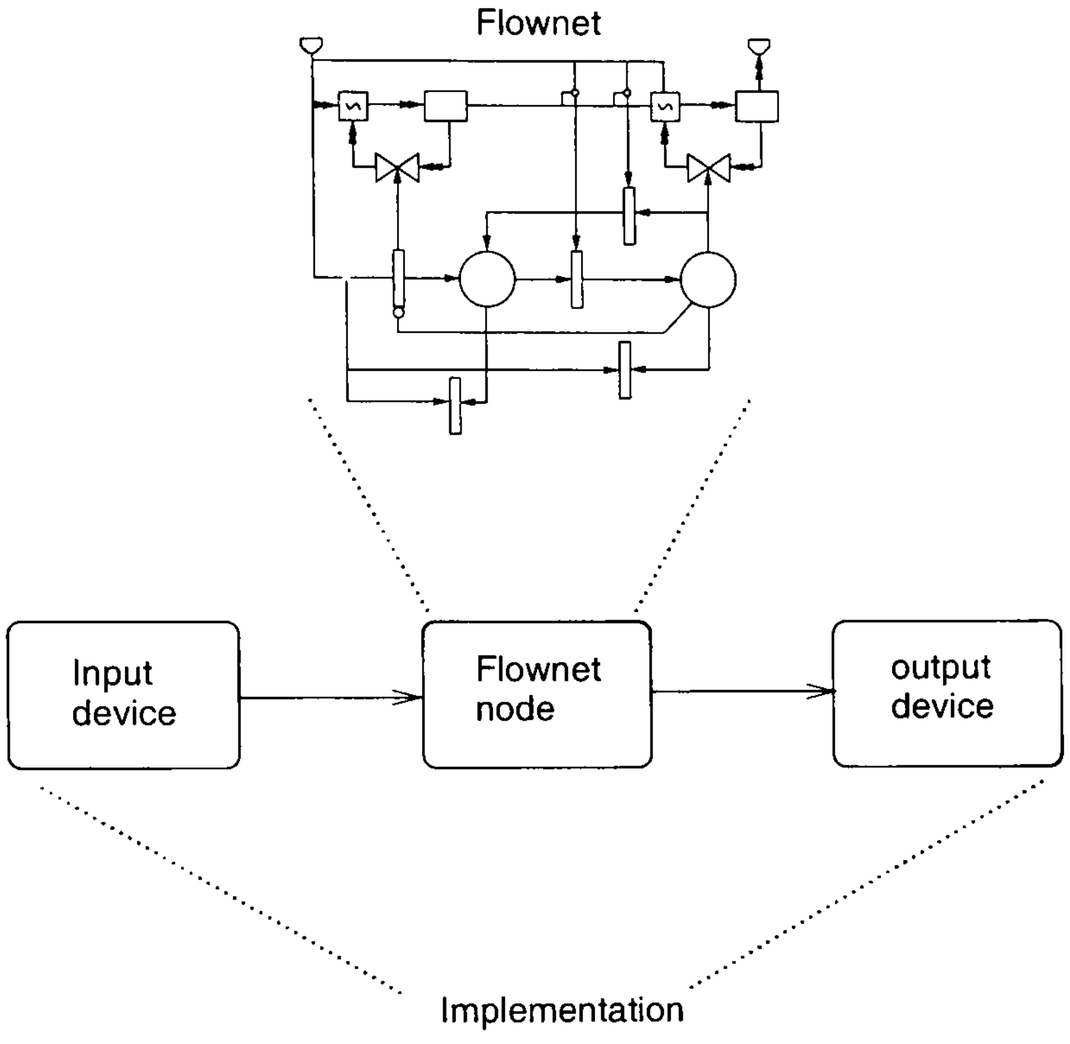

Figure 4.2: Combining the advantages of Flownets for behavioural design with data flow networks for prototyping

4.2 Prototyping interaction techniques

In this section we shall demonstrate how Marigold supports the transition from Flownet specifications of an interaction technique to a prototype.

4.2.1 Building the specification

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 provides a means of specifying Flownets visually. The resulting design for the mouse-based flying interaction technique is illustrated in figure 4.3. The toolbar at the top of the HSB contains an option for each of the node types (e.g. state, transition and transformer) and each of the connection types (continuous and discrete). The HSB enforces the syntax of the specification and only allows legal

connections between nodes. The tool also tries to maintain clarity of specification by automatically formatting the visual connections between components.

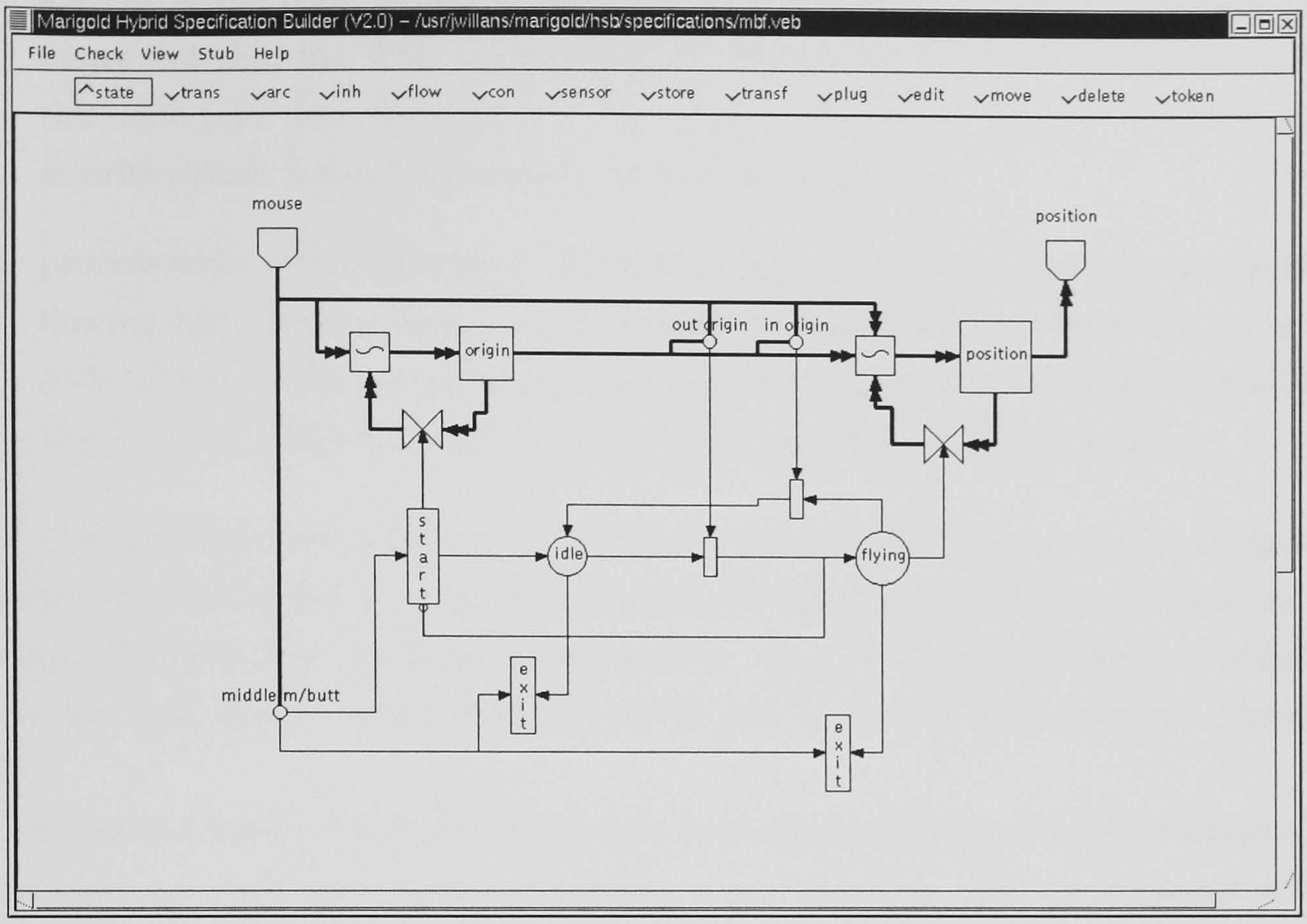

Figure 4.3: Flownet specification for the mouse-based flying interaction technique

Within the HSB it is necessary to add a small amount of code to some of the nodes of the specification. This code describes the semantics of those components more precisely. There are three types of code that can be added. We will describe these in the context of the mouse-based flying example:

1. 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. The code added to the mouse plug is illustrated in figure 4.4 (a). An integer variable represents the state of the mouse buttons and a vector represents the mouse position. For variables flowing from plugs, it is necessary to define whether its mapping to the external environment should be relative (environment = environment + plugVar) or absolute (environment = plugVar). Variable code is also used to define data which reside in the stores of the specification.

2. conditional code - this is placed in all sensors and some transitions. It describes the threshold state of the data for firing the component. Illustrated in figure 4.4 (b) is the code added to the middle m/butt sensor. As can be seen from figure 4.4 (b), the HSB informs the developer which data flow in and out of the node (the data they are able to access). The code specifies that when the middle mouse button is pressed, the sensor should fire.

3. process code - this is placed in all transformers and denotes how the information flowing into the transformer is transformed. Illustrated in figure 4.4 (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, a stub of the interaction technique can be generated. This is achieved by selecting a menu option and specifying a target filename in the resulting dialogue box. Flownet specifications, along with the information added to the nodes, can also be saved to a file which can be loaded back into the Marigold HSB.

The second stage of the specification to prototype refinement involves integrating the interaction technique specification into a presentation. This is achieved within the prototype builder (PB).

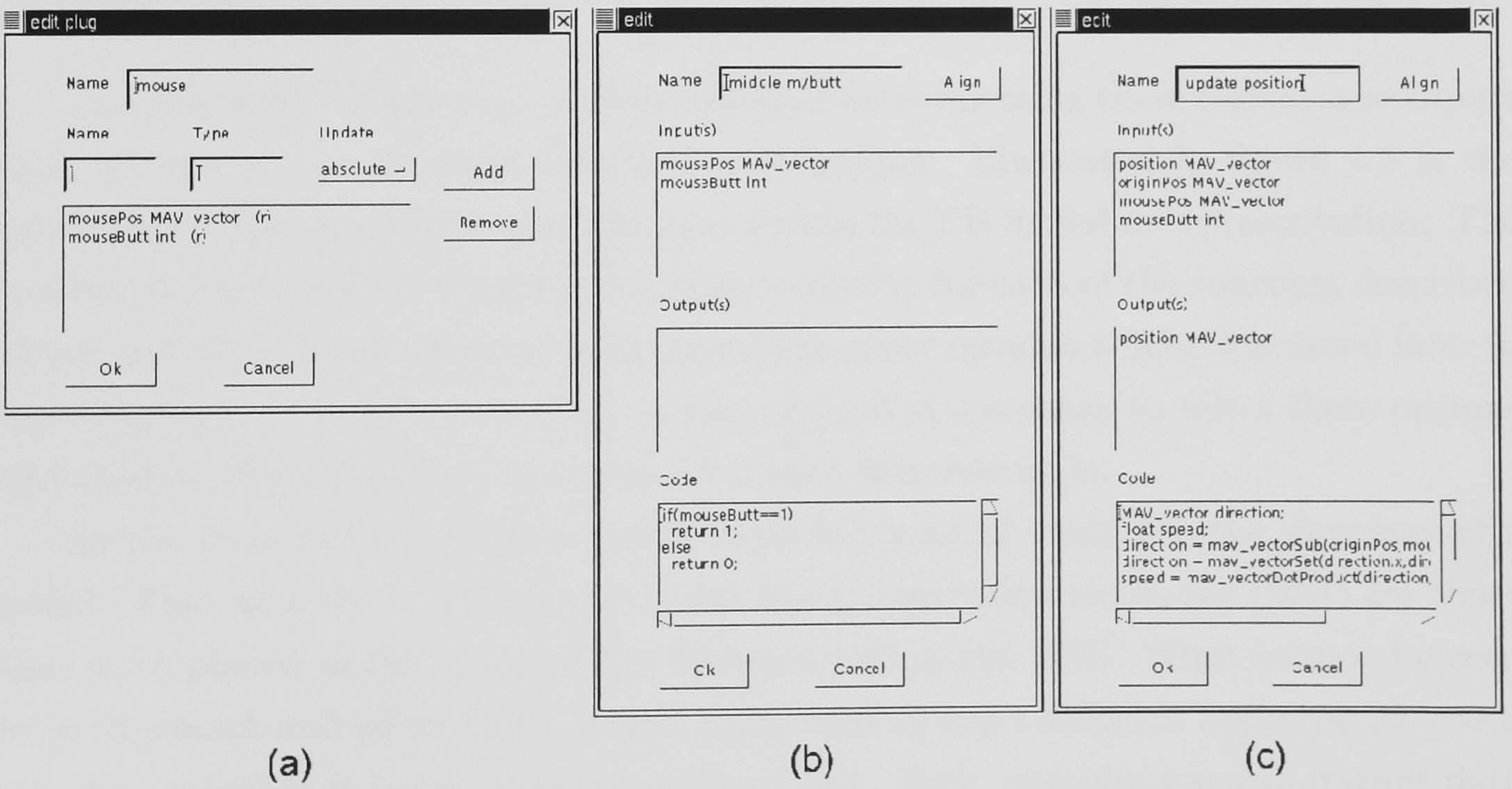

Figure 4.4: (a) Adding variables to the mouse input plug (b) Adding conditional code to the middle mouse button sensor (c) Adding process code to the position transformer

4.2.2 Constructing a prototype

The stub generated from the HSB is an environment-independent description of the behaviour. By this, we mean that it does not make commitments to the inputs and outputs from the behaviour. In order to explore the technique in an implementation prototype context, it is necessary to 'plug' it in to a presentation using data flow networks. This stage of the refinement is supported by the Marigold prototype builder (PB). The presentation is modelled as four concepts:

Definition 1 *Interaction devices (de) are physical devices which act as an input to interaction techniques.*

Definition 2 *Viewpoints (vp) visually render a subset of world objects.*

Definition 3 *World objects (wo) are visually perceived by the user in the virtual environment. World objects may also be used to represent the user, or part of the user, within the environment (often referred to as avatars).*

Definition 4 *Cursor objects are specific instances of the world objects that provide an indication to the user of the interaction device's state (e.g. positional feedback).*

The Marigold PB provides a visual method for connecting these elements as inputs and outputs to one or more interaction technique. Illustrated in figure 4.5 is the mouse-based flying interaction technique within the PB linked to a presentation. The toolbar at the top of the diagram contains an option for each of the concepts described above and also Flownet behavioural stubs (the other options will be discussed later in this chapter). In order to construct a prototype it is necessary to select these options and click on the workspace to create an object instance node.

As can be seen from figure 4.5 each node has a set of variables (the signature of a node). The variables for the mouse-based flying interaction technique (mbf) are those that were placed in the plugs of the Flownet within the HSB. What cannot be seen from this black and white figure is that each variable has a different background colour denoting whether it is an input, output or both. Each variable that can output data is annotated with a letter describing whether it provides an absolute (a) or relative (r) mapping. The relation between the nodes are defined by creating transitions between these variables which defines a data flow. The tool automatically verifies that the variables being joined are of the same type.

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.

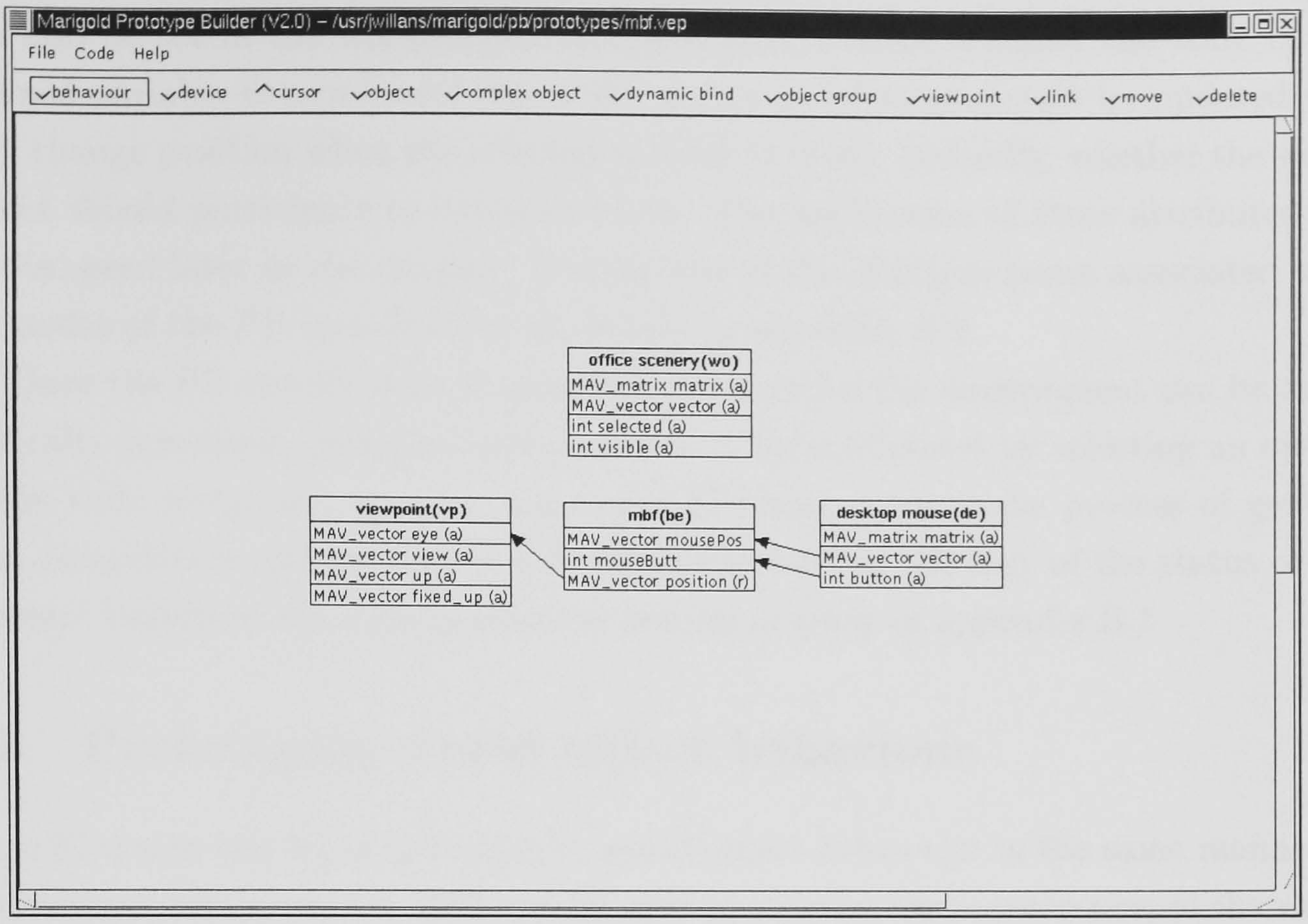

Figure 4.5: Prototype specification for the mouse-based flying interaction technique

Additionally, we have inserted an office desk world object so that the movement of the viewpoint can be perceived by the user. However because the desk remains static during interaction, it is not linked to any other environment concept. When a world object is inserted it is necessary to specify the file location of the world object created by a 3D modeller. In the case of an input device, the location of a device stub is required.

These device stubs are short textual files which can be constructed easily by the developer. They relate the output of each device to a common data layer (in a manner similar to that presented in [Faisstnauer, Schmalstieg, and Szalavári 1997]). The data layer is represented by the variables that appear in the node of the device stubs of the PB specification. This consistent interface allows devices to be easily substituted within interaction techniques. A number of stubs have been constructed for common devices, the process for adding additional device stubs is described in appendix B.3. In order to insert a Flownet behaviour into the PB specification, the file location of the stub generated from the HSB must be specified.

It is necessary to define the initial positional state of the viewpoint and world object concepts via a dialogue box. This is accessed by editing the respective node. For world objects, there are also options in this dialogue to select whether the object should be initially visible and/or initially selected. A number of further attributes

can also be set in the world object dialogue box. Firstly, whether the state of the selected variable determines if the world object positioning should be updated (i.e. only change position when the selected variable is true). Secondly, whether the world object should participate in dynamic binds. The application of these attributes will be discussed later in the chapter. Screenshots of the dialogue boxes associated with the nodes of the PB specification are shown in appendix B.2.

Once the PB specification is complete the code for the environment can be automatically generated, compiled and executed. This is achieved by selecting an option in the code menu and specifying a target filename. During the process of generation, compilation and execution, a dialogue box informs the user of the status of the process. Details of the code generation process is given in appendix B.1

4.3 Prototyping world object behaviour

Since Flownets can be used to specify world object behaviour in the same manner as interaction techniques, the HSB can be used to support the construction of the world object behavioural specifications. Similarly, the PB can be used to integrate the stub of a virtual environment world object behaviour (generated from the HSB) into a presentation. For world object behaviour, this integration would be a definition of the relation between the behaviour and the world object.

When world objects are exported from a 3D modeller, they must be decomposed (separated) according to how they are required to behave in an environment. For instance, a freezer unit may have a door which can open and close, and a number of drawers which can open and close. Therefore, the freezer's drawers, door and the freezer unit itself all become separate world objects. However, the behaviour would usually be described using one design. This need to link many world objects to a single behaviour can result in a complicated PB specification. Moreover, every time a world object with a behaviour is required within the PB specification, it must be relinked to the behaviour. In practice, it would be advantageous to be able to reuse this relation.

In order to address this problem, the Marigold toolset consists of a third tool called the complex object builder (COB). Rather than defining the relation between the stub of a world object's behaviour and the world object directly using the PB, the COB is used to perform this same task. Any additional information required from the external environment is also made explicit using this tool. From the COB a stub is generated which encapsulates all the details of how the world object will behave and appear. This stub is a reusable complex world object node that can be used in

any PB specification. To exemplify this process we will use an elaboration of the door world object described in chapter 3.

4.3.1 Building the specification

Illustrated in figure 4.6 is the discrete part of the Flownet specification for the behaviour of the door within the HSB. Shown in figure 4.7 is the continuous part of the specification also within the HSB. Although the HSB supports the construction of Flownet specifications using one view, it is preferable to split the continuous and discrete part of a larger specifications to maintain clarity of presentation. These views can be rapidly switched via a menu option. This Flownet specifies that the door can be opened, closed, locked and unlocked. It also incorporates constraints specifying that the door cannot be opened when locked and cannot be completely closed when locked. Code is added to some of the nodes of the specification and a stub of the behaviour is generated (in the same manner described in the previous section for interaction techniques).

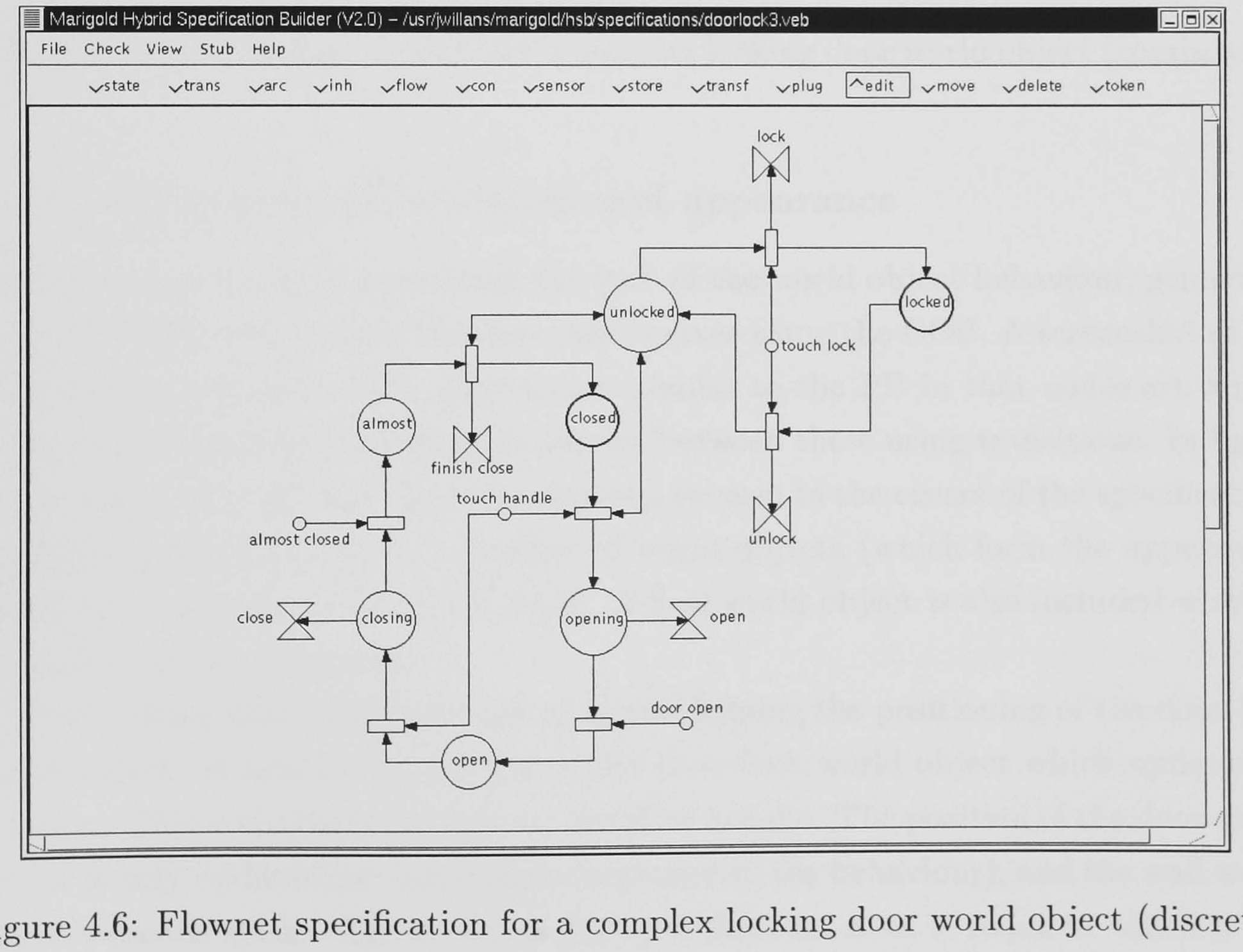

Figure 4.6: Flownet specification for a complex locking door world object (discrete)

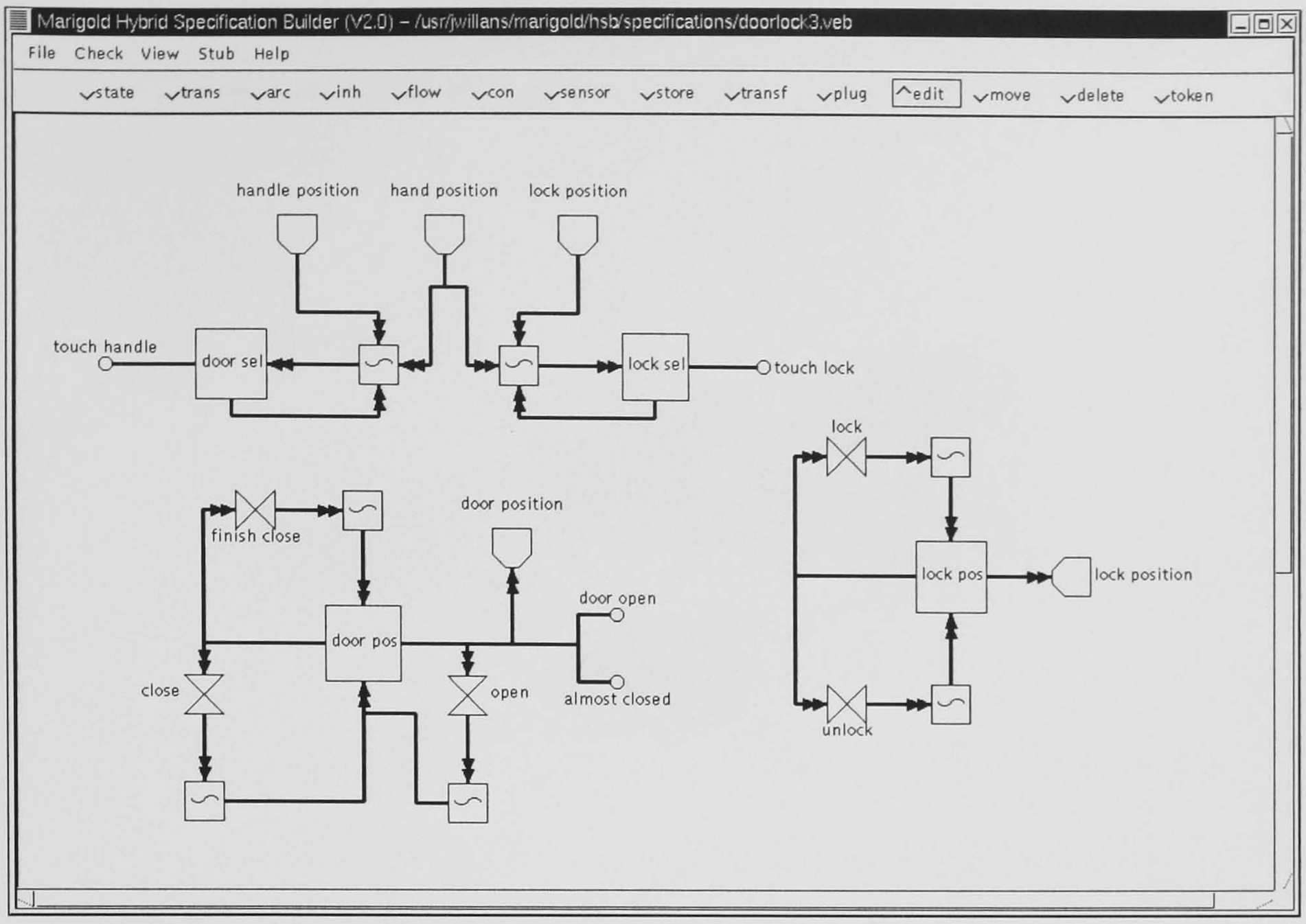

Figure 4.7: Flownet specification for a complex locking door world object (continuous)

4.3.2 Integration of behaviour and appearance

The next stage involves integrating the stub of the world object behaviour, generated from the HSB, with the world objects themselves using the COB. A screenshot of the COB is shown in figure 4.8. This tool is similar to the PB in that nodes are added to the specification and relations expressed between these using transitions. In figure 4.8 the stub of the locking door Flownet can be seen in the centre of the specification. We have linked this stub to a number of world objects (which form the appearance of the door), and an external link node. A wall world object is also included which is not linked to any behaviour.

The locking door behaviour has an input defining the positioning of the door lock world object. It also has an output to the door lock world object which updates its position. This is similarly the case for the door handle. The position of the door world object is only updated (no information is passed *to* the behaviour), and the wall world object remains static. The external link specifies that data is required which is not contained within the specification. In this case the position of the virtual hand world object (or whatever causes a selection) must be linked at the PB level of refinement. The relative positioning of the world objects are also set within the COB by editing their properties (see appendix B.2). From the COB, a stub of the specification is generated which encapsulates a description of how a world object should behave and appear.

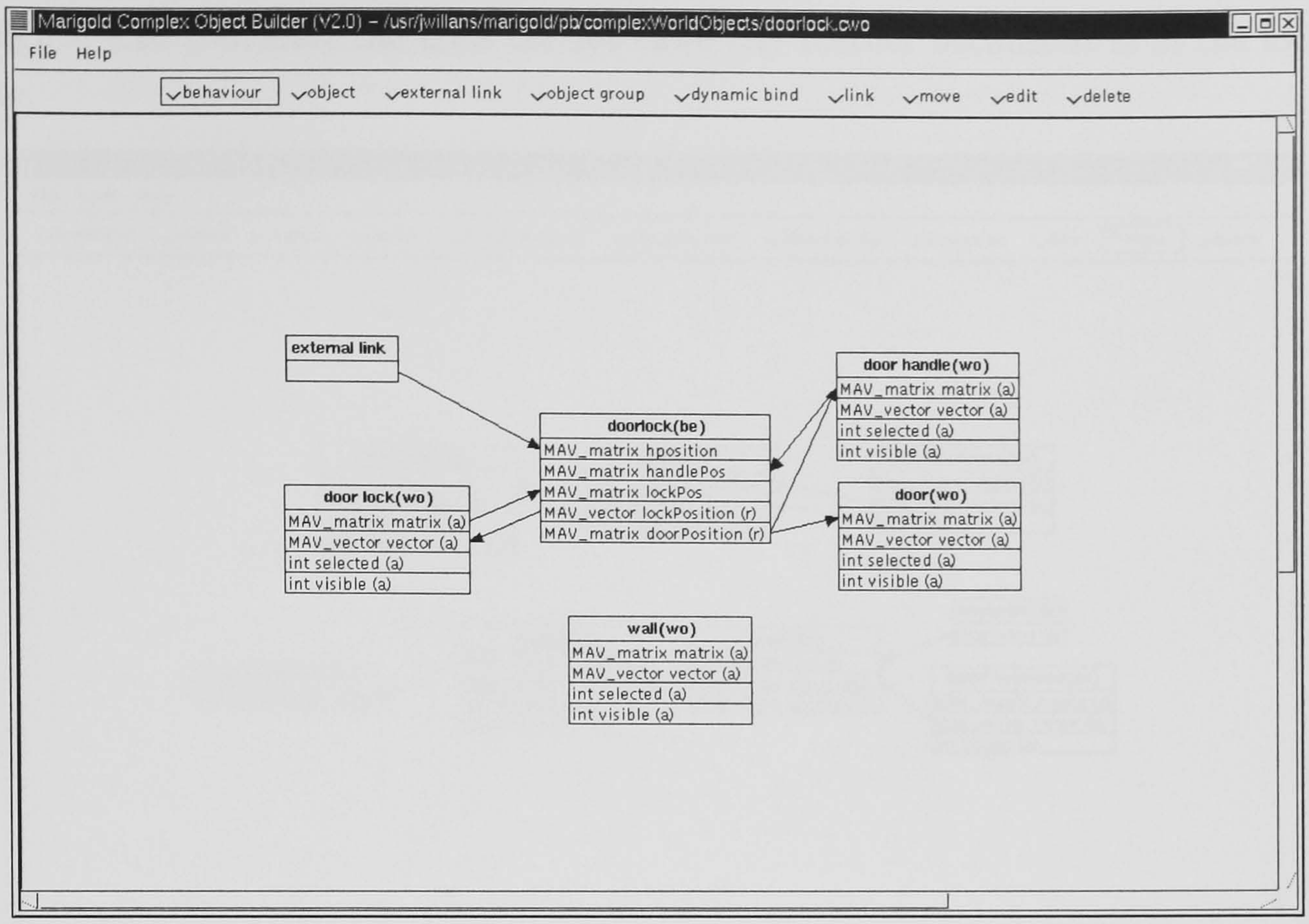

Figure 4.8: Complex object specification for a locking door world object

4.3.3 Constructing a prototype

A COB stub can be imported into the PB. Figure 4.9 shows the integration of the locking door complex object (co) into the mouse-based flying specification described earlier. In addition, we have included a simple manipulation interaction technique (sman, also described using a Flownet) within this specification which controls the position of a virtual hand world object using two mappings of the keyboard device. The locking door complex object node (door with lock) can be seen on the left side of the specification. The one variable exposed within this node is that specified as an external link within the COB specification in figure 4.8. This variable is linked to the position of the virtual hand. It is necessary to set the initial state of the complex object stub by editing its properties. The position part of this state is offset against the underlying relative position of the world objects defined in the COB. As in the previous example, code can be automatically generated, compiled and run from this PB specification.

Screenshots of the prototype generated from this PB specification (figure 4.9) are illustrated in figure 4.10. In addition to being able to navigate the environment using mouse-based flying, the virtual hand world object can be used to interact with the door and its lock using mappings of the keyboard. In figure 4.10 (top left) the door is closed and locked. In figure 4.10 (top right) the door is unlocked and fully open.

In figure 4.10 (bottom) the door cannot close any further because it is in the locked state.

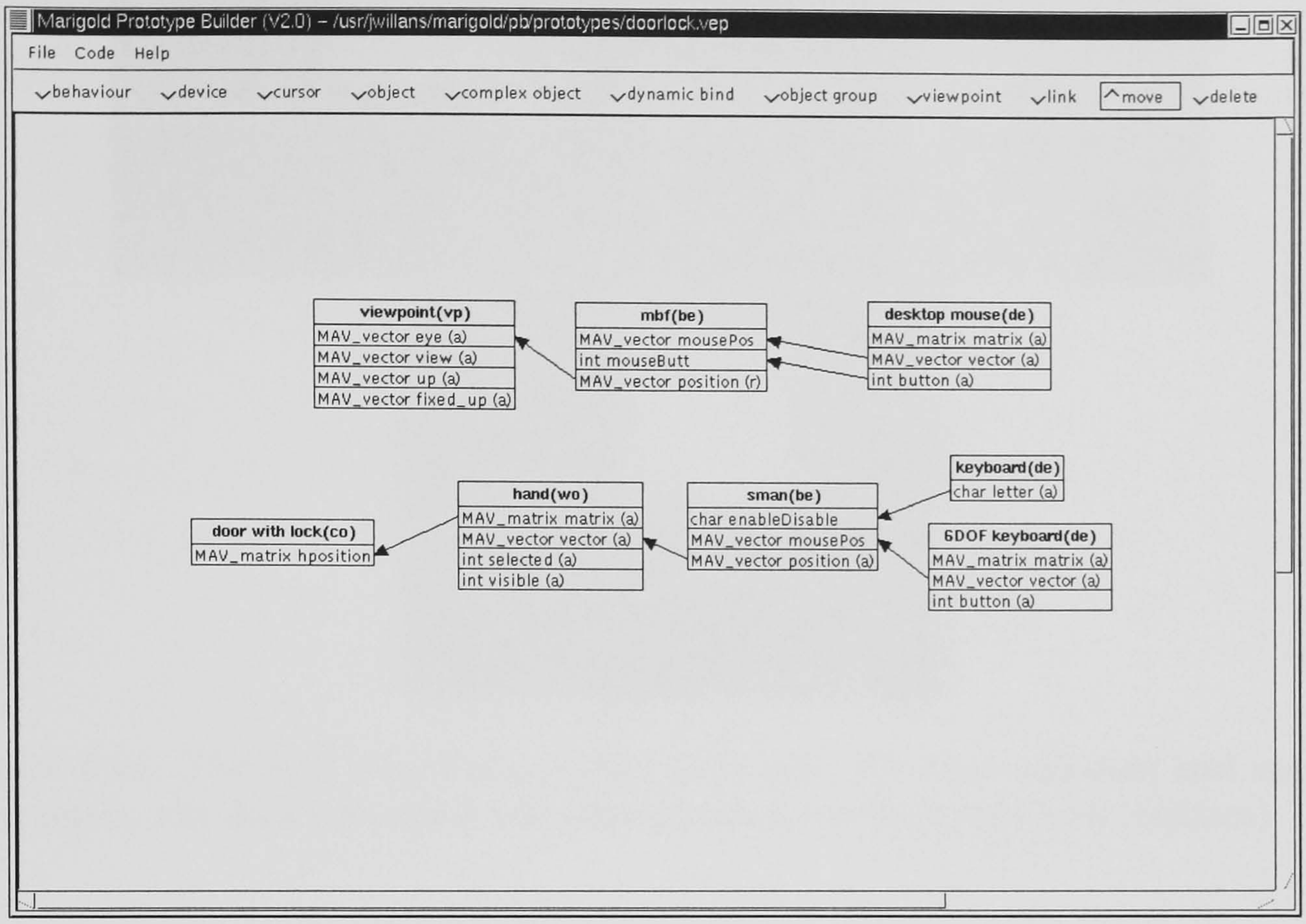

Figure 4.9: Mouse based flying prototype specification expanded to include a simple manipulation interaction technique and a locking door world object

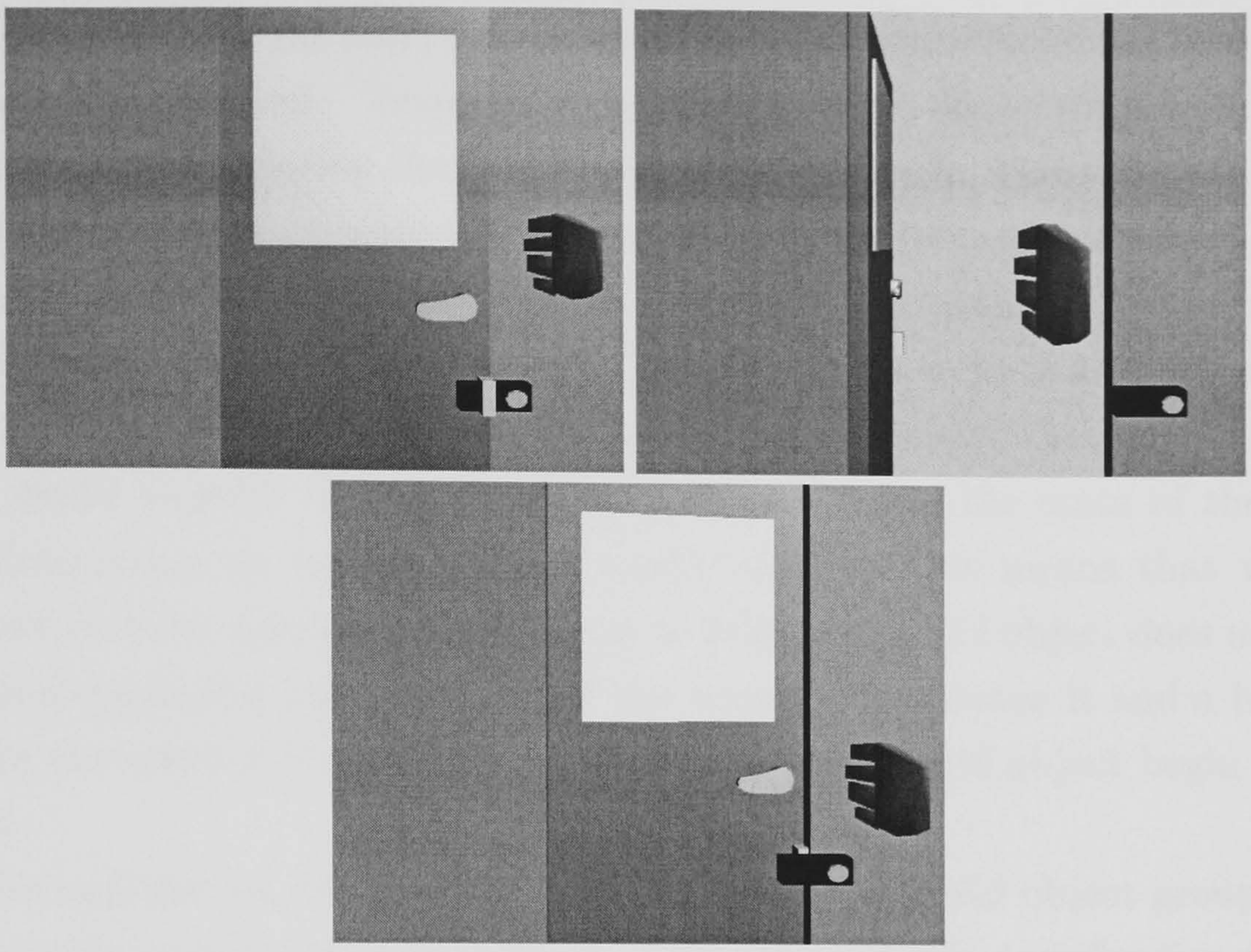

Figure 4.10: The door closed and locked (top left), the door unlocked and opened (top right), the door prevented completely closing by the locked lock (bottom)

4.4 Non-static binding

Flownet specifications are not concerned with the presentation external to the behaviour, they abstract from this by interfacing to plugs. The PB and the COB tools provide a means of binding a presentation to the plugs of Flownets using data flow networks. The binding style described in the previous sections is based on the common data flow approach where static links are created between nodes. This means that the relation that exists between the behaviour and the presentation must be explicitly specified and that this relation persists at runtime. However, often it is desirable that the relation between a behaviour and the presentation is expressed in more dynamic terms. For instance, 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. This type of non-static binding is supported by two additional constructs within the Marigold PB and COB, namely world object group and dynamic binds. In this section, we describe these constructs.

4.4.1 World object grouping

A world object group node is placed around a number of world object nodes within a PB specification. Although, a world object group node does not display variables in the same manner as other nodes (these were omitted for conciseness of specification),

there are slots on the right hand side of the node which correspond to those variables within a world object node. Transitions can originate and target these slots. A world object group is encapsulating the world objects into an array. Transitions to and from the world object group are mapped to each of the grouped world objects.

In itself a world object group is still a static binding mechanism, however it can be used in conjunction with a world object's selected variable to form a type of non-static bind. In section 4.2 we mentioned that one of the properties that can be set when editing a world object's dialogue box determines whether the state of the selected variable determines an update of that world object. This means that when this option is set and the selected variable is set to false, the world object does not change position and remains static regardless of the transitions between it and a behaviour. Only when the selected variable becomes true does the world object begin to accept behaviour.

To illustrate the use of this in combination with the world object group consider the PB specification illustrated in figure 4.11. Within this specification the world object group encapsulates the world objects ball one and ball two. These objects are set to only update when their selected variable is true. In this example, we have linked the world object group to a selection interaction technique (select, also described using a Flownet) which uses positional information from the world object group, to determine when the selector world object intersects with one of these. When this is the case, this interaction technique sets the selected variable of the appropriate world object to true. Since the position of the selector is linked to the world object group, all world objects within this group which have their selected variable set to true will have their position set according to that of the selector world object which is controlled by a simple manipulation interaction technique (sman).

4.4.2 Dynamic binding

The form of non-static binding described in the previous section depends on a behaviour (Flownet) determining an appropriate context for the binding to take place (in the previous instance, that the selector intersects with one of the world objects). Often, with virtual environments the binding context is that a world object occupies a certain space. For instance, that world objects should bind to the opening and closing behaviour of a drawer world object when they are placed within the drawer space. The dynamic bind construct allows such conditions to be described.

A dynamic bind node is added to a PB specification in the usual manner, whereupon Marigold requests the name of a world object file created using a 3D modeller. The position of this world object is set by editing its properties in the same way as a regular world object and transitions are used to link behaviours to the dynamic bind

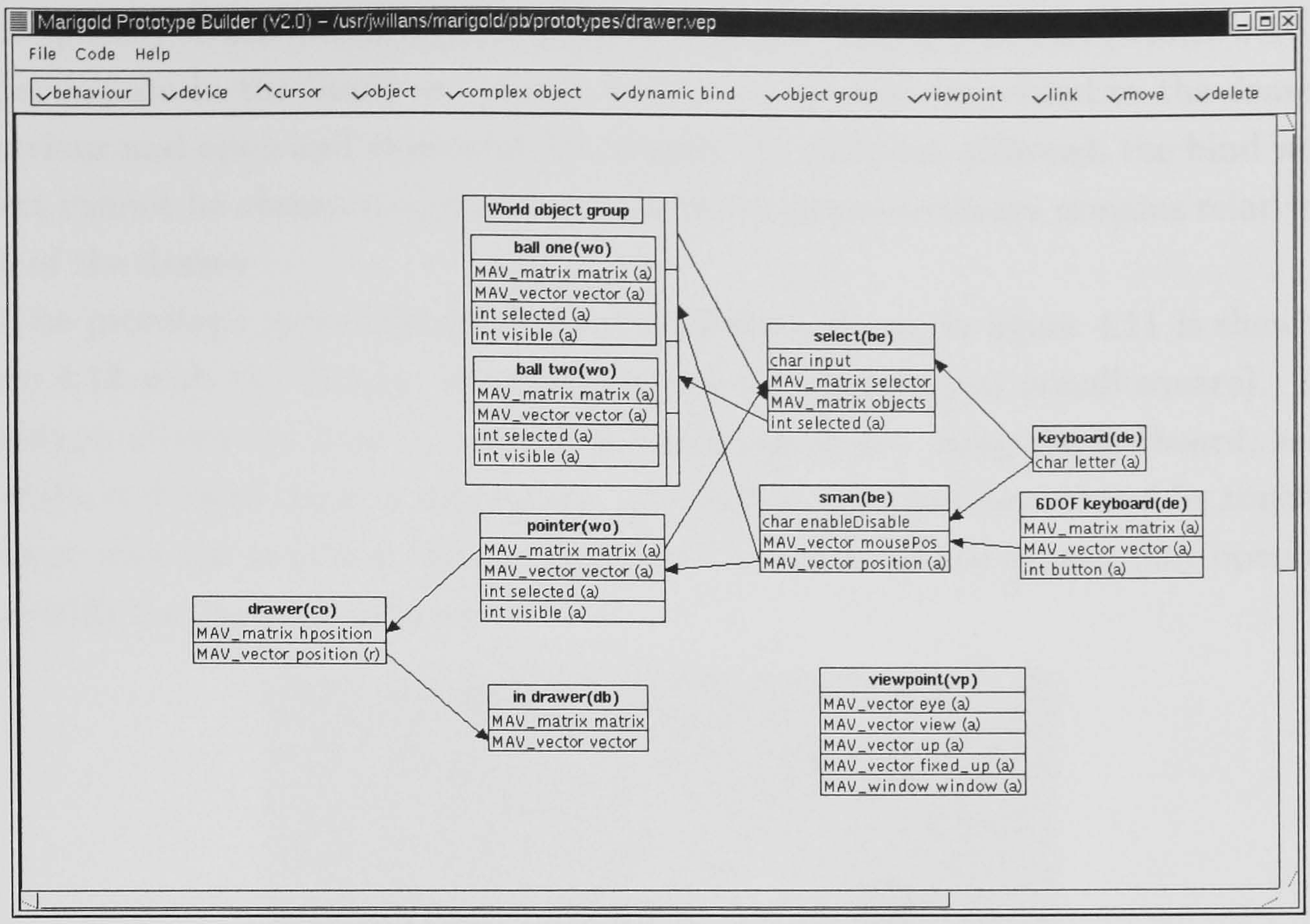

Figure 4.11: A specification illustrating the two forms of non-static binding constructs supported by Marigold

node. The world object associated with a dynamic bind represents a space within the virtual environment which a regular world object must occupy in order to bind to any behaviours linked to the dynamic bind node in the PB specification. The type of occupation a world object must make in order to satisfy the bind is also set in the properties of the dynamic bind node, this can either be a partial or full intersection with the world object which represents the binding space. It is also necessary to specify within the properties of a dynamic bind node whether the bind itself should also be updated by any associated behaviour. The dialogue box to these properties is described in appendix B.2. In order to participate in a dynamic bind, a world object's dynamic bind attribute must be set to true by editing its properties (as discussed in section 4.2).

In figure 4.11 a dynamic bind construct (db) has been added to the PB specification (in drawer) and linked to the position of the drawer world object. For this dynamic bind, we constructed a world object to represent the space inside the drawer (not displayed in the environment) and set its positioning attributes such that it was initially located inside the drawer world object. In this example, we specified that an object should be fully within the space defined by the drawer world object, and that the binds world object should behave according to the behaviour of the drawer.

Consequently, when world objects, such as ball one and/or ball two (which were set to participate in the bind) are placed within the drawer, they bind to the drawer's behaviour and open and close with the drawer. In addition, although the bind world object cannot be observed in the environment, its position always remains relative to that of the drawer.

The prototype generated from the specification shown in figure 4.11 is shown in figure 4.12 with the drawer, two balls and the selection object (small square). This prototype allows the user to control the selection object using the keyboard, select the balls and place them in the drawer. The drawer is opened and closed by touching its front with the selection object. When the ball(s) are in the drawer they open and close with the drawer's behaviour.

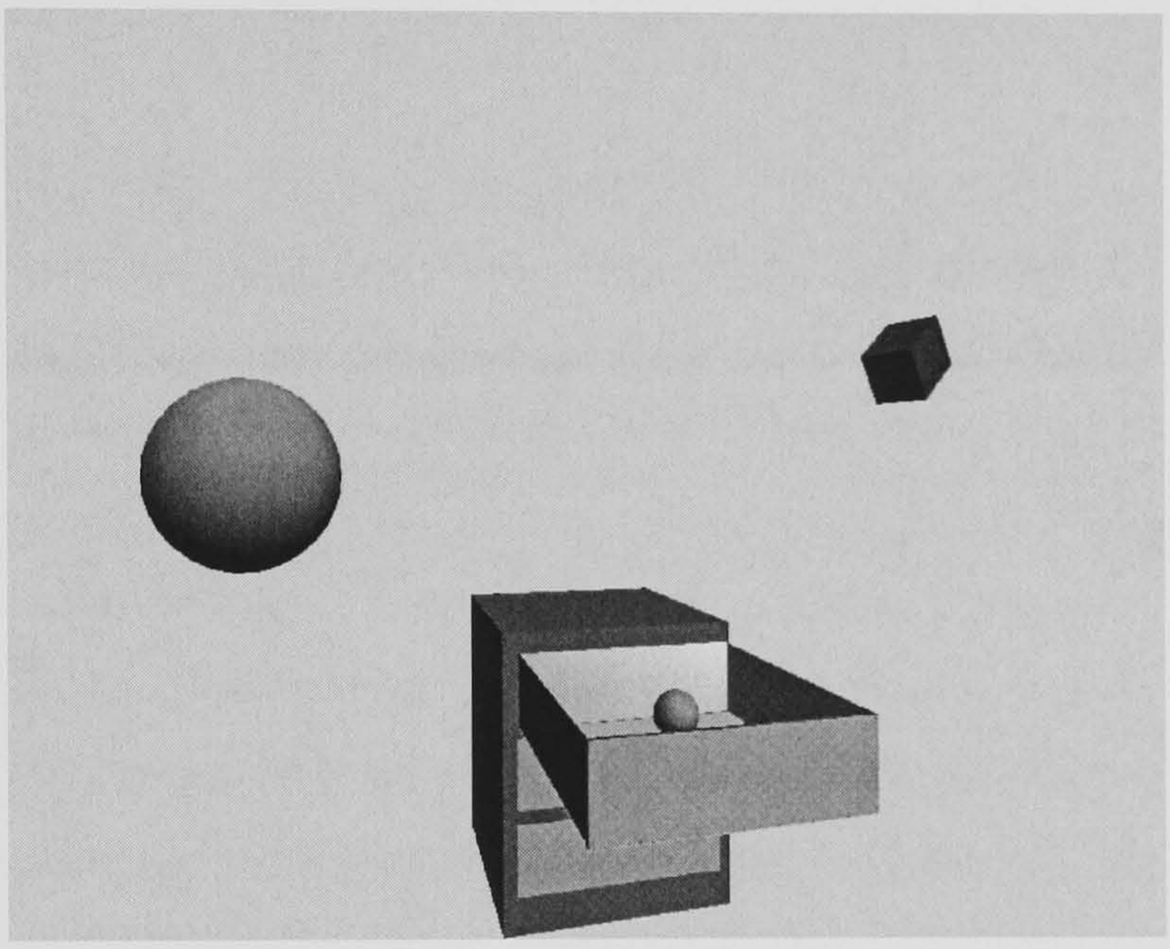

Figure 4.12: Screenshot of the drawer world object

4.5 Conclusion

In this chapter we have presented an approach for the prototyping of behavioural designs specified using the Flownet formalism. This approach is supported by the Marigold toolset which provides a transition from Flownets to an implementation using Maverik/C. The transition is achieved by 'plugging' the Flownet designs into a presentation using data flow networks. Although traditional data flow networks were found to work well for the basic aspects of specifying a virtual environment prototype, they were found to be limited with respect to two areas.

The first limitation concerns the definition of the relation between world object behaviour and the concepts that form the world object's appearance. These relations must be redefined every time it is required, this hinders reuse. Additionally, the

complexity of these relations can compromise the clarity of a PB specification. In order to address this, the COB tool was added to the Marigold toolset which allows the relation to be defined independent of a PB specification, in a manner that can be reused in different contexts.

The second limitation concerns the style of binding between the behaviour and environment concepts. With traditional data flow networks this is static and persists at runtime. Often it is the case that dynamic relations need to be expressed. This is addressed by the use of non-static binding mechanisms within the data flow specification. These support a method of specifying dynamic relations that can take place between behaviours and world objects at runtime.

Chapter 5

Analysing Flownets

In chapter 4 we described how the Marigold toolset supports a transition from Flownet specifications of virtual environment behaviour to a prototype. This allows characteristics of the behaviour to be explored by the user in an implementation context. In this chapter we explore the extent to which this can be complemented by the (automatic) analysis of the characteristics of Flownet designs.

5.1 Introduction

Prototyping is a powerful approach to evaluating a design because it involves the user [Myers 1989]. However there are features of a design that cannot be guaranteed to be demonstrated using a prototype. Firstly, evaluating a design using a prototype is informal and consequently imprecise. Secondly, with a prototype it is not possible to be certain that characteristics of a design have been analysed exhaustively. For these reasons, a flawed design can be understood to be correct using prototyping in isolation.

A complementary technique is the formal analysis of design specifications. Formal analysis can address the deficiencies of prototyping because it is precise and exhaustive [Campos 2000]. A prerequisite of the application of such analysis is that the design specification that is to be evaluated is also formal (can be described mathematically). The process of formal analysis involves formulating the characteristic of the design that requires evaluating as a precise property. This property is then applied to the formal design specification and a result is determined which specifies whether the property holds.

There are two approaches to determining the analysis result. The first of these is manual proof. The disadvantage of this approach is that a manual process is error prone and is hard to do by designers. The second approach involves automating the analysis. With this approach there can be more confidence about the result

since automation implies repeated accuracy. An additional advantage of automated analysis is that, in most cases, a result can be derived in less time and with less ingenuity than a manual proof.

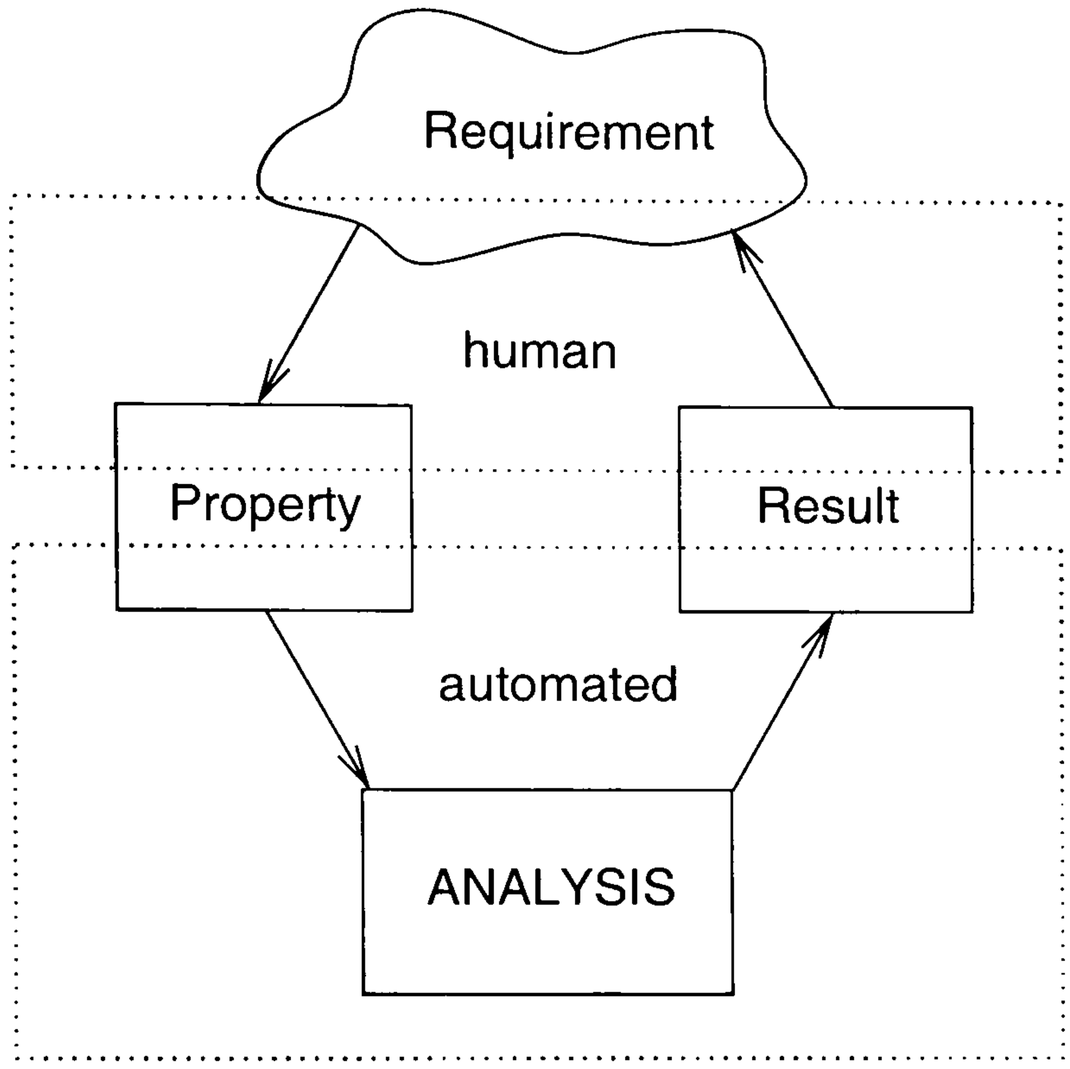

Figure 5.1: An overview of the analysis process

Illustrated in figure 5.1 is an overview of the process of analysing design specifications. From this it can be seen how automation can do much of the work in the analysis process. However as can also be seen from figure 5.1, it is still necessary for a human to translate the requirements into properties. If this is performed incorrectly then the results of the analysis are meaningless. Therefore, it is important that there is a close correspondence between the language of the requirements and the language of the properties. Similarly, the result of the analysis must still be interpreted by the human. If the result of this is that a property holds, then no further clarification is necessary. However, if the property does not hold, then the design must be revised. Therefore, the result of a failed analysis should express with as much clarity as possible under what circumstances the property fails.

This discussion has motivated a criterion for analysing properties of design specifications:

- The use of automation can help assure the accuracy of the analysis.
- Properties must be specified in a language that has a close mapping to the requirements.
- Meaningful results should be given when a property fails to hold.

Using the above criteria, this chapter explores the extent to which the analysis of design specifications can be applied in the context of Flownet designs and virtual environment properties.

5.2 Properties

As expressed in the previous section, properties play a central role in the formal analysis of design specifications. In this section we discuss the types of properties that are relevant to the analysis of virtual environment behavioural specifications constructed using Flownets.

One way of thinking about an interactive system is that it consists of two parts. The first of these is the core system which realises the functional requirements of the system (the reason it is built). The second of these is the interface between the user and the core system whose job it is to support the user in interacting with the system. With the core system the concern is its correctness according to the functional requirements. With the interface the primary concern relates to ensuring that it is usable. Since these two parts are concerned with different requirements, they can be designed and evaluated separately[1].

For virtual environments, the distinction between the core system and the interface is less clear. This is because the interface is not only supporting the interaction of the user, but also the functional requirements of the system. That is, there is no real system core beyond the interface, since the interface is the reason the system was built. The consequence of this is that requirements pertaining to both correctness and usability can be considered in the context of design specifications of virtual environment interfaces. In the following sections we discuss the properties that are relevant to virtual environments in these two areas. The final section discusses conflicts that may occur when dealing with both types of properties in the same design.

5.2.1 Correctness

In the context of safety critical systems, analysis of correctness is an important part of ensuring the system behaves in a manner consistent with their requirements. Correctness properties are usually classed into one of two groups: safety and liveness[2]. Safety properties specify that some undesirable behaviour cannot take place in the system. For instance, in the case of a nuclear reactor, a safety property might specify that it cannot be in the state of meltdown. Liveness properties specify that desirable

[1] A remaining issue is ensuring an accurate relation between the interface specification and that of the core system. This issue is explored in [Doherty, Campos, and Harrison 2000]

[2] Although there has been a suggestion that this categorisation is insufficient [Naumovich and Clarke 2000], it is rich enough for the purpose of our discussion.

behaviour can eventually take place. For instance, in the case of a nuclear reactor, a liveness property might be that the rods (the positions of which control the reaction) can be raised.

Virtual environments are often concerned with simulating the real world at different levels of realism. In the same way as safety critical systems, there is behaviour that we want to ensure takes place (liveness properties) and behaviour that we want ensure never takes place (safety properties) within the simulation. For instance consider the behaviour of the locking door discussed in chapter 4. Physical constraints on a door prevent it from being in certain states. These can be expressed as safety properties:

- The door *cannot* be open and closed simultaneously.
- The door *cannot* be locked and unlocked simultaneously.

Undesirable sequences of states can also be derived and expressed as safety properties:

- A closed and locked door *cannot* be *immediately* opened.
- A closed and locked door *cannot* be *immediately* closed.

Similarly, there are certain sequences of behaviour that the door can exhibit. These can be expressed as liveness properties:

- The door *can* be unlocked and *then* opened.
- The door *can* be closed and *then* locked.

All the properties discussed in this section relate to the discrete behaviour of the design specification for a virtual environment. There are also properties which can be formed about the continuous nature of the behaviour, for instance: the opening speed of the door should accurately simulate that of a real door. Although such properties can offer valuable insight into the design, Flownets are not a rich enough representation for the analysis of these. This is because the continuous behaviour is described at a high level of abstraction and does not formally specify timing constraints and data transformations.

5.2.2 Usability

Understanding properties relating to usability is less straightforward than those of correctness. This is because the properties are encapsulating knowledge about how a system should be formed based on characteristics of the way users interact. However,

past interactive system research (particularly in psychology) has developed a knowledge base of properties that are known to contribute to the usability of a system.

In [Fields, Merriam, and Dearden 1997] a distinction is made between descriptive and prescriptive representations of interactive systems. Descriptive representations are concerned with exposing characteristics of the system such that these can be analysed for usability. Prescriptive representations are concerned primarily with specifying how the system should be built. The advocated approach to usability analysis is the former, where the usability requirements determine the representations that should be used [Fields, Merriam, and Dearden 1997; Campos and Harrison 1998]. Clearly, this approach will yield a better system compared to the prescriptive approach, since a broader range of usability requirements can be identified and analysed. However, Flownets are a prescriptive representation since they are intended to describe a design. Consequently, it is necessary to consider not only which usability requirements are desirable but also which usability requirements (when formed into properties) can be analysed within the representation. This compromise reduces the range of requirements that can be considered. However, any analysis achieved is without the time consuming task of building further representations.

As noted in the previous section, it is difficult to express behavioural properties about the continuous components of a Flownet since the formal detail of such properties (timing, data transformation) are not described within the representation. This is not to say that useful usability analysis cannot take place on the continuous behaviour. A valuable usability property in the context of the locking door is: the virtual hand should be able to reach the door handle within a certain time period. However because of the abstract nature of the continuous components, our exploration of behavioural usability properties must focus on the discrete part of Flownets[3]. For interactive systems, the analysis of discrete behavioural representations for usability properties has been explored extensively (for instance, see [Sufrin and He 1990]). In the context of Petri-nets there are two properties relating to usability that are often analysed [Palanque, Bastide, Dourte, and Silbertin-Blane 1993; Bastide and Palanque 1990]. The first of these is the availability of states within the design:

- The user should be able to access every state of the behaviour.

This property is a liveness property and ensures there are no states of the interface that are not accessible by the user. The second property relates to absence of deadlock of states within the design:

- The user should be able to interact with the interface regardless of state.

[3]The HyNet formalism (discussed in chapter 2) would be more suitable for reasoning about such continuous properties because continuous behaviour is formally described.

This is also a liveness property which ensures that the user is not going to encounter the system in a state where it will no longer interact. Clearly properties of this type are relevant in the virtual environment context. However as we will discuss in the next section, there is potentially a conflict between these usability properties and the correctness properties described in the previous section.

Another way of viewing the states of a Flownet is as a mode. This is because the state is determining how the input of the user is interpreted. A particular concern of interactive systems is that they should adequately indicate the current mode so that the users do not suffer from mode confusion [Degani 1996]. Mode confusion occurs when the user thinks the system is in one mode when it is actually in another. The result is that the user misunderstands how the system will interpret their input.

Within Flownets, the continuous components provide a mapping between the state of interaction and the output to the external environment. By examining the mapping between the states and continuous outputs it is possible to determine which states output information to the external environment and which do not. From this it is not possible to say that the property of adequate representation of modes is satisfied, because there is no knowledge within a Flownet about how the data is rendered onto the external environment. However, it is possible to determine when the property cannot hold. The property of mode confusion can be described more succinctly:

- The user should be able to observe the state of interaction.

5.2.3 Discussion

At the beginning of this section we made a distinction between the correctness requirements of the system which ensure the functionality of the system, and the usability requirements which ensure that the system is usable. We argued that because of the nature of virtual environments, their design specifications capture both these types of requirements.

Since we are dealing with two sets of requirements with different concerns, a potential conflict can be identified between the desirability of the properties. This is because it is often necessary to reproduce usability problems that exist within the real world. For instance, if a behavioural design is constructed to simulate the behaviour of a gas oven (as we will do in chapter 7) then if the oven is in the state of the gas being on, there should be no (visual) external representation to the user of the gas. However if we apply the mode confusion property, then the mode must be rendered in order for the property to hold. Therefore, when usability properties fail to hold, it is necessary to ensure that this is not as a result of characteristics of the design pertaining to the correctness requirements.

5.3 Building a reachability tree

The types of properties we have discussed, with the exception of those relating to mode confusion, are concerned with the discrete part of a Flownet. In order to consider the discrete part of a Flownet independently of the continuous behaviour, an assumption must be made. This assumption is that all sensors relating the continuous to the discrete behaviour can always fire. When this is the case, the Flownet can be reduced to a standard event/condition Petri-net, since the semantics of the discrete part (as given in appendix A) are based on these, as explained in chapter 3.

With Petri-nets, behavioural properties are analysed by deriving a reachability tree [Reisig 1982]. The tree lists all possible traces of behaviours supported by the Petri-net. A reachability tree is produced by recursively firing each of the available transitions of the net beginning with the initial marking, and recording the new marking as leaves. This is a simple process for Petri-nets of the type we are using since each place can only contain one token and therefore has a boolean value.

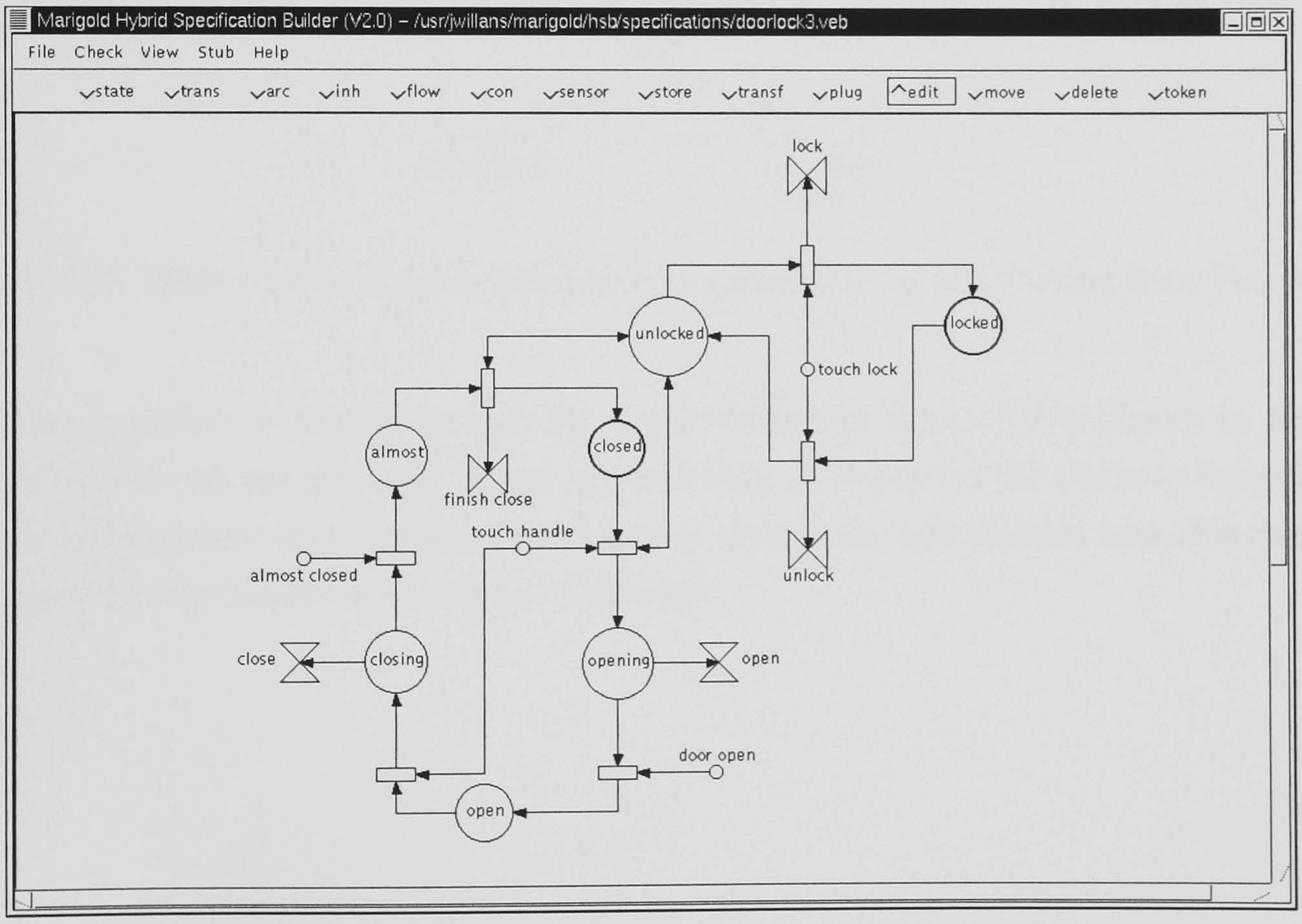

Figure 5.2: Discrete part of a Flownet specification for a locking door world object

Consider once again the locking door world object behaviour discussed in chapter 4. The discrete part of this Flownet is again shown in figure 5.2 within the Marigold HSB. Illustrated in figure 5.3 is the first part of the reachability tree derived from this Petri-net. At the root of the tree is the initial marking of the net. Here a '1' represents a token in a place and '0' representing the absence of a token in a place.

This marking specifies that there is a token in both the *locked* and the *closed* states. The only new marking that can be derived from the initial marking is for the token to be moved from the *locked* state to the *unlocked* state, as illustrated in figure 5.2 (a).

From this marking there are two options for new markings, these become leaves of (a). Firstly, as shown in figure 5.3 (b), the tokens can be moved from the *unlocked* and *closed* states and placed in the *unlocked* and *opening* states. Secondly, as shown in figure 5.3 (c), the token can be moved from the *unlocked* state back to the *locked* state. Since the latter marking has already occurred on the path from the root, the tree does not need to be extended beyond this node. Consequently, *stop* is placed next to this node and analysis need not continue further on this branch of the tree . However, further markings can be derived from figure 5.3 (b).

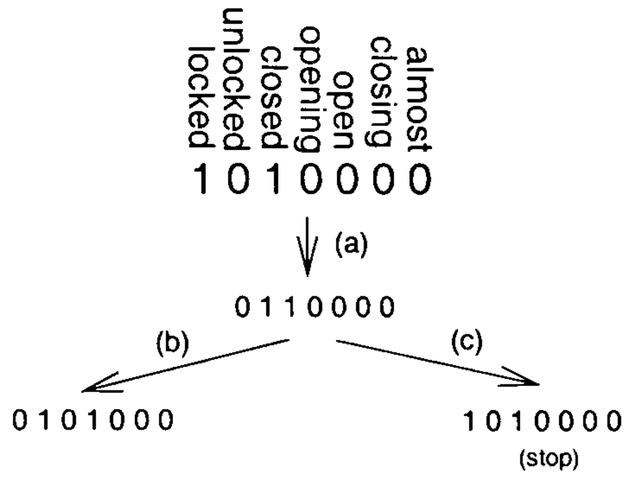

Figure 5.3: First part of the reachability tree generated for the locking door Petri-net

The complete reachability tree for the Petri-net of figure 5.2 is shown in figure 5.4. This lists all the possible states and ordering of states for the discrete behaviour of the locking door world object. In the next section we will discuss how this can be analysed for the properties we have discussed.

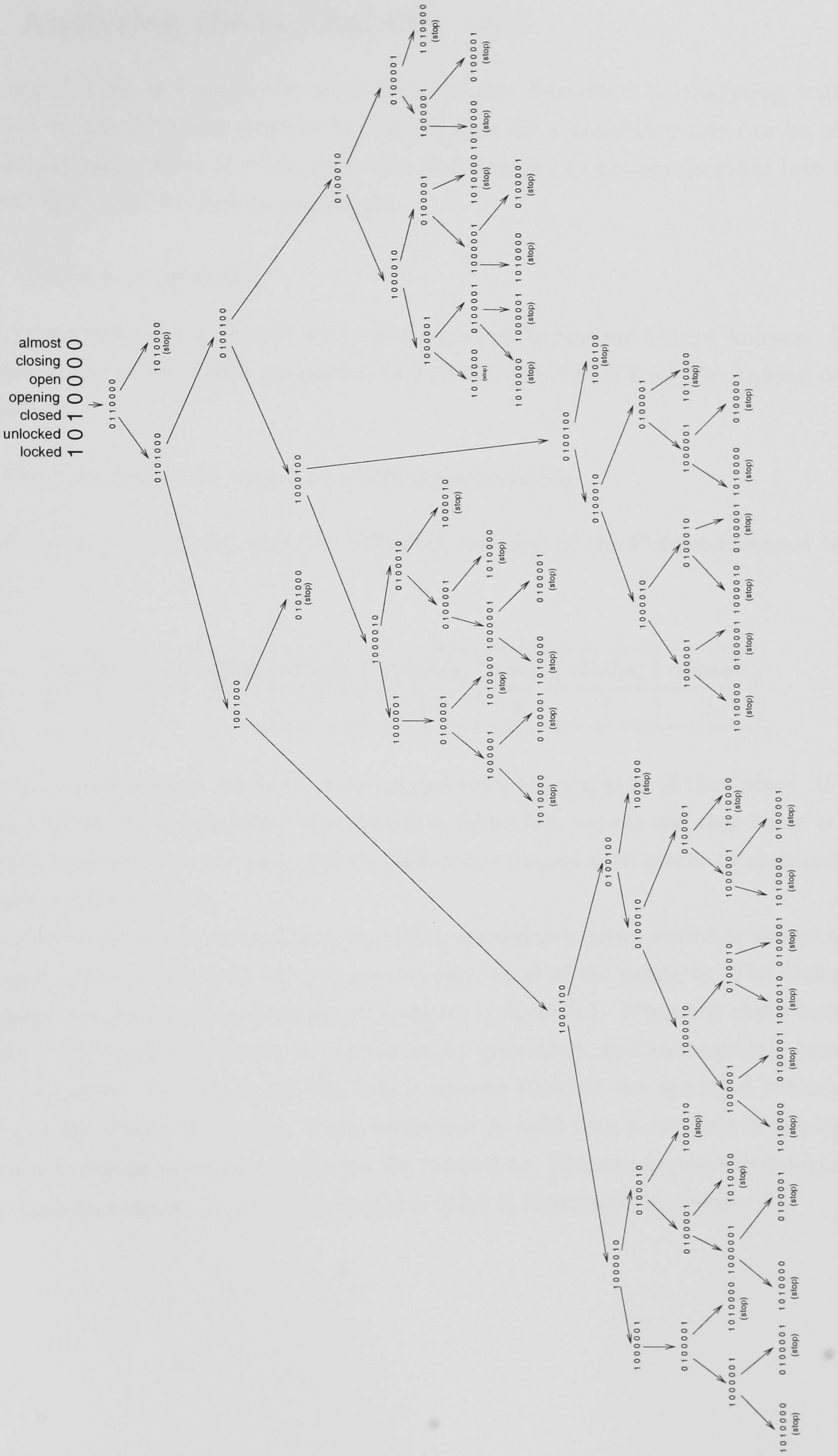

Figure 5.4: Complete reachability tree generated for the locking door Petri-net

5.4 Analysing the reachability tree

In section 5.2 we discussed the properties we are interested in analysing within Flownets. In the following sections we discuss how the reachability tree can be used to facilitate the analysis of such properties and how it can be incorporated into the Marigold HSB and checked automatically.

5.4.1 Safety properties

Safety properties are concerned with ensuring some behaviour cannot happen. Let us consider one of the safety properties described in section 5.2 for the locking door world object:

- The door *cannot* be open and closed simultaneously.

This property is specifying that the following marking of the Petri-net cannot take place:

locked	unlocked	closed	opening	open	closing	almost
?	?	1	?	1	?	?

Where a ? specifies that we are not concerned with the marking of that place. If we manually parse the reachability tree shown in figure 5.4, we can see that there is no node that matches this marking and the behaviour cannot take place. Consequently, the safety property holds.

In order to do this automatically, the *check reachability* menu option is chosen and a dialogue appears asking for the parameters (marking) of the property. The dialogue expressing the property described above is shown in figure 5.5. When the *check* button is pressed, the reachability tree is automatically generated, and subsequently parsed for the property. The Marigold HSB then reports whether the specified behaviour relating to the property is valid. If the behaviour is valid then a scenario is displayed which illustrates an example context for the behaviour. This can be used to determine how to change a design to satisfy a property (this is illustrated in chapter 7).

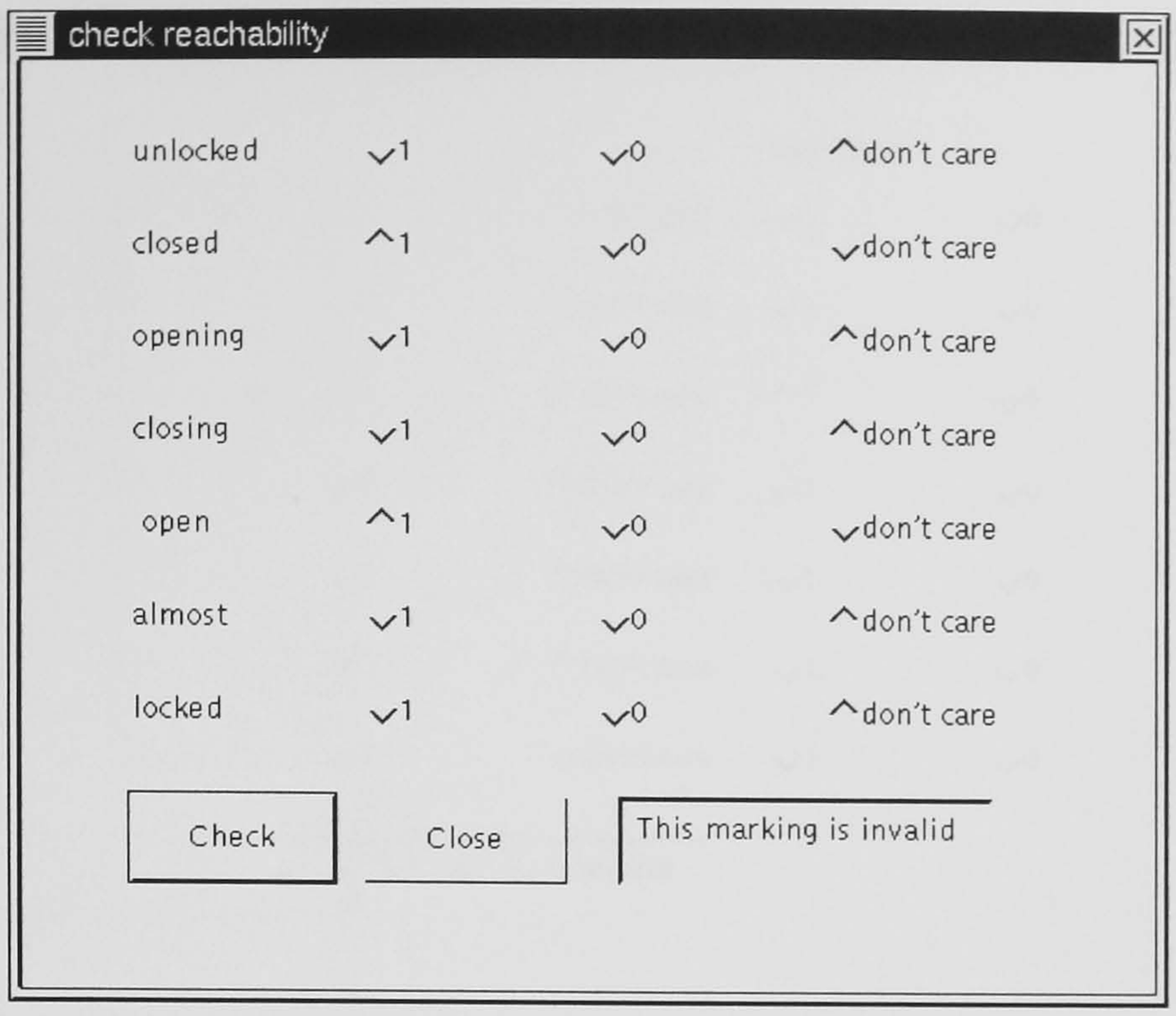

Figure 5.5: Dialogue box to check the reachability of a specific marking for the locking door

Let us consider one of the further liveness properties pertaining to correctness described in section 5.2. This described that for the locking door world object:

- A closed and locked door *cannot* be *immediately* opened.

This property is slightly more complex because it specifies that a sequence of states should not exist. That is, a behaviour does not occur where in the first state the door is closed and locked and the next state it is opening:

	locked	unlocked	closed	opening	open	closing	almost
first state	1	?	1	?	?	?	?
next state	?	?	?	1	?	?	?

Again the behaviour can be checked against the reachability tree in figure 5.4. This property holds since there is no sequence of nodes in the tree that matches the property.

Illustrated in figure 5.6 is the Marigold HSB interface to the analysis of sequence reachability properties. This is similar to the dialogue shown in figure 5.5 with an additional facility that specify the parameters of the *next state* marking. As in the previous example, the reachability tree is generated and the result derived automatically.

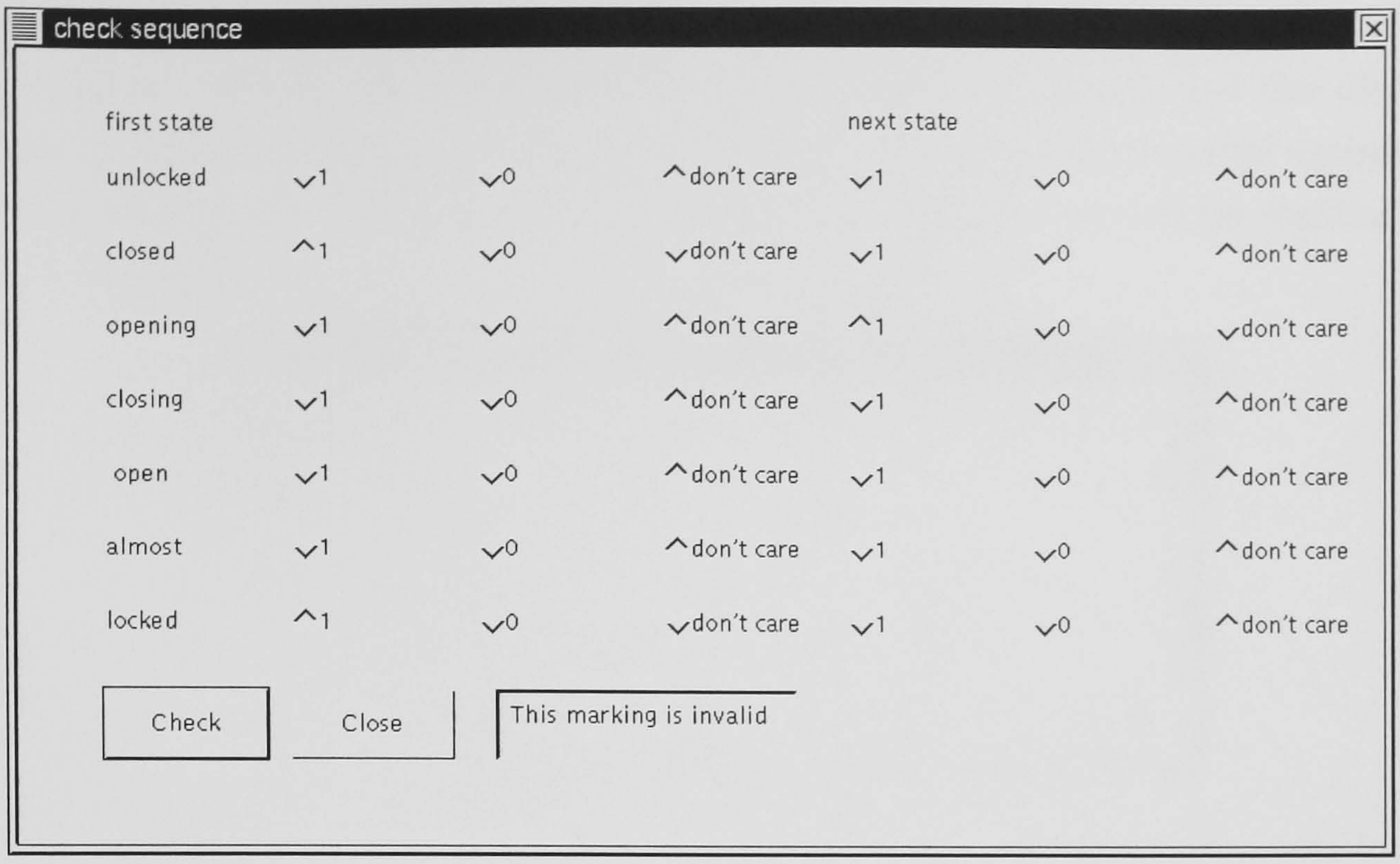

Figure 5.6: Dialogue box to check the reachability of a sequence of markings for the locking door

5.4.2 Liveness properties

Liveness properties are concerned with ensuring some behaviour can take place. In order to contribute to usability, one class of liveness property discussed in section 5.2 is that:

- The user should be able to access every state of the behaviour.

In order for this property to hold it is necessary to make sure a token is placed in every state of the behaviour at some point within the reachability tree. For the locking door, it is necessary to check the reachability tree of figure 5.4 to ensure each of the following markings are contained somewhere within the tree (each row is a separate analysis):

	locked	unlocked	closed	opening	open	closing	almost
check locked	1	?	?	?	?	?	?
check unlocked	?	1	?	?	?	?	?
check closed	?	?	1	?	?	?	?
check opening	?	?	?	1	?	?	?
check open	?	?	?	?	1	?	?
check closing	?	?	?	?	?	1	?
check almost	?	?	?	?	?	?	1

Since the property is a generic one, it is not necessary to specify any additional parameters. Within the Marigold HSB a menu option is chosen and the dialogue shown in figure 5.7 appears reporting the result of checking the markings against the reachability tree. It can be seen from this figure that all the states of the locking door are accessible.

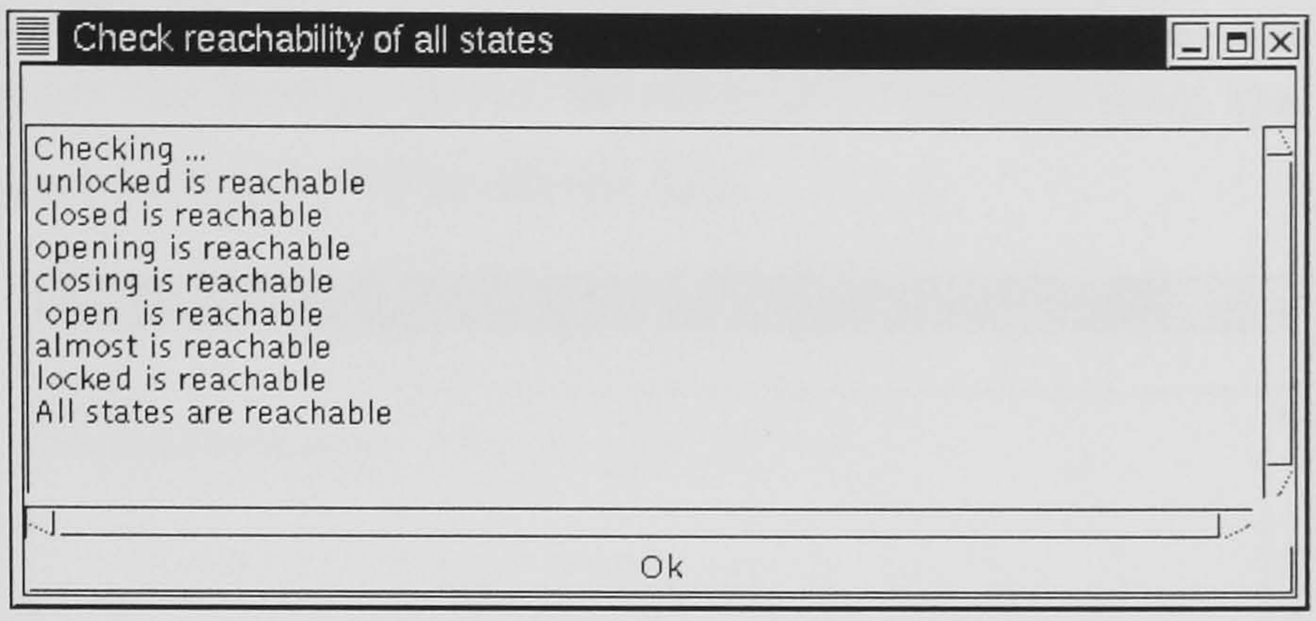

Figure 5.7: Dialogue box reporting that all states are reachable within the locking door

In order to illustrate this property failing to hold, an extra state (labelled *extra state*) was added to the locking door behavioural specification. This state has no initial token, and there were no arcs targeting the state in order to pass a token. When the analysis is applied this time, the dialogue illustrated in figure 5.8 appears. This indicates that the property fail to hold and the state which is responsible for this.

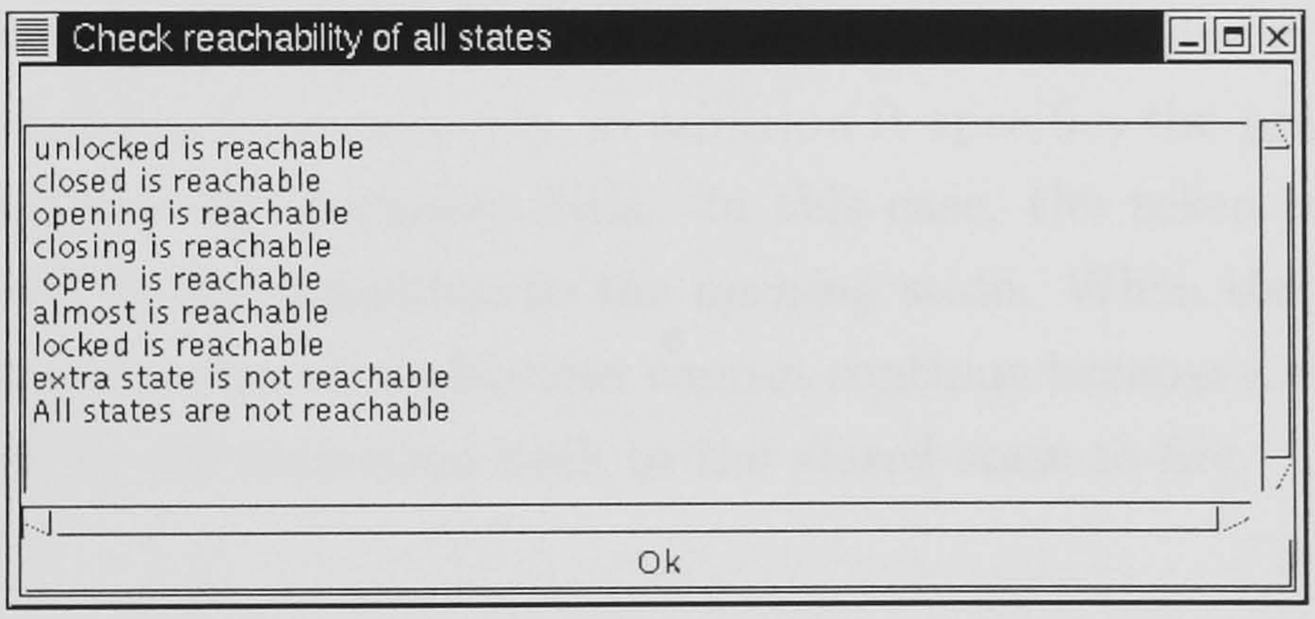

Figure 5.8: Dialogue box reporting that all states are not reachable within the locking door

Another type of liveness property that appears to contribute to usability (section 5.2) is that:

- The user should be able to interact with the interface regardless of state.

In terms of the reachability tree, this property is specifying that there is always at least one transition in the Petri-net which can fire (that deadlock cannot occur). This means that on no occasion should the reachability tree terminate. For the locking door, and its reachability tree illustrated in figure 5.4 each of the branches does terminate, however this is because the marking was encountered earlier in the behavioural trace. Consequently, the tree could continue infinitely, and the property described above does hold for the locking door. When this is checked from the Marigold HSB, the dialogue shown in figure 5.9 confirms this.

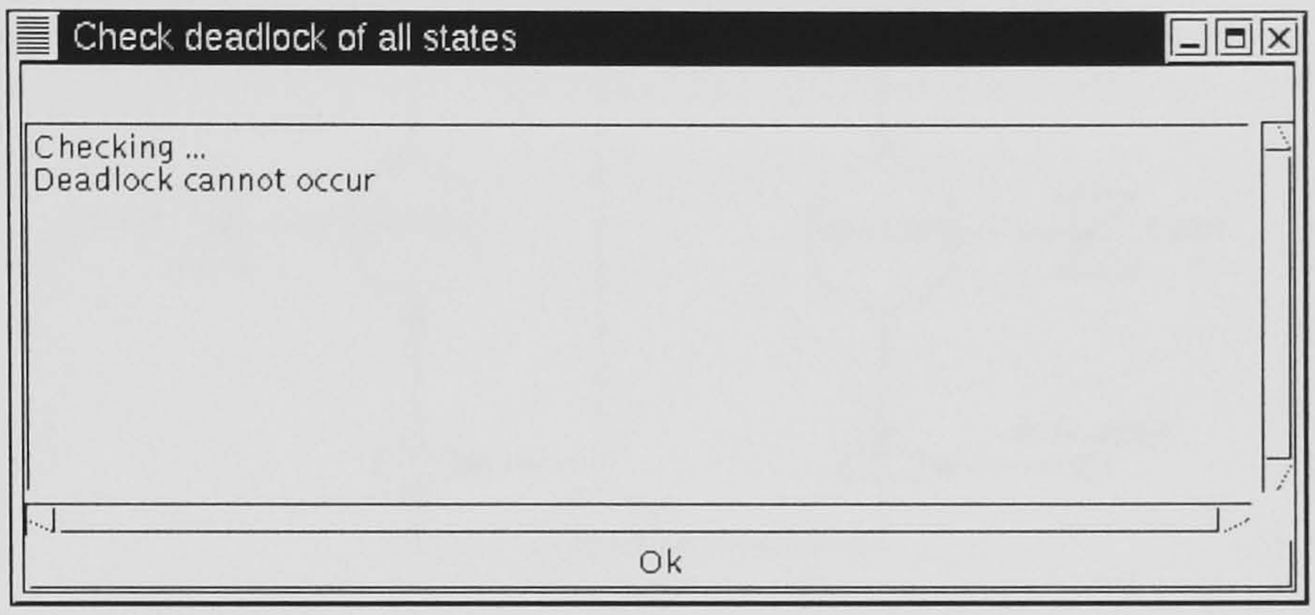

Figure 5.9: A dialogue box specifying that the locking door is free from deadlock

To illustrate the result of the property failing to hold, an amended version of the discrete part of the locking door is shown in figure 5.10. Within the amended design, the transition between the *closed* and *opening* state no longer replaces the token it removes from the *unlocked* state (as is the case in the original design shown in figure 5.2). When the analysis is applied this time, the resulting dialogue specifies the anticipated failure of the property, in addition it specifies the precise marking(s) of the Petri-net where the property fails. In this case, the token is removed from the *unlocked* state by the transition to the *opening* state. When the token is moved around to the *almost* state, the behaviour cannot continue because a token is required in *unlocked* state for the transition back to the *closed* state to fire.

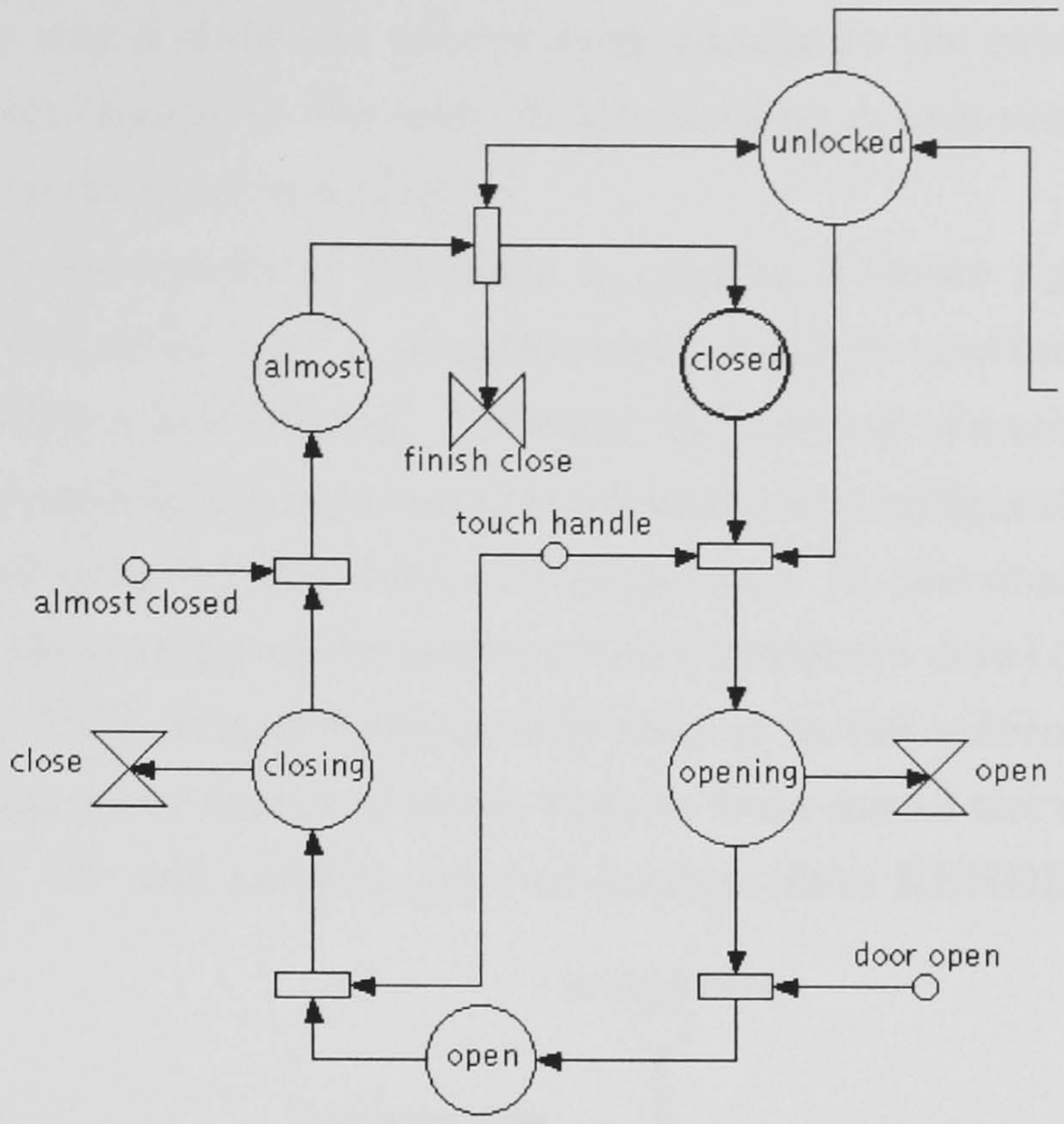

Figure 5.10: An amended design of the locking door to illustrate deadlock

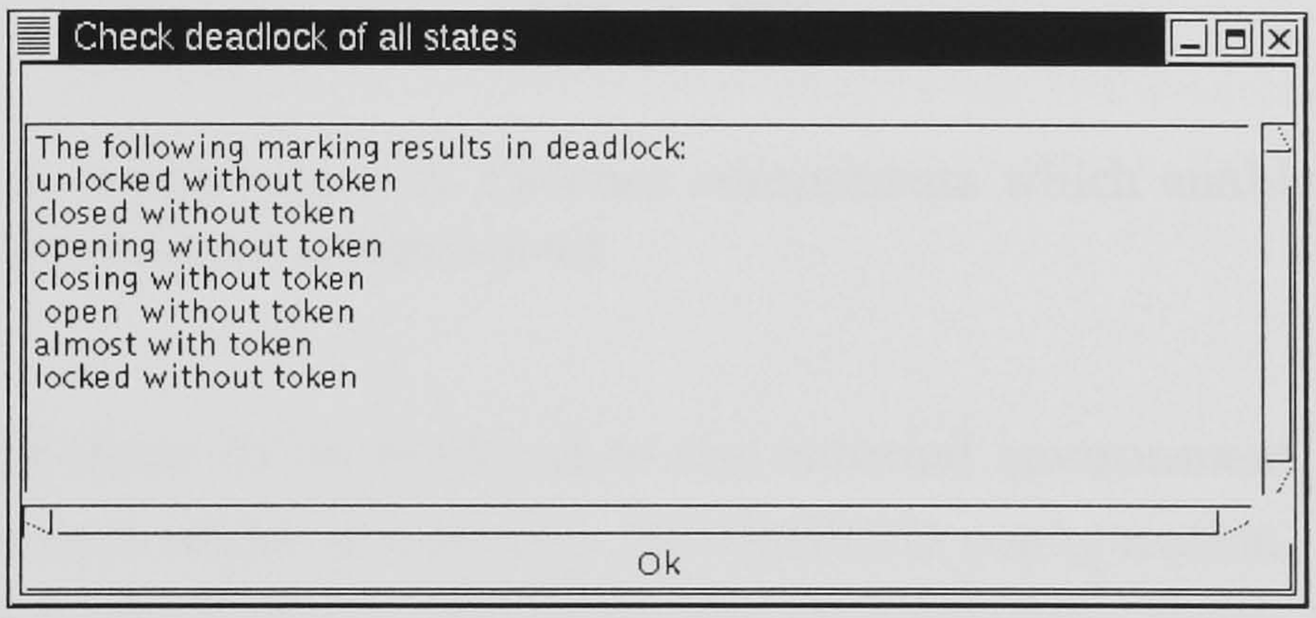

Figure 5.11: The dialogue reporting that the amended design of the locking door suffers from deadlock

5.5 Mode confusion analysis

The final usability property discussed in section 5.2 is concerned with ensuring that mode confusion does not take place:

- The user should be able to observe the state of interaction.

As described in section 5.2, the states of a Flownet can be considered as modes because they define how the user's interaction is interpreted. We are interested in how each of

the states within a Flownet design maps to the external environment via plugs. This is because the only way a state can render some change to the external environment, and indicate a mode change to the user, is by enabling continuous behaviour which transforms and outputs data to a plug.

In our informal description of Flownets in chapter 3 (more rigorously defined in appendix A), we described how a discrete state controls continuous behaviour by enabling and disabling a flow control. Although the process of enabling a flow control does not render a change to the external environment, the configuration of components shown in figure 5.12 does (at this level of abstraction). In this configuration the flow control is enabling the continuous transformation (*transform data*) of the data residing in the store (*data*). This data is subsequently output to the external environment. If a transformer targets more than one store, then at least one of the stores must output its data via a plug. We will call this type of configuration RENDERED.

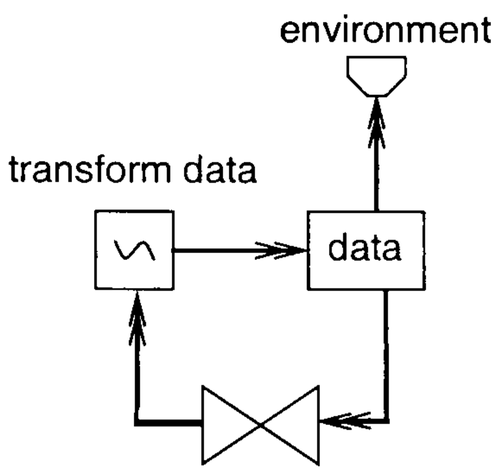

Figure 5.12: The configuration of Flownet components which enables the rendering of a change to the external environment

In order for a state to be rendered to the external environment, it can directly enable a flow control which is part of a RENDERED configuration. An example of this is shown in figure 5.13 (a). Alternatively, all transitions which target the state must enable flow controls which are part of a RENDERED configuration, an example of this is shown in figure 5.13 (b). This ensures that some state rendering takes place just before the state is reached. The final method of ensuring that a state is rendered is if all states that target the state enable a flow control that is part of a RENDERED configuration. Even though the new state does not perform a rendering, since one of the old states must have been continuously rendering, the absence of this consequently indicates the new state. This is illustrated in figure 5.13 (c).

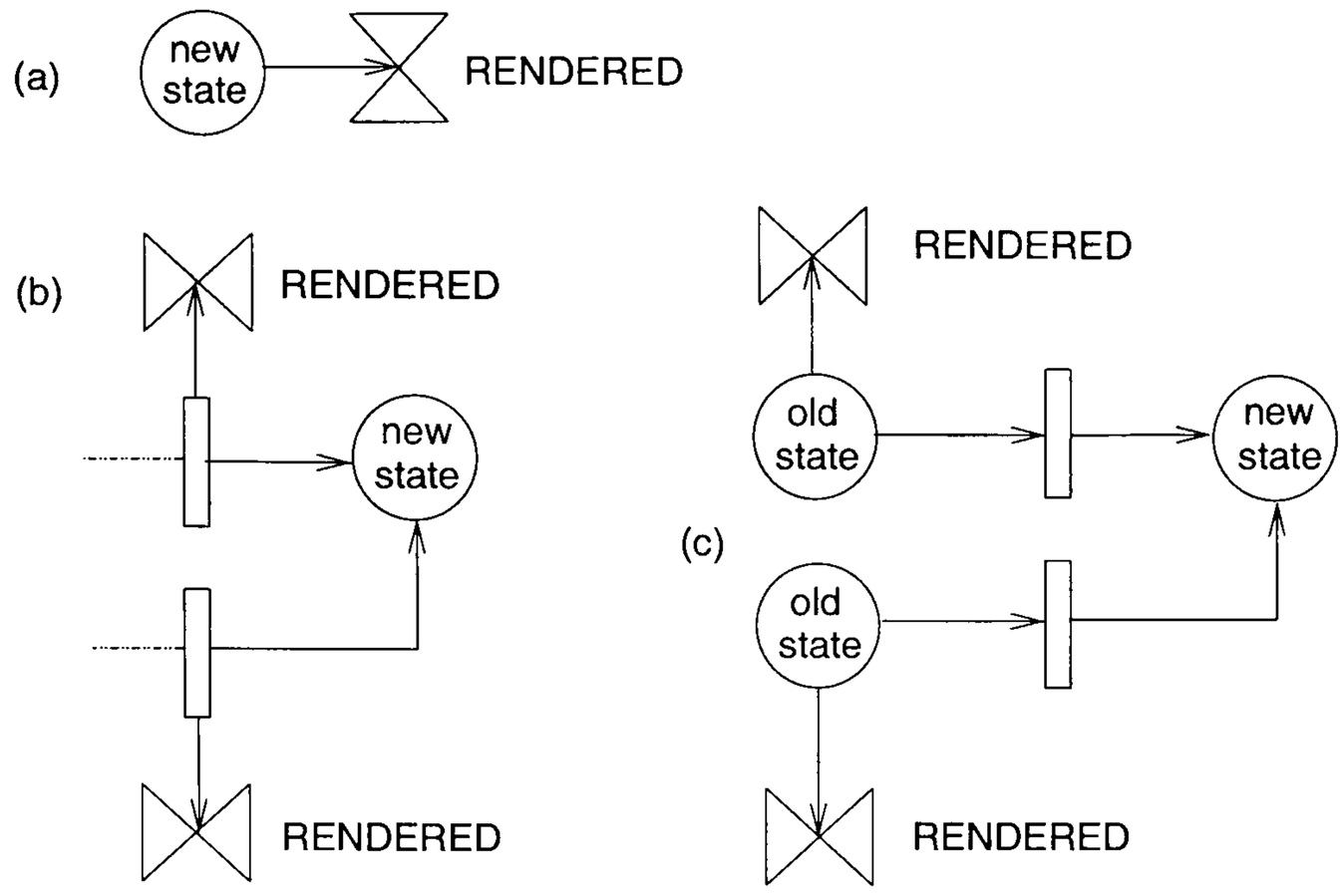

Figure 5.13: Rendering a new state to the external environment

5.5.1 Applying the analysis

The HSB incorporates a mechanism for checking the presence of the configurations illustrated in figure 5.13. Consider again the mouse-based flying interaction technique described in the chapter 4 (figure 5.14). When, the *mode confusion* analysis option is chosen from the Marigold HSB menu bar, the dialogue shown in figure 5.15 appears. This reports that there is a failure in the rendering so mode confusion can take place. The analysis also reports that the *idle* state is responsible for this. Consequently, the user may not able to perceive from the state of the environment whether the technique has not been started or whether they are in the *idle* state.

In order to address this problem, the design of mouse-based flying was revised to incorporate a mapping of the *origin* position to the external environment. This revision is shown in figure 5.16. In this design, when the *start* transition fires and places a token in the *idle* state, the resulting transformation is output to the *origin pos* plug. Figure 5.17 shows the application of mode confusion analysis to the revised design which confirms that the property no longer fails.

5.5.2 Discussion

As indicated in section 5.2 the analysis approach described previously is not able to ensure that mode confusion will not take place. There is no guarantee that a RENDERED transformation does transform the external environment in such a way that the user is aware of the mode. For instance, the transformation enabled by a new state may be the same transformation that took place in the previous state. Even in the case where the transformation is unique, a Flownet does not determine that the

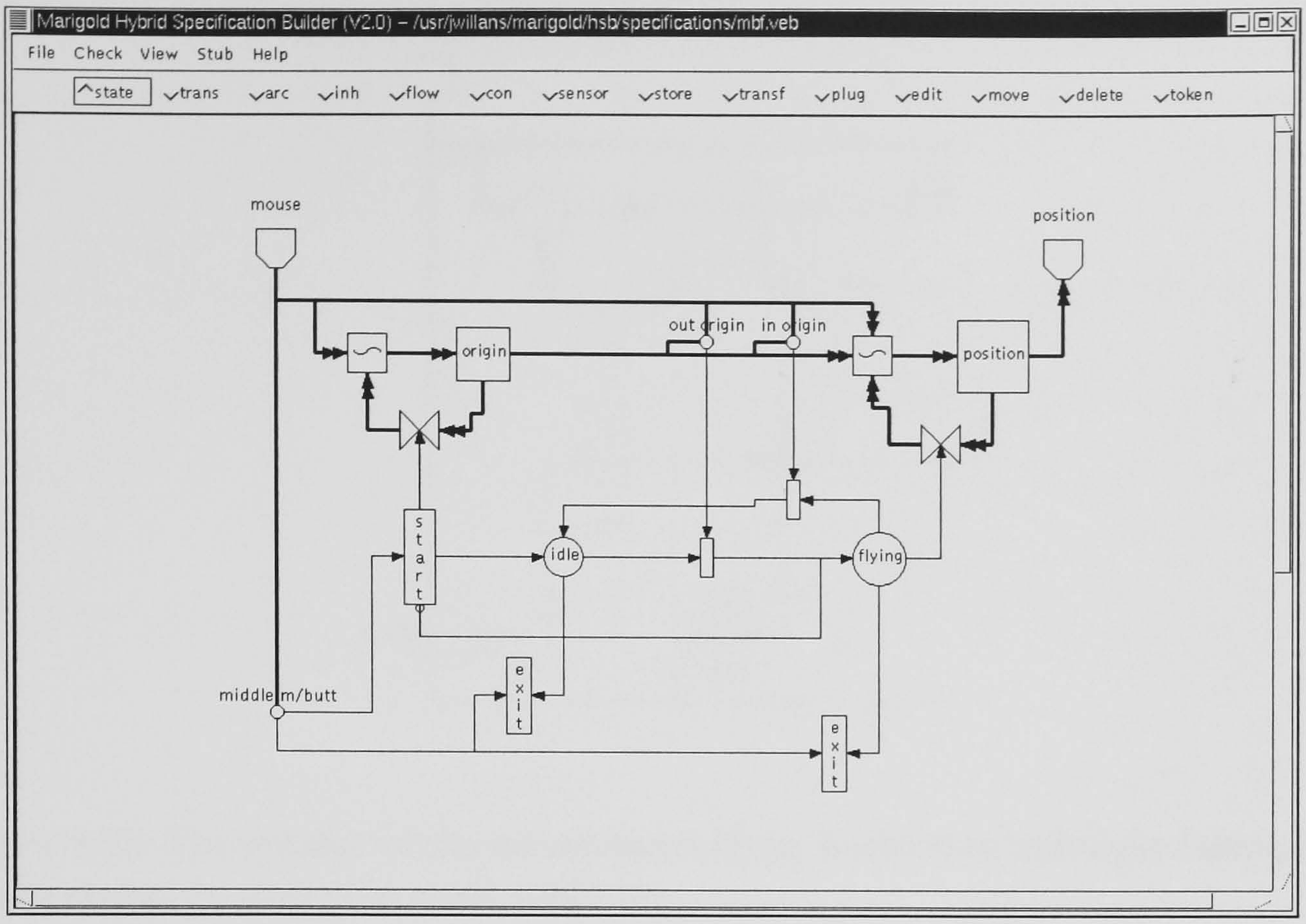

Figure 5.14: Flownet specification for the mouse-based flying interaction technique

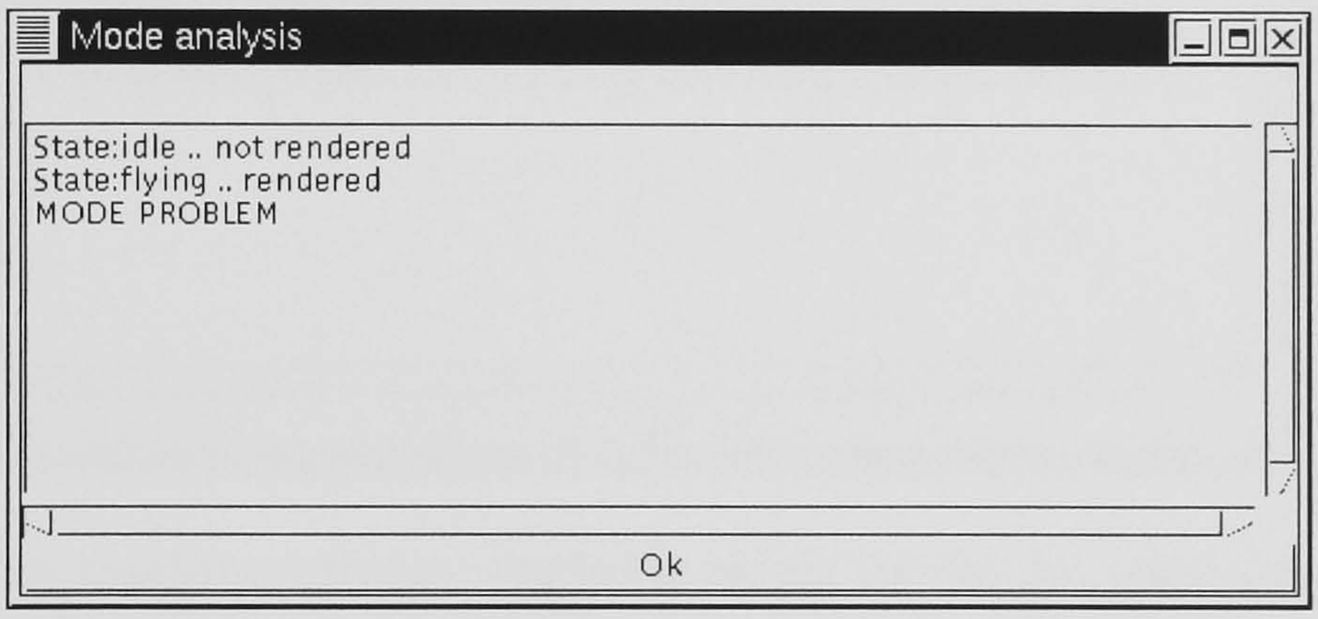

Figure 5.15: The dialogue to the mode checking analysis reporting that mouse-based flying may cause mode confusion

data is presented in a manner that prevents mode confusion. Hence, the analysis will never report that mode confusion will not take place, rather: *moding seems okay*.

However, we can say with a greater level of certainty that unless states are related to the external environment in the manner described in the previous section, then it is not possible for the user to observe the mode of interaction. As illustrated in the previous example, the results of this analysis can be subsequently used to revise the design.

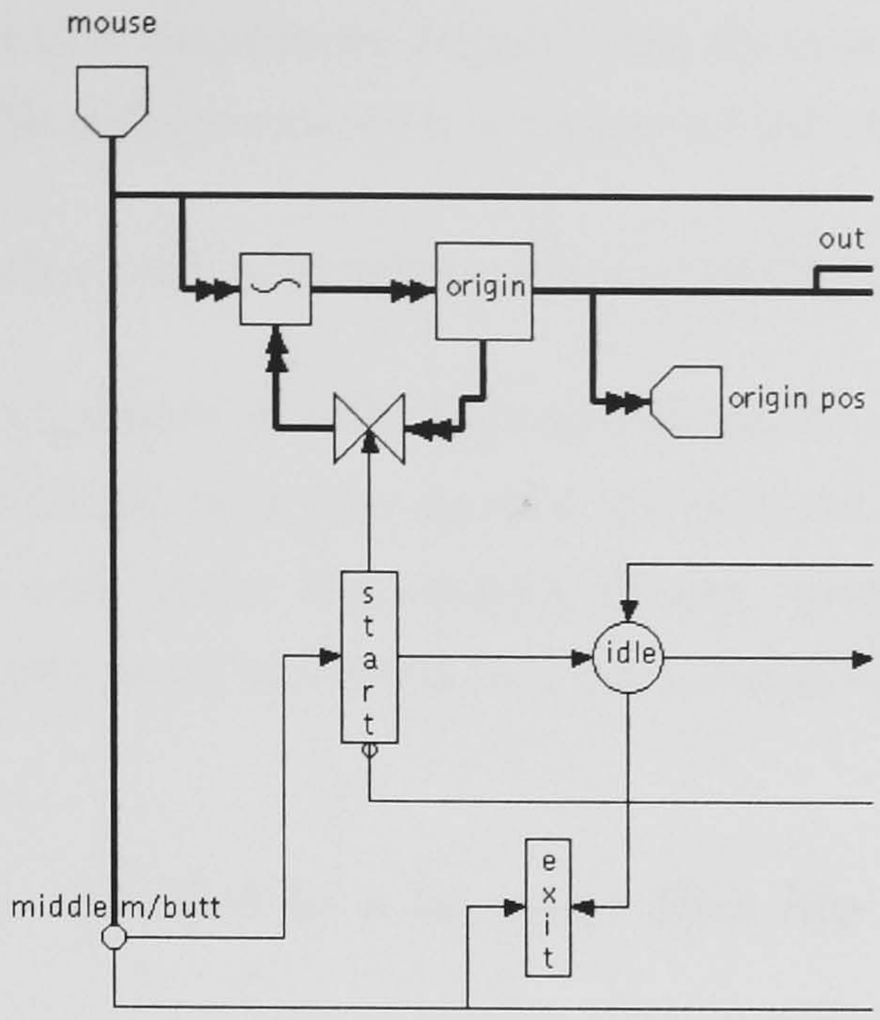

Figure 5.16: The revision of the mouse-based flying interaction technique taking into consideration the potential mode confusion

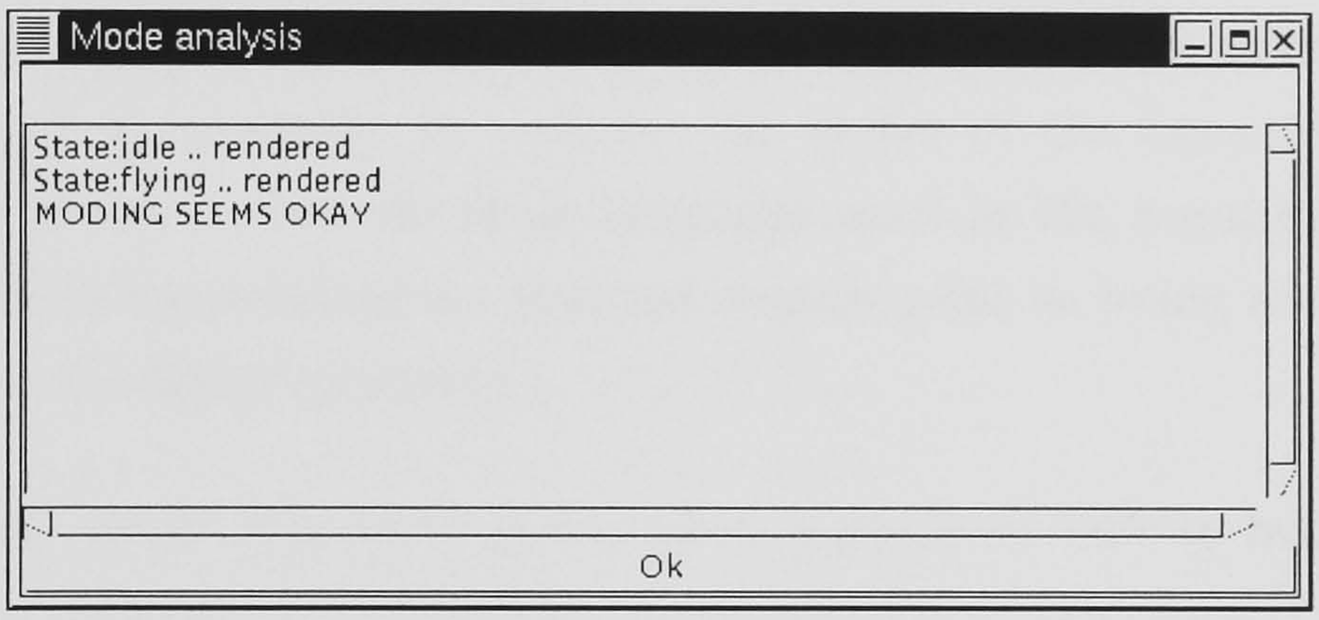

Figure 5.17: The mode confusion analysis result of the revised mouse-based flying design

5.6 Discussion

We have described how the analysis of Flownet specifications can provide useful insight into the designs of virtual environment behaviour. The main departure from traditional interface analysis approaches is that this analysis is concerned with properties concerning correctness in addition to usability. This insight is a valuable perspective on virtual environment interfaces and highlights the complexity of their successful design. In a wider context, it suggests that those forms of interface evaluation techniques concerned with usability cannot be transferred directly to the domain of virtual environments.

In the introduction to this chapter we argued that there are three desirable criteria for an analysis process. We will discuss each of these in view of the presented approach:

- The use of automation can help assure the accuracy of the analysis.

Each of the evaluation techniques is fully automated within the Marigold HSB. Consequently, we can be confident that the results are accurate. In addition, by using automation the analysis can derive the results almost immediately. This has been the case for all the Flownets analysed including the locking door example used in this chapter[4].

- Properties must be specified in a language that has a close mapping to the requirements.

The usability properties (including mode confusion) are generic, so there is no need to specify any additional parameters when applying such analysis. This analysis is achieved by choosing a menu option. By contrast, the correctness properties are concerned with specific characteristics of the Flownet and require additional parameters. These parameters are specified in the Marigold HSB dialogues (figure 5.6, for instance) where it is necessary to indicate the states of the behaviour that can be reached. These states are in the same language used in the requirements. Thus, it is a short step from formulating an abstract requirement to being able to specify the requirement as a checkable property.

- Meaningful results should be given when a property fails to hold.

If an undesirable behaviour is supported by the design (a safety property fails) the interesting issue is the causation of the behaviour. When safety properties relating to correctness properties are checked within the Marigold HSB, and found to fail, the tool gives an example of a scenario which would lead to this being the case (a state trace). This insight can be used to redesign the behaviour. When usability properties that fail to hold in a design are analysed, the Marigold HSB reports either the state or the sequence of states that cause this to be the case. Again, this can be used to pinpoint the problem with the design.

By meeting these criteria, Flownet analysis provides a practical insight into the meaning of behavioural designs without resorting to the comparatively time consuming approach of prototyping. A limitation of the described approach is that the analysis is restricted to one Flownet specification. As illustrated in chapter 4, an

[4]To give some indication of speed, the time taken to return results for the locking door properties described in this chapter ranges from 1 to 91 milliseconds on a Pentium 400MHZ with 128MB of ram.

environment is likely to be designed using many specifications. Since these cannot be composed for analysis, there is no insight into the relation between the behaviours. This must be evaluated from a prototype. Although an advantage of analysing smaller units in this way, is that the result is derived in a much shorter time.

A facility within the Marigold HSB, not explored in this chapter, is the ability to export the Petri-net part of the Flownet description into a file format loadable by the integrated net analyser tool (INA) [Roch and Starke 1999]. INA offers a powerful approach to analysing Petri-nets beyond those facilities supported directly by the Marigold HSB. Potentially for larger nets, analysis results can be derived faster using the INA tool since it offers the ability to reduce the net while still preserving the necessary semantics to prove a property.

5.7 Conclusion

In this chapter we have described how requirements concerning virtual environment behaviour can be formulated into properties and subsequently analysed within Flownet specifications. We have argued that these properties should concern both correctness and usability requirements and demonstrated how properties of these types can be automatically checked from the Marigold HSB. A particular emphasis has been on ensuring the approach is usable by supporting the natural specification properties, and returning meaningful results when properties fails to hold.

Evaluating Flownet specification in this manner offers a complementary technique to evaluating behavioural designs using prototypes in isolation. This is particularly the case since the facilities to support the analysis of Flownets are built into the toolset (Marigold) which also supports the prototyping of Flownets.

Chapter 6

Virtual environment requirements specification

In the previous chapters we have explored evaluation approaches which aim to ensure that designs of virtual environment behaviour are correct. Within these approaches there has been a tacit understanding that virtual environment designers will have a clear idea about the requirements for the virtual environment. In practice, this is not the case. An important challenge is communicating the requirements of the end-user to the designer in a manner that ensures the resulting designs are correct. This chapter presents an approach to eliciting and specifying requirements when the virtual environment is based on the real world.

6.1 Introduction

The study of the virtual environment development process described in [Kaur, Maiden, and Sutcliffe 1996] demonstrates that in practice designers are unclear about the requirements for designs:

> It was easy for a designer to overlook what a user would be focussing on in a model and spend equal amounts of time working on important and less important parts, for example designing a chair with height adjustments. Designers found problems judging the perceptibility of visual features to the user; often creating over-complex environments because they assumed users would notice every detail.

They note that designers used photographs and informal conversations with the users to guide the designs [Kaur, Maiden, and Sutcliffe 1996]. However, the designers did not apply a formal approach of eliciting and documenting the requirements of the users. Such an approach is critical to accurately informing a designer of the

requirements for designs [Sommerville 1996, p64]. In this chapter we explore the specification of requirements for virtual environments.

There are two major stakeholders in requirements specification: end-users of the system and designers [Cybulski and Reed 1999]. Each of these stakeholders has different and conflicting concerns. Designers prefer the requirements specified in a form that allows them to be easily mapped to designs. Users prefer to communicate their requirements in a familiar (non-technical) language. These concerns must be addressed if a requirements specification approach is to be successful:

- An approach should elicit the requirements in a language natural to the users.
- An approach should refine the elicited requirements into a language natural to the designers.

The approach presented in this chapter aims to address this dual concern.

6.2 Overview

In this section we give an overview of the approach to requirement specification. The output of the approach is a specification for a virtual environment designer which documents the requirements. Using the specification, along with reference material such as photographs, a designer is clearly informed about detail of the real world that the user requires reproduced in the virtual environment.

The first step in the approach is forming a problem statement which describes the overall aim of the virtual environment being designed. The problem statement is refined to a series of scenarios. In the scenarios the users describe episodes of interaction with the real world which they require simulated in the virtual environment. Scenarios are a natural language for the user to express requirements because they deal with their viewpoint rather than a conceptualisation of the requirements at an abstract level [Kutti 1995]. This also enables consideration of different viewpoints of potential users (stakeholders). For instance, in a building fire evacuation environment (as presented in [Higgett and Bhullar 1998]) one concern might be the viewpoint of the occupants of the building, but a further concern might be the viewpoint of the fire service.

Although scenarios document user requirements, these requirements are clouded by a clutter of non-relevant detail (as far as the designers are concerned) as well as being spread across multiple representations. Interpreting the implications of scenarios directly to a design is non-trivial and error-prone. To make the transition from requirements to design more reliable, the requirements that are important to a designer must be extracted and structured into a usable form.

In order to understand what the important requirements are for a virtual environment designer, it is necessary to examine design considerations that the designers must make. For the software part of the virtual environment interface there are, as discussed in chapter 1, two major components that must be considered. Firstly, the world objects that are rendered to a user and, secondly, the behavioural rules that determine how the environment responds to user interaction. When the virtual environment is based on the real world, the major concern is the level of realism with which these two components of the environment simulate the real world. In view of this, we derive four main considerations a virtual environment designer must make. We call these the *key* requirement types:

- World object appearance. World objects of the environment must appear at a level of detail which is appropriate to their role. If a world object is not a critical part of the requirement then it can appear at a low level of detail (wire frame, for instance). Indeed a world object that is not critical to the requirements but appears at a high level of detail (relative to other world objects) can give false cues to interaction and lead to usability problems. On the other hand, a world object which is critical to the requirements should appear at a high level of detail (photo realistic, for instance).

- World object decomposition. World objects must be decomposed appropriate to the behaviour they support. To illustrate this, consider a virtual drawer unit. If the drawer unit remains static during the execution of the virtual environment, it can be constructed using a single world object. However, if a drawer in the drawer unit should open and close, the drawer unit must be decomposed into two world objects: the drawer and the drawer unit. If the drawer unit has a handle which turns, then the handle becomes a further world object.

- World object behaviour. If the behaviour of a world object is not critical to the requirements then it can be simulated at a rough level of realism. For instance, a virtual door may simply have an open and closed state. If the behaviour is critical to the requirements then it may be simulated more realistically and include movement between the open and closed state.

- User behaviour. Behaviour (interaction techniques) must be provided that enables the user to interact with the virtual environment. For instance, if it is necessary for a user to be able to open a virtual drawer, then the design of the behaviour must take this into consideration.

The scenarios are analysed and requirements of the types discussed above are identified. These key requirements are used to build a specification we call a require-

ments tree[1] (a single tree is constructed for each environment). The purpose of the requirements tree is to consolidate the key requirements of the scenarios into a coherent form that can be interpreted by designers. An overview of the approach is illustrated in figure 6.1.

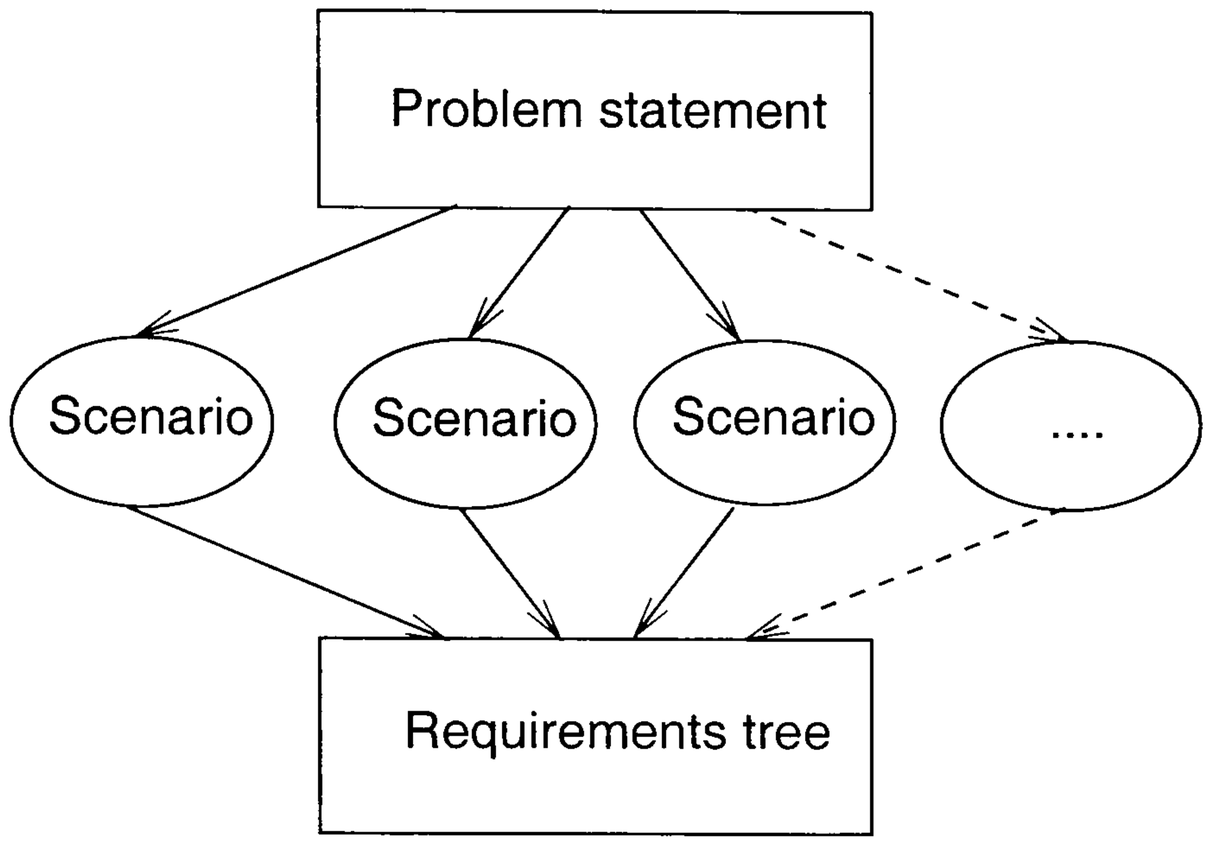

Figure 6.1: Overview of requirements specification approach

6.3 Applying the approach

Two forms of specification are used in this approach. Scenarios are used to extract the requirements from the user (section 6.3.1). The requirements tree is used to structure the key requirements for the designer (section 6.3.2). The process involves analysing the scenarios in order to construct the requirement tree (section 6.3.3). Once the requirement tree is constructed, it can be used as a basis for designs (section 6.3.4). The following simple problem statement will be used to illustrate the discussion: "provide a virtual environment that allows the user to explore their office".

6.3.1 Eliciting user requirements

Scenarios provide an effective way of eliciting requirements from users [Kutti 1995]. The user describes typical interactions with the real world which should be simulated in the virtual environment and these are documented. This approach places no constraints on how the interaction is described. This allows the user to (unconsciously) indicate a level of real world realism which concerns them. To illustrate this, consider the following scenario-like description of dealing with a photocopy in the office space:

[1] Not to be confused with a scene graph tree specification often used in computer graphics.

> To photocopy a document, I open the lid, place my document on the glass plate, close the lid, and press the green switch. I then open the lid, remove my document and collect the photocopy.

and from the perspective of someone changing the photocopier's toner:

> To change the toner cartridge on the photocopier I open the maintenance door and raise the access flap. I then release the toner lock and the toner cartridge moves towards me. I pull the cartridge out of the photocopier and slide in a new one. I push the cartridge completely into place, lock the cartridge and close both the access flap and then the maintenance door.

These descriptions are dealing with a real world photocopiers which can support both interactions. But the virtual environment photocopier would need to include different realism depending on which scenario it should support. For instance, in order to support the photocopying of a document in the virtual world, it is not necessary to facilitate access to the toner cartridge. In this way, scenarios are a filter on the real world which identifies detail which the user considers important and blocks detail which they consider unimportant.

To model the scenarios we use state transition diagrams. The states of the formalism describe a static representation of the real world, and the transitions between the states describe the behaviour that occurs in order to transform the real world from one state to the next. State transition diagrams are particularly desirable (compared to the more commonly used structured textual descriptions) since multiple scenarios that differ only slightly can be modelled within the same diagram using transition branching. An example of a state transition scenario description is shown in figure 6.2, describing how the user can open a window in their office.

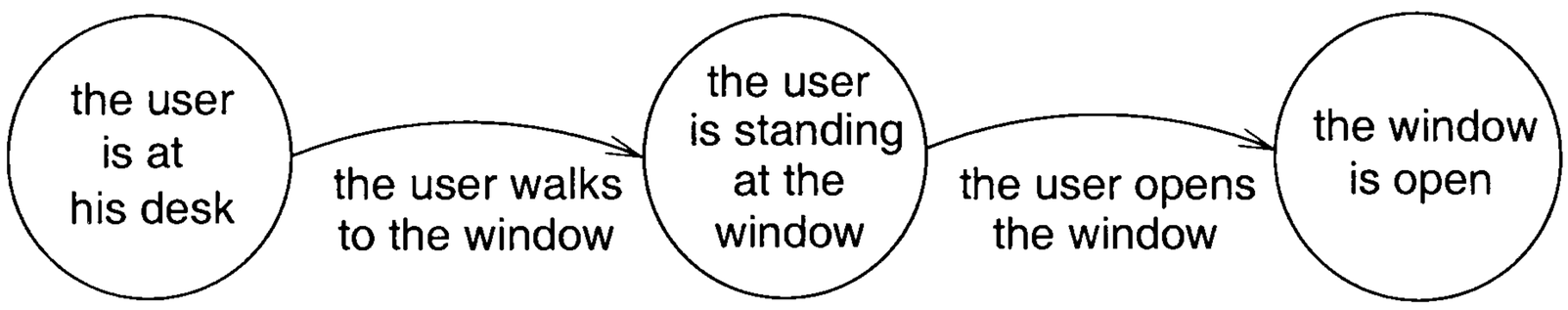

Figure 6.2: A scenario describing how the user opens a window in their office

6.3.2 Specifying designer requirements

Once the scenarios have been specified, they must be analysed so that the key requirements can be gathered. In order to describe the requirements, and the relations between the requirements, we use the requirements tree (see figure 6.3 for an example).

Prior to analysis of the scenarios, there are two nodes in this requirements tree: the *environment* which is the root node, and the *user* which is a branch of the root node. The relation between a parent node and a child node in the tree should be read as *consists of*. Thus, initially the environment *consists of* a user. Two types of nodes can be added to the tree during the analysis of the scenarios.

The first of these are world object nodes. These can either be a child of the environment node or a child of an existing world object node. The structure of world object nodes in the tree describes the decompositional requirements of world objects. For instance in figure 6.3, *world object 1* should be decomposed into *world object 1a* and *world object 1b*. For each world object node added to the tree at the lowest level, it is necessary to annotate it with one of the following tags describing its significance:

- Background: are not critical to the scenario.
- Contextual: are important to the scenario but not the focus.
- Task: are central to the scenario.

This tag specifies the requirements concerning the relative level of realism with which world objects should appear. For example in figure 6.3, *world object 1a* is *task* so should appear at a relatively high level of detail while *world object 1b* is *contextual* and can appear at a relatively lower level of realism.

The second type of node that can be added to the requirements tree are the behavioural nodes. These can either be a child of a world object node or of the user node and are annotated with a <be> tag to differentiate these from world object nodes. These nodes describe the behavioural requirements for world objects and for the user.

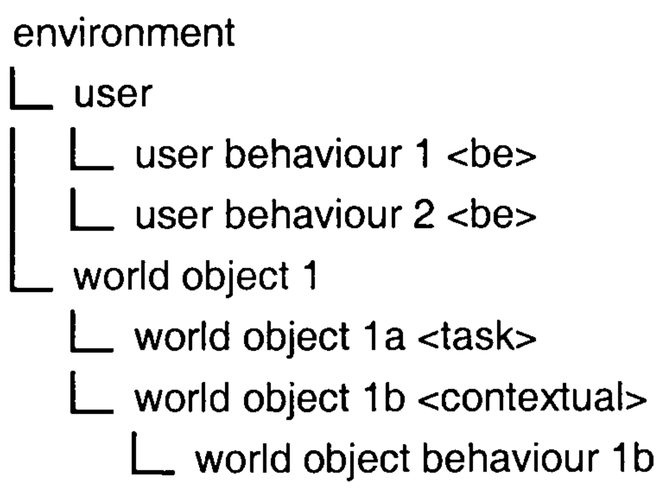

Figure 6.3: The structuring of key requirements in the requirements tree

6.3.3 From scenarios to requirements tree

We now illustrate the transition of the key requirements from the scenarios to a tree using one scenario (figure 6.2). Prior to analysing the scenario, the tree consists of an *environment* and a *user* (figure 6.4 a).

The first state of the scenario relates that *the user is sitting at his desk.* It can be deduced therefore that a *chair* and a *desk* world object are implicated and should be added to the tree (figure 6.4 b). These world objects are *contextual* - they are not the focus of the scenario.

In the next transition *the user walks to the window.* There should therefore be a *window* world object in the environment. It appears that this is not the focus of the scenario, so the label is again *contextual.* No further requirements are established by the next state.

The user next *opens the window.* This establishes the behavioural requirements that the window can be opened and closed and produces the need to decompose its existing representation in the requirements tree into a *window* and a *window frame* world object. The *window frame* is described as *contextual*, but the *window* itself as *task* since this is critical to the scenario. The *window* is also assigned the behavioural requirement that it can be *opened* and *closed.* Nothing further is added by the last state of the scenario. The final requirements tree is shown in figure 6.4 (d).

The requirements catalogued in the requirements tree should support all scenarios that are analysed. Consequently, during analysis existing requirements in the tree can be upgraded but never downgraded. To illustrate the reasoning behind this point consider two scenarios A and B. Scenario A determines that the requirements tree should include a door world object. Scenario B determines that the same door should have open and close behaviour. If scenario A is analysed followed by B, then the door should be upgraded to include the open and close behaviour. This is because the requirements tree still supports both scenarios since it is of no consequence to scenario A that the door should behave. However, if scenario B is analysed followed by A, the door should not be downgraded to have no behaviour, since this will no longer support scenario B.

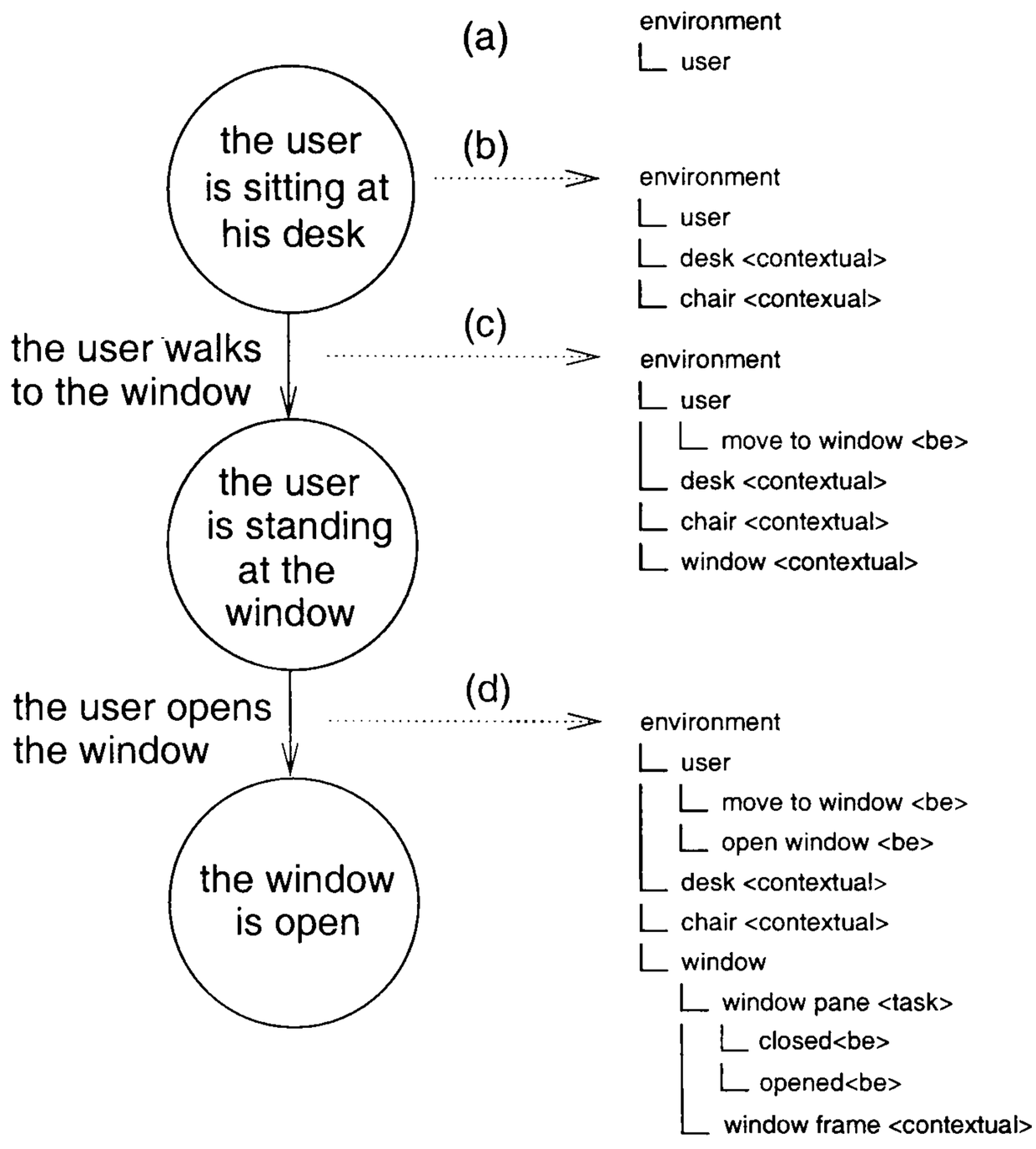

Figure 6.4: The evolution of the requirements tree (right) as the example scenario (left) is analysed

6.3.4 From requirements tree to designs

The important characteristic of the requirements tree is that it coherently documents the virtual environment requirements for designers. This enables designers to be informed of the requirements as they are designing virtual environments.

The structure of the requirements tree describes how world objects should be decomposed. As illustrated in figure 6.5 this structure can be used to construct the world objects using a 3D modeller. In this example the *window pane* and *window frame* become separate world objects in order to support the requirement that the window should open and close. Similarly, the requirements tree informs the designer the level of realism with which each world object component should appear. In figure 6.5 the *window pane* has a high realism of appearance because it is the focus of interaction, while the other three world objects has a (comparable) lower realism of appearance.

The requirements tree describes the behavioural requirements which the environment should meet. In the case of world objects this is a listing of the behaviours a world object should support. For instance, in figure 6.6 the window pane can be

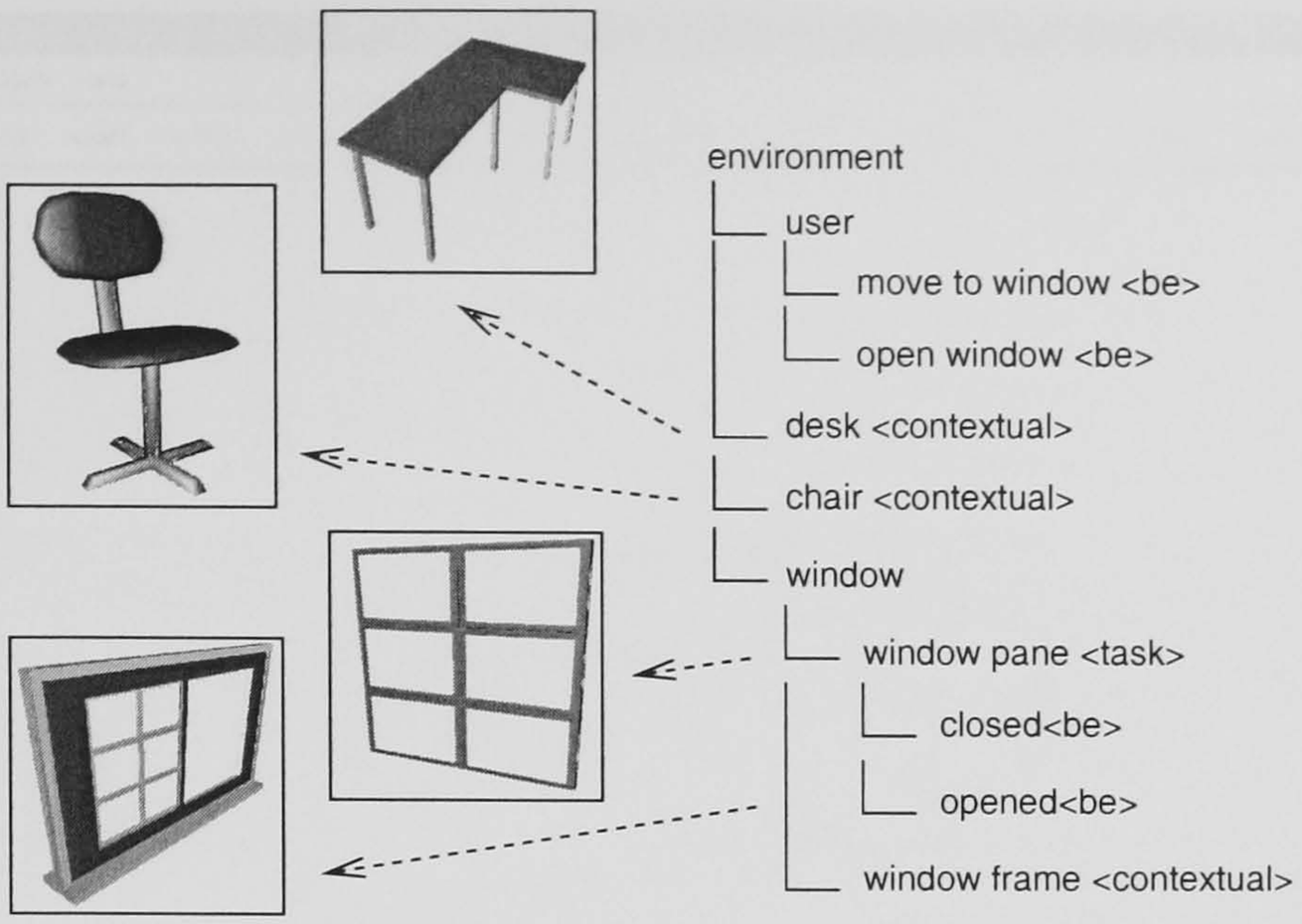

Figure 6.5: Interpreting the world object requirements from the requirements tree

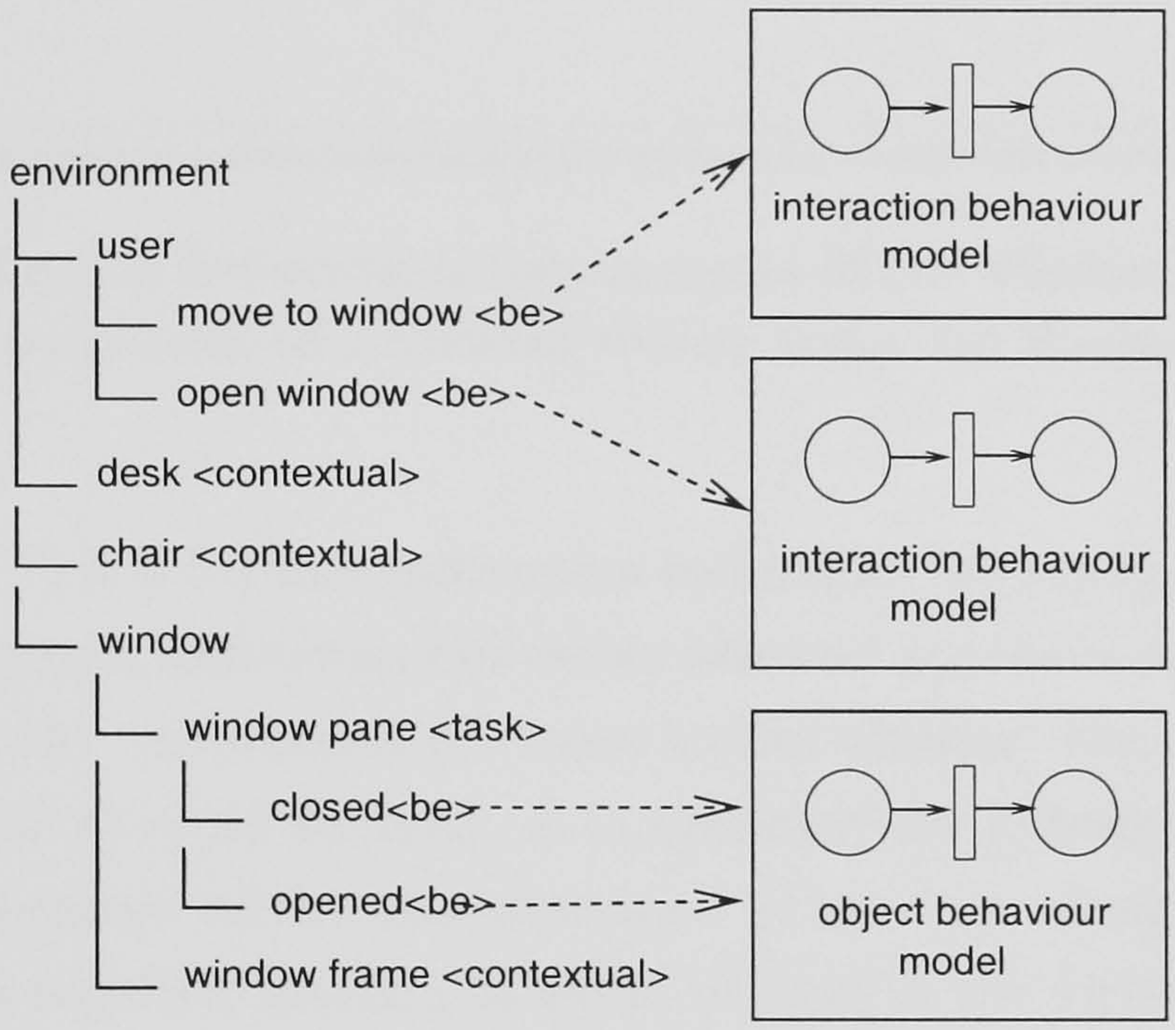

Figure 6.6: Interpreting the behavioural requirements from the requirements tree

opened and *closed*. These behavioural requirements can be mapped directly to a behavioural specification such as Flownets as exemplified in figure 6.7. With Flownets each of the world objects behaviours within the requirements tree becomes a discrete state in the Petri-net. This can be augmented with further detail to form a complete design.

A designer is clearly informed of the behavioural requirements that must be met by the designs of interaction techniques. These requirements can also be realised using behavioural designs such as Flownets. However, there is not the straightforward mapping between requirements of this type and their designs in the manner of world

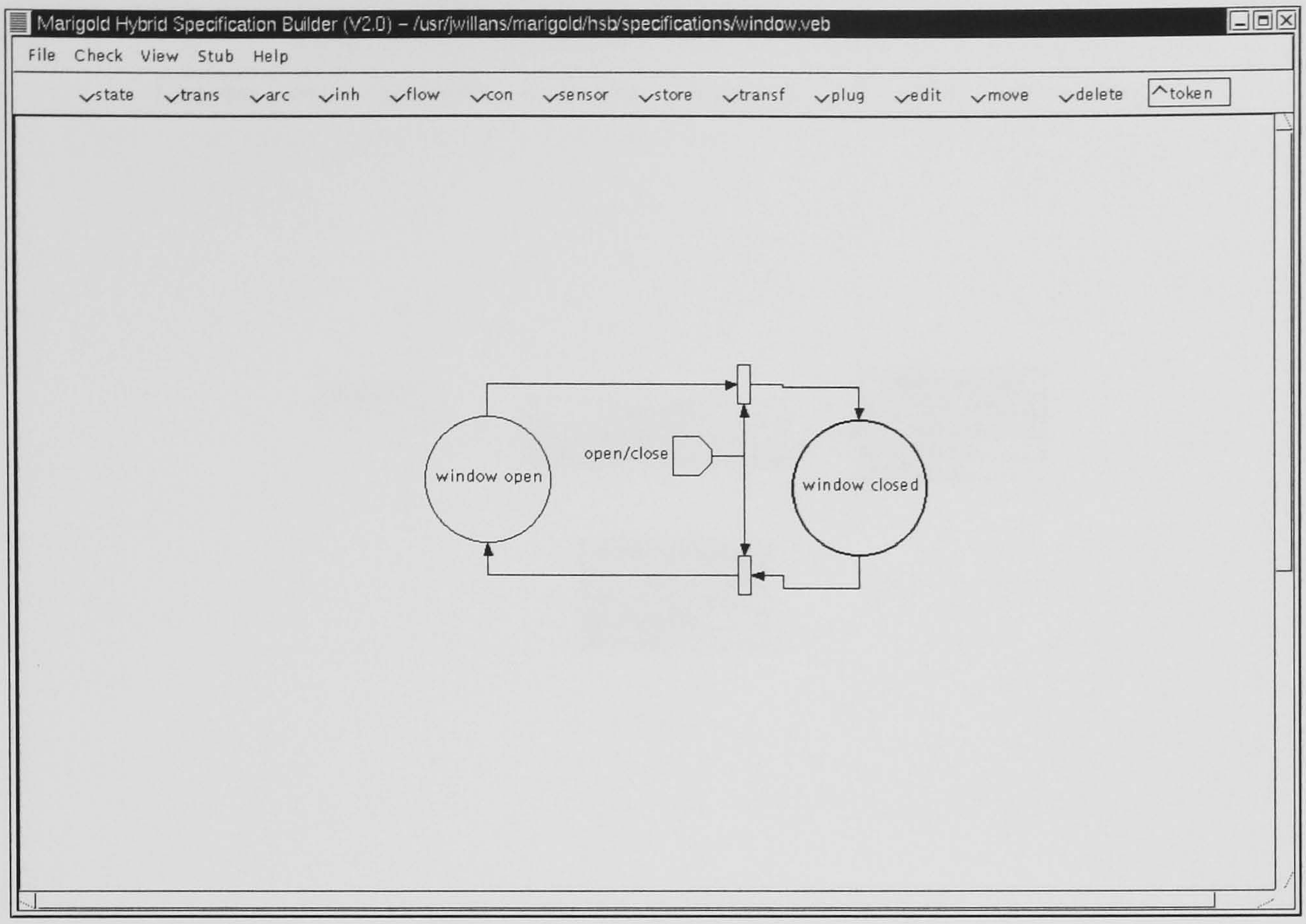

Figure 6.7: Mapping the behavioural requirements of the window pane world object onto the discrete component of a Flownet design using the Marigold HSB

object behaviour. This is because interaction techniques are rarely a direct simulation of real world interaction techniques but rather abstract approximations. For instance, in order to realise the requirement of "move to the window" the mouse-based flying interaction (chapter 4) could be used. It is necessary for a designer to consider the behavioural requirements of the user described in the tree, along with photographs (and maps) of the required location of world objects in the virtual environment, in order to design (or determine) appropriate techniques.

The grouping of world objects and their behavioural requirements in the requirements tree enables a designer to understand how requirements are related to each other. For example, the requirements tree of figure 6.6 shows that the requirements for a window *window* consists of two world objects and two behaviours. This can guide the construction of a Marigold COB specification which brings together designs of world objects and their behaviours (figure 6.8).

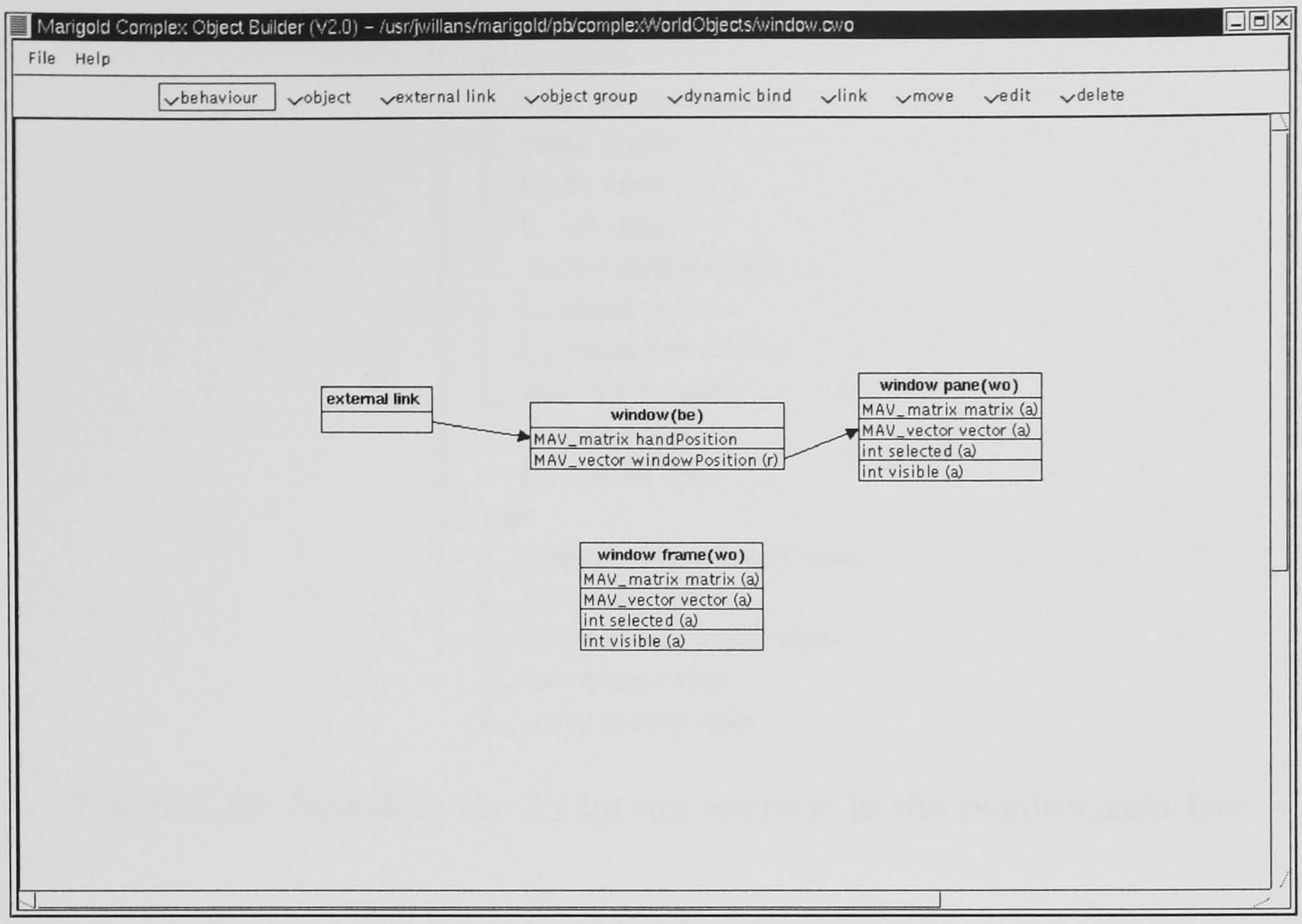

Figure 6.8: A Marigold COB specification for the opening window

6.4 Kitchen example

In this section we demonstrate the application of the requirements specification approach with a larger example. The problem statement addressed in this example is: "provide a virtual environment to train a chef to use typical kitchen equipment to cook a breakfast". For this problem statement we derive three scenarios dealing with the frying of an egg, the heating of beans and the making of toast.

The scenario for frying an egg using a gas oven (figure 6.9) results in the requirements tree (figure 6.10) containing an *oven* world object. This is decomposed into a *gas switch*, *ignition switch* and a *flame* to facilitate the behaviour associated with each of these world objects. In addition, there is an *oven unit* world object which supports no interaction. The *user* has associated behaviour which supports their interaction with the gas oven.

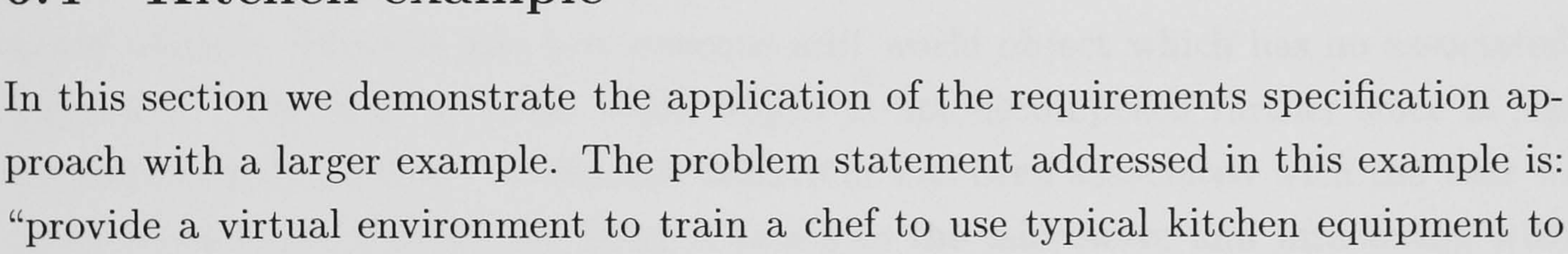

Figure 6.9: Using an oven to fry an egg scenario

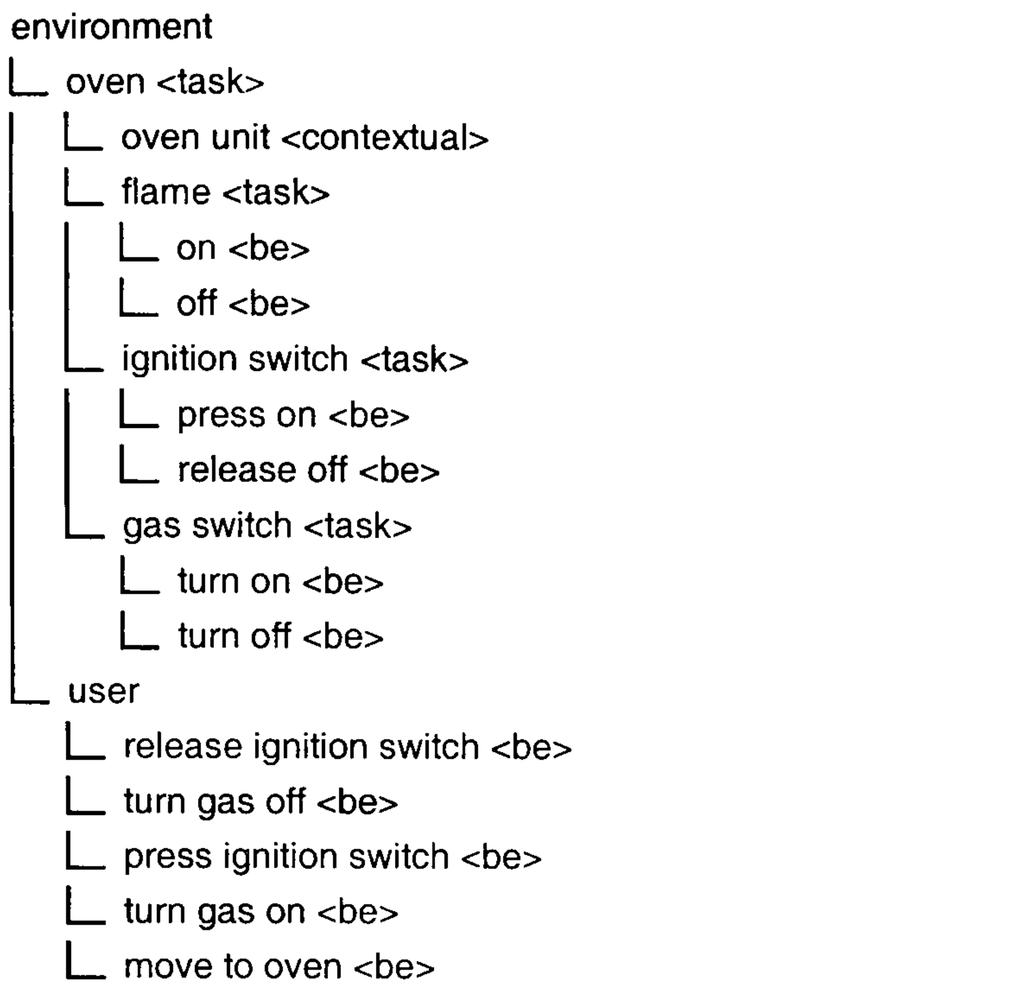

Figure 6.10: Including the frying egg scenario in the requirements tree

The scenario for cooking beans using a microwave (figure 6.11) has augmented the requirements tree (figure 6.12). Within this, the tree includes two additional world objects the *microwave* and the *bowl of beans*. The *microwave* is decomposed into an *on switch*, *timer* and a *door* to facilitate the behaviour associated with each of these world objects. There is also a *microwave unit* world object which has no associated behaviour. The *bowl of beans* world object is not decomposed further since it has no associated behaviour. Additional behaviour has been associated with the *user* in order to facilitate moving the bowl of beans to the microwave and interacting with the microwave in order to heat the beans.

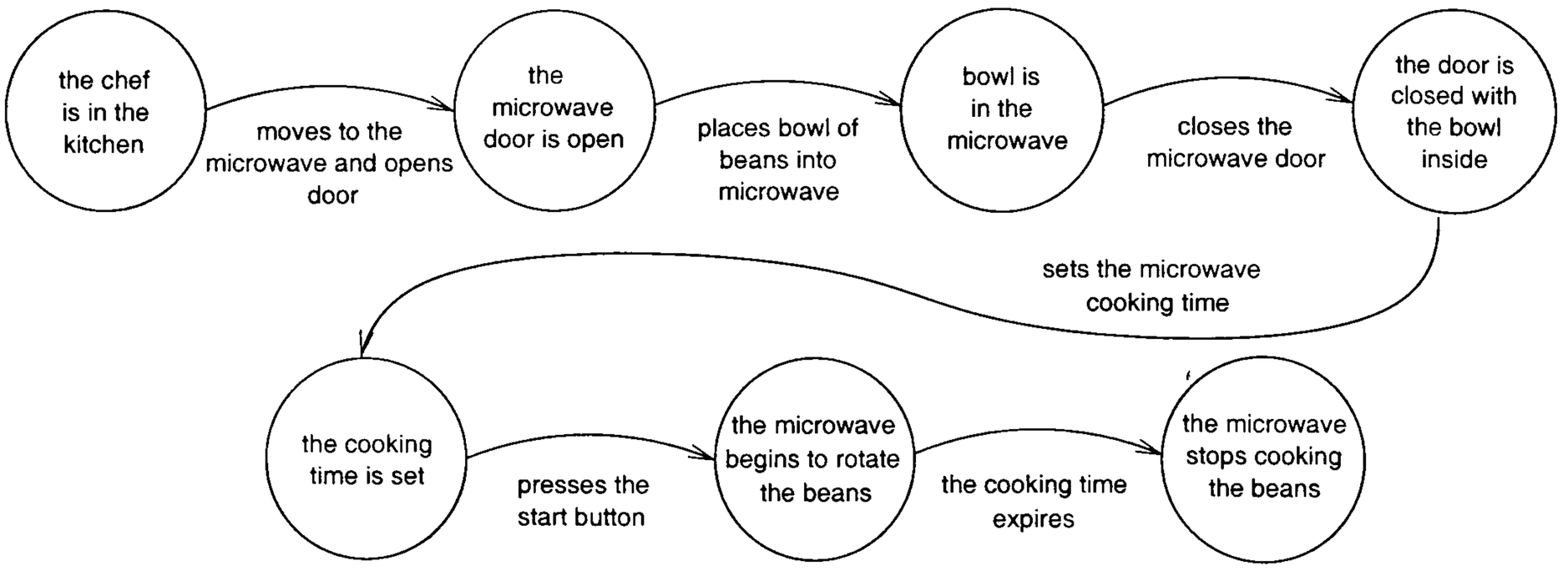

Figure 6.11: Using the microwave to heat beans scenario

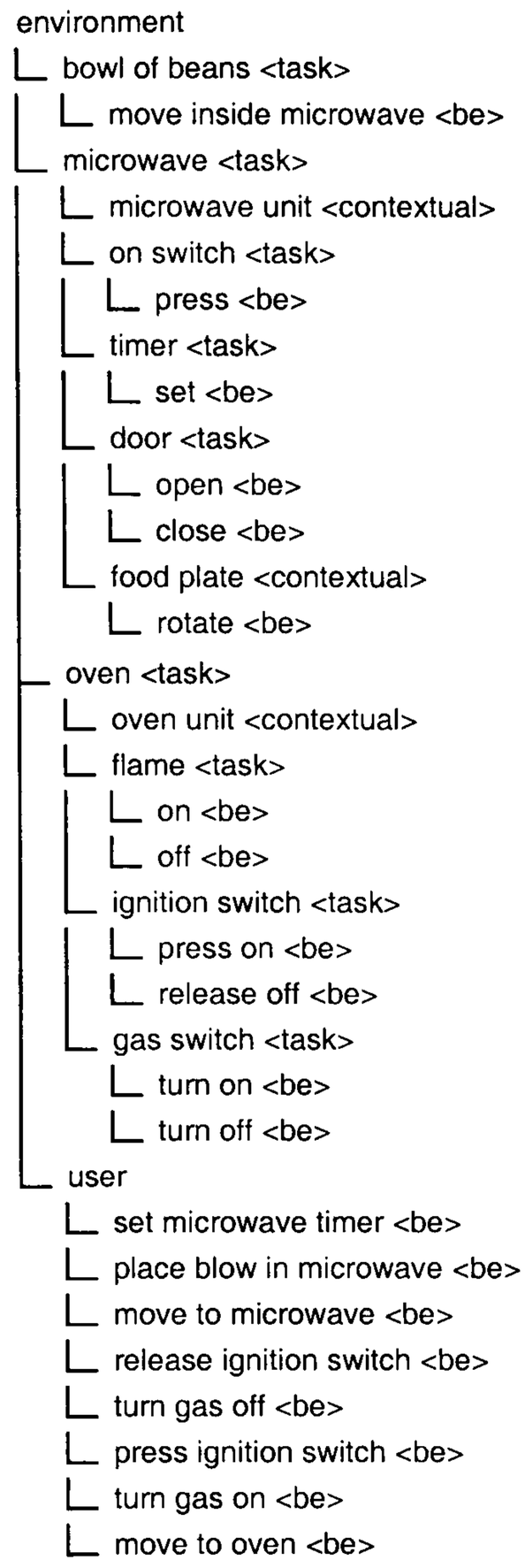

Figure 6.12: Including the microwave scenario in the requirements tree

The final scenario is making toast using a toaster (figure 6.13). The requirements tree (figure 6.14) has been further extended to include two additional world objects the *toast rack* and the *toaster*. The *toast rack* is decomposed into the *rack* itself and the *toast* that is initially held within the *rack*. The *toast* has an associated behaviour so that it can be moved from the *rack* to the toaster. The *toaster* object is decomposed into the *toaster* unit and the *slider*, which has an associated behaviour. Additional behaviour has been attributed to the *user* in order to facilitate the transfer of the *toast* from the *rack* to the *toaster* and the interaction with the *toaster* itself.

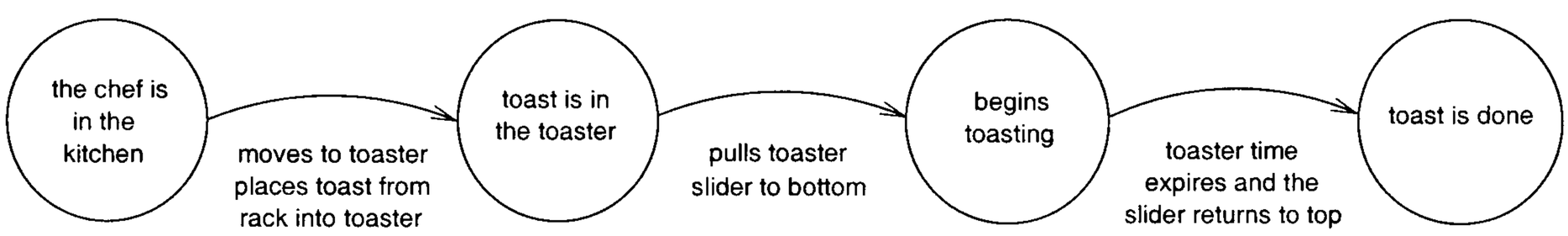

Figure 6.13: Using the toaster to make toast scenario

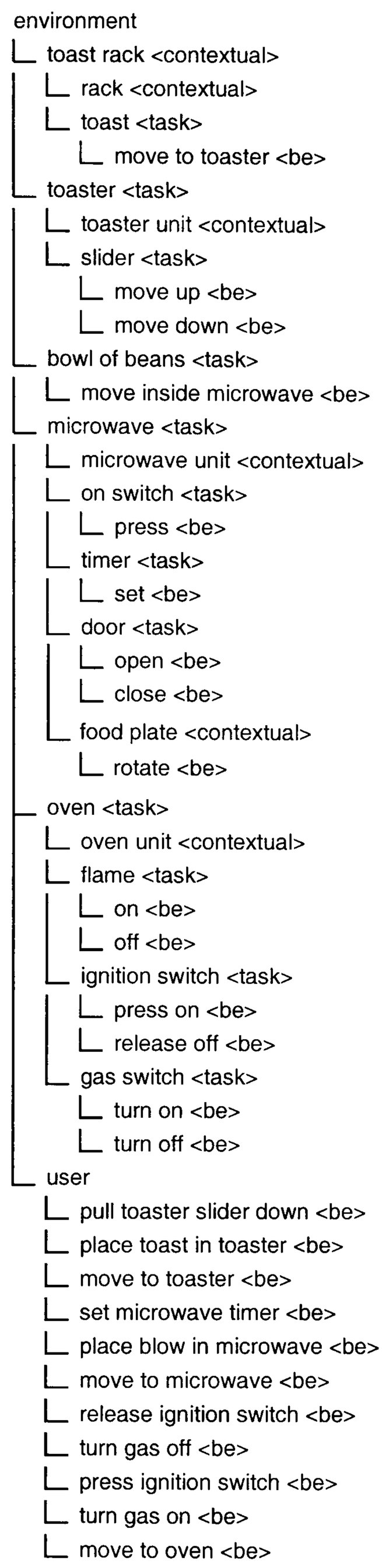

Figure 6.14: Including the toast scenario in the requirements tree

6.5 Primrose

Applying the described approach by hand can be tedious and error-prone. Rarely do scenario descriptions fit inside a state or on a transition and hence a specification may have to be compromised to accommodate this restriction, reducing the accuracy of the requirements. The requirements tree must be redrawn for almost every step of the analysis with the danger that a designer may shortcut analysis of the scenarios and attempt to consider multiple states/transitions concurrently.

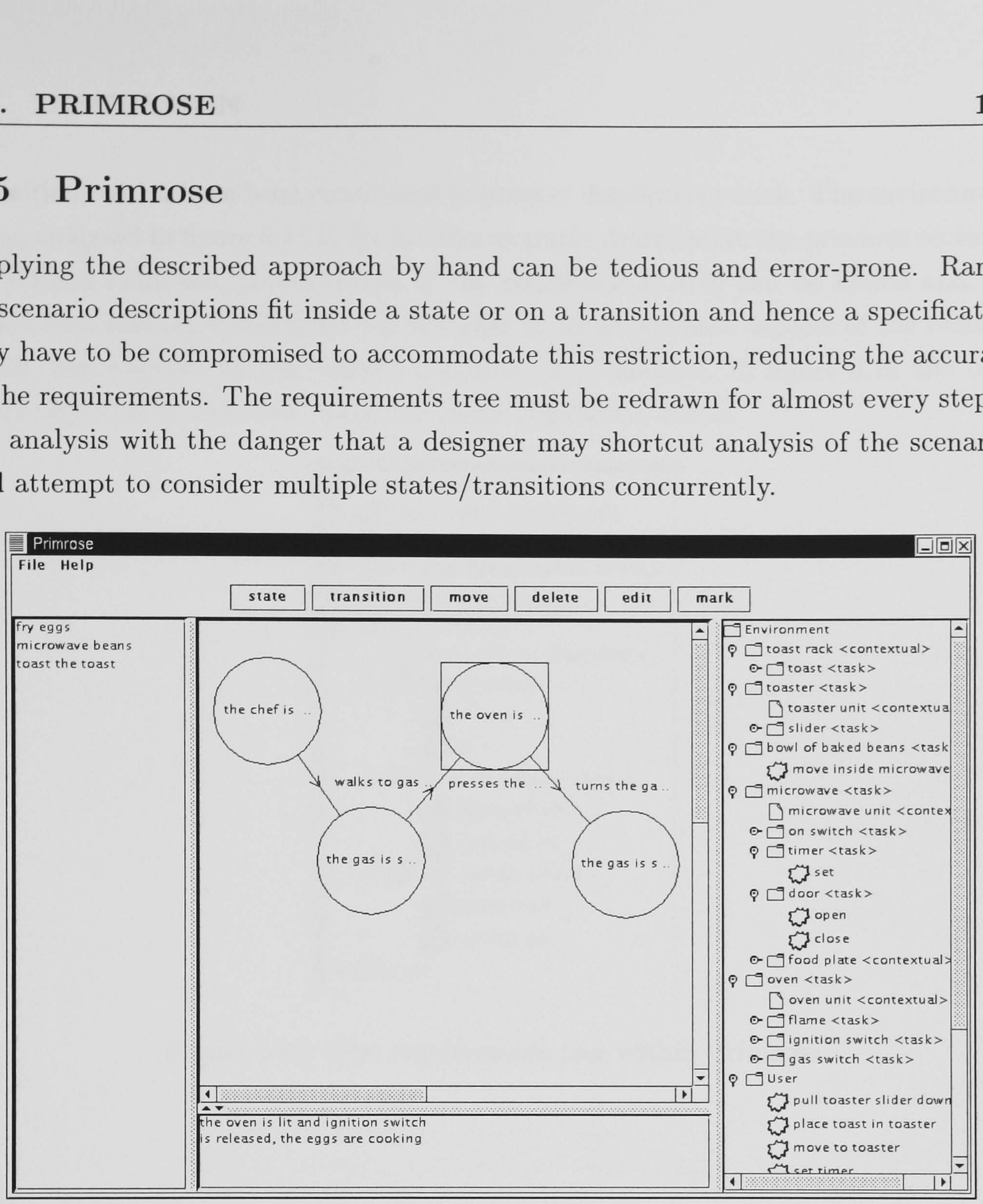

Figure 6.15: A screenshot of the Primrose tool

The Primrose tool is designed to support the requirements specification approach. It is written using Java/Swing and has been tested for portability across unix and windows based platforms. A screenshot of this tool is shown in figure 6.15. At the left side of the tool is a list of the scenarios for the environment being designed, new scenarios can be added to this dynamically. In the centre of the tool is an area for creating the state transition description of the selected scenario (using point and click). The full descriptions of states and transitions are displayed in the text area below the diagram allowing long comprehensive descriptions of each part of the scenario. At the right hand side of the tool is the requirements tree which can be dynamically updated and revised during the analysis of scenarios. Within this, behavioural nodes are described using the icon rather than the <be> tag of the previous section. During scenario analysis it is possible to mark those states and

transitions which have been considered to prevent duplicating work. The environment being analysed in figure 6.15 is the kitchen example described in the previous sections.

Within Primrose, sub-branches of the requirements tree can be folded and unfolded thus making it easier for the designer to focus on those aspects of the requirements tree relevant to the current concerns. For instance, in figure 6.16 the *oven* world object of is unfolded, while the other objects are folded.

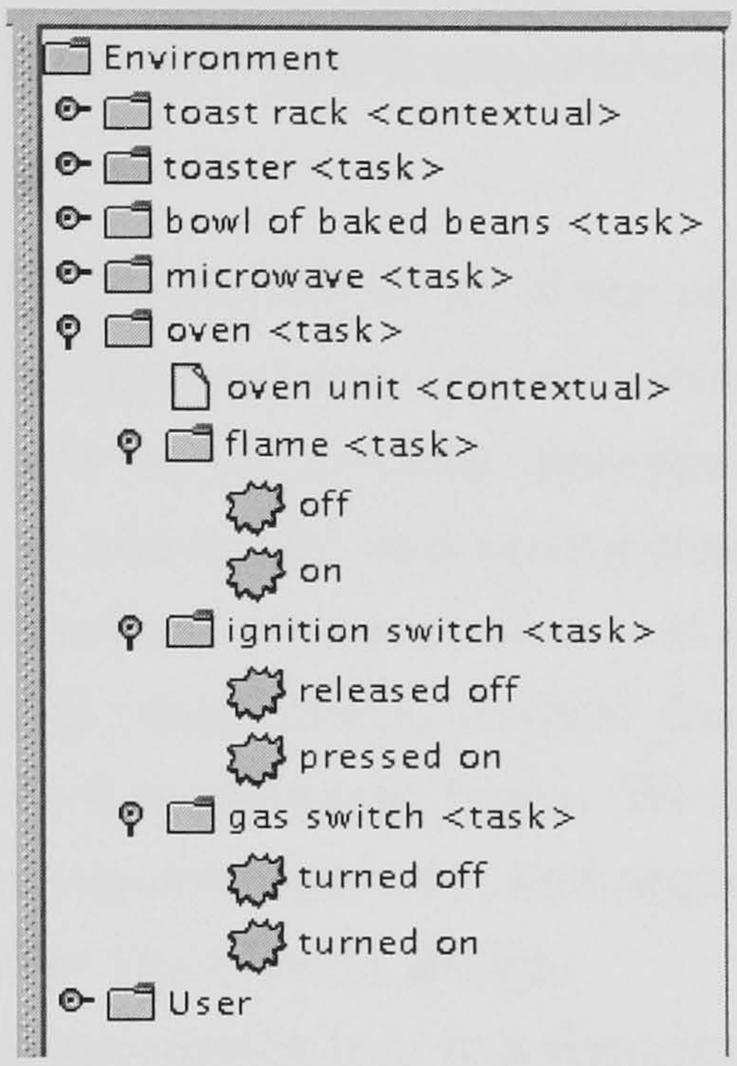

Figure 6.16: The requirements tree within Primrose

6.6 Discussion

The previous sections have introduced an approach for the requirements specification of virtual environments. In the introduction to this chapter we expressed two desirable criteria for such an approach:

- An approach should elicit the requirements in a language natural to the users.

We have used scenarios to elicit the requirements from the users, these are structured using state transition diagrams. Scenarios have long been recognised as a good method of recording existing work practices which are easy for users to adopt [Ben, Tawbi, and Souveyet 1999]. This is because the story-telling style of scenarios places few constraints on how they should be constructed. In the presented approach the user is describing interaction with the familiar domain of the real world. As such the scenarios have the additional desirable characteristic of abstracting from implementation factors which may hinder the elicitation.

A criticism that can be applied to scenarios is that they do not necessarily expose all the requirements. Indeed this criticism can be applied to most requirements specification approaches since there is no guarantee that the specified requirements are indicative of the real requirements. In the previous chapters we have pursued an approach to ensuring designs are correct using the Marigold toolset. This allows any missed requirements to be identified and integrated into the design.

- An approach should refine the elicited requirements into a language natural to the designer.

Although scenarios document the requirements of the virtual environment, these requirements are clouded by a clutter of non-relevant detail (for the designer) as well as being spread across multiple representations. Interpreting the implications of scenarios directly to a design is non-trivial and error-prone. We have introduced the requirements tree as an intermediate representation that addresses this problem. The requirements tree consolidates requirements derived from scenarios and structures them according to a number of requirement types. We have demonstrated how scenarios can be mapped to the requirements tree, and argued that the coherency of the requirements tree makes easier their use in design.

An omission from the requirements tree is a description of how the state of the environment influences what behaviour can take place. For instance, in the "fry egg" scenario, a gas and a spark is required as a precondition for a flame to appear. These dependencies can be used as the basis for test cases for behavioural analysis (chapter 5). Early experimentation of the approach did attempt to incorporate this information by including multiple representations of the tree which were associated with scenario states. Therefore, each state would have a tree describing the precondition and a tree describing the post condition. It was found, however, that even when supported by the Primrose tool that this was cumbersome. Having multiple representations of the tree was confusing and difficult to consolidate to a single specification for a designer. More significantly, we found that the existing approach made it possible to determine the behavioural dependencies by considering the real world object using the level of realism defined within the requirements tree.

6.7 Conclusion

In this chapter we have explored requirement specifications for virtual environments which are based on the real world. The presented approach begins with a problem statement expressing the overall aim of the environment. This statement is refined to a series of scenarios which allow the user to describe episodes that the virtual

environment should support in terms of concrete real world interaction. The scenarios are then analysed and key requirements are structured in a manner that allows them to be easily mapped to a virtual environment design.

In order for an approach to be valuable, practical concerns relating to its usage must be considered. The Primrose tool was developed to support the application of the approach.

Chapter 7

Case studies

In the previous chapters we have described the Marigold toolset which supports the exploration of virtual environment behavioural designs, and an approach supported by the Primrose tool which enables the specification of virtual environment requirements. In this chapter we present two case studies which illustrate the application of these contributions.

7.1 Introduction

This thesis has made a number of steps towards *integrating behavioural designs into the development process of virtual environments.* In chapter 2 we argued that Flownets are a good formalism for the design of virtual environment behaviour. In chapter 4 we presented an approach to evaluating the behavioural designs using prototypes built using data flow networks. In chapter 5 we presented an approach for the evaluation of Flownet designs using automatic analysis. Both these approaches are supported by the Marigold toolset. In chapter 6 we presented an approach to requirements specification prior to the design of the behaviour (and world objects). This approach is supported by the Primrose tool.

This chapter contains two case studies that demonstrate the application of Marigold and Primrose. Our aim is:

- To illustrate how the evaluation approaches supported by the Marigold toolset behave in a realistic development context.

- To illustrate the extent to which the additional use of the approach supported by Primrose guides the formulation of designs.

7.2 Navigating a landscape

In this case study we design an environment which supports the navigating of a user around a large natural landscape. During navigation the user should be able to observe visual features distributed around the landscape. The environment will eventually be placed in a public museum where we can assume that users will be from a variety of backgrounds. This is a good case study for assessing the application of the approach because:

- Effective navigation is frequently critical to a well designed virtual environment, yet rather than being designed, reusable toolkit interaction techniques are often employed.
- The case study is a realistic one since museums and similar fora are constantly seeking methods of broadening access to information.

7.2.1 Initial design

A first step in being more concrete about what the requirements of the environment are, is to evaluate an existing technique to see how well it works in practice. Figure 7.1 shows a prototype specification which uses the mouse-based flying interaction technique (discussed earlier in the thesis) to navigate the landscape. This is a sensible initial choice because it requires a commonly available hardware configuration.

When we evaluated the prototype generated from this specification, desirable and undesirable characteristics of this technique emerged. It was advantageous that the technique can control speed in addition to orientation, because this provides a means of slowing down to accurately locate interesting landscape features and quickly pass those of little interest. However, because mouse-based flying does not control navigation on the y axis there was no way to descend to observe more detailed landscape because the user is at a static height. When this static height was lowered, we collided with the landscape and quickly became disoriented. Therefore, a technique is required which retains the ability to control speed but provides additional navigation on the y axis.

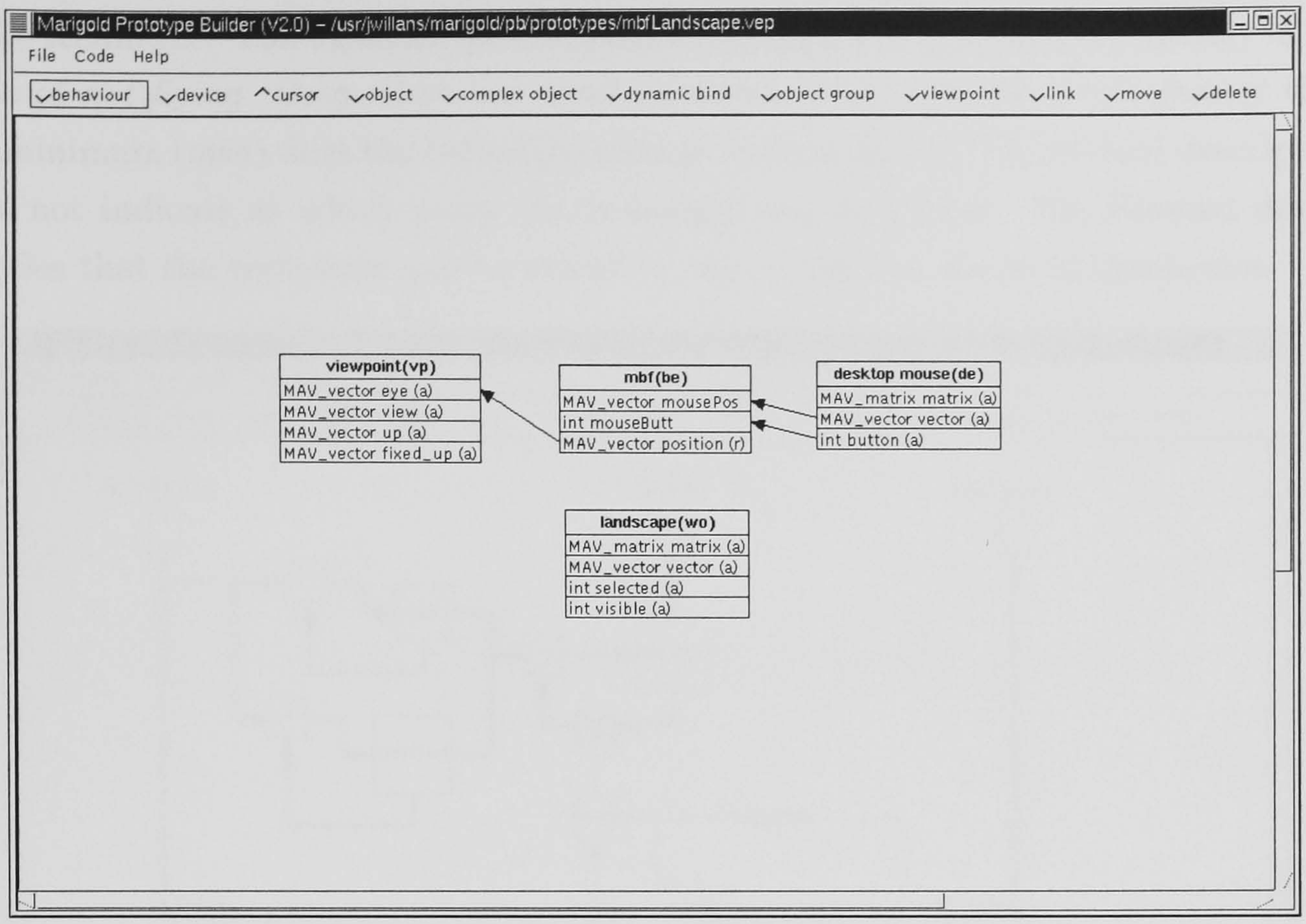

Figure 7.1: A prototype specification using mouse-based flying technique to navigate the landscape

7.2.2 Two-handed flying

In [Mine, Brooks Jr, and Sequin 1997], Mine et al. present a series of techniques which utilise the proprioceptive senses of the users body position to promote a better understanding, for the user, of their state of interaction. One of these techniques is two-handed flying where the user controls the direction and speed of navigation using their arms. The following description of the technique is taken from [Mine, Brooks Jr, and Sequin 1997]:

> The direction of flight is specified by the vector between the user's two hands, and the speed is proportional to the user's hand separation. A dead zone (some minimum hand separation e.g. 0.1 metres) enables users to stop their current motion quickly by bringing their hands together (a quick and easy gesture).

Since the speed and direction of navigation is controlled by the users hands, this technique provides control of the speed and also navigation on the z axis. Our next step was to move from this informal explanation of the technique to a reified Flownet design. The resulting design specification for (an interpretation of) this technique is shown in figure 7.2. This clarifies the technique and the additional design decisions we have made. For instance, the textual description gives no indication of the initial state

of the technique. The Flownet specification states that the technique is initially in the state of *not flying*. Only when the *hand positions* reach a distance (d) greater than the minimum (min) does the technique change state to *flying*. The textual description does not indicate at which point the technique can be exited. The Flownet design clarifies that the technique can be exited in any of the two states of interaction.

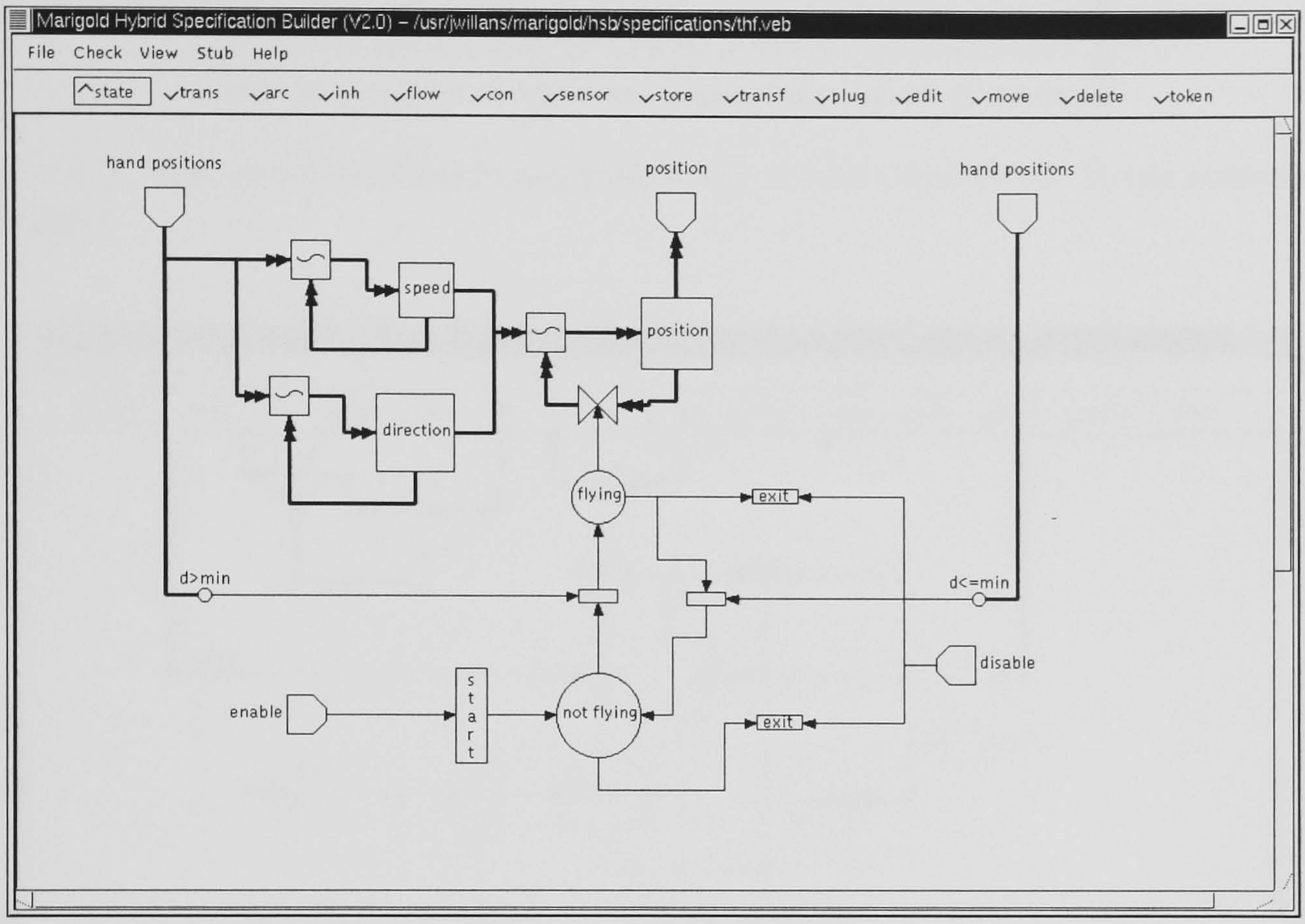

Figure 7.2: Flownet specification for the two-handed flying technique

7.2.3 Mode confusion analysis

Once the Flownet specification for the two-handed flying technique was built and we were satisfied it matched the textual description, we checked the design for mode confusion using the HSB. The result of this analysis confirmed that there was a potential mode confusion problem. This is illustrated in figure 7.3.

When the technique is in the *not flying* state, no change is rendered to the external environment. The user is unable to perceive from the state of the environment whether or not the technique is active when stationary. To overcome this problem the technique was revised, this is shown in figure 7.4. This incorporates additional constructs so that whenever the technique becomes active or inactive, an appropriate notification is passed to the outside environment (the output to the *thf* plug). The revised specification showed no symptoms of mode confusion.

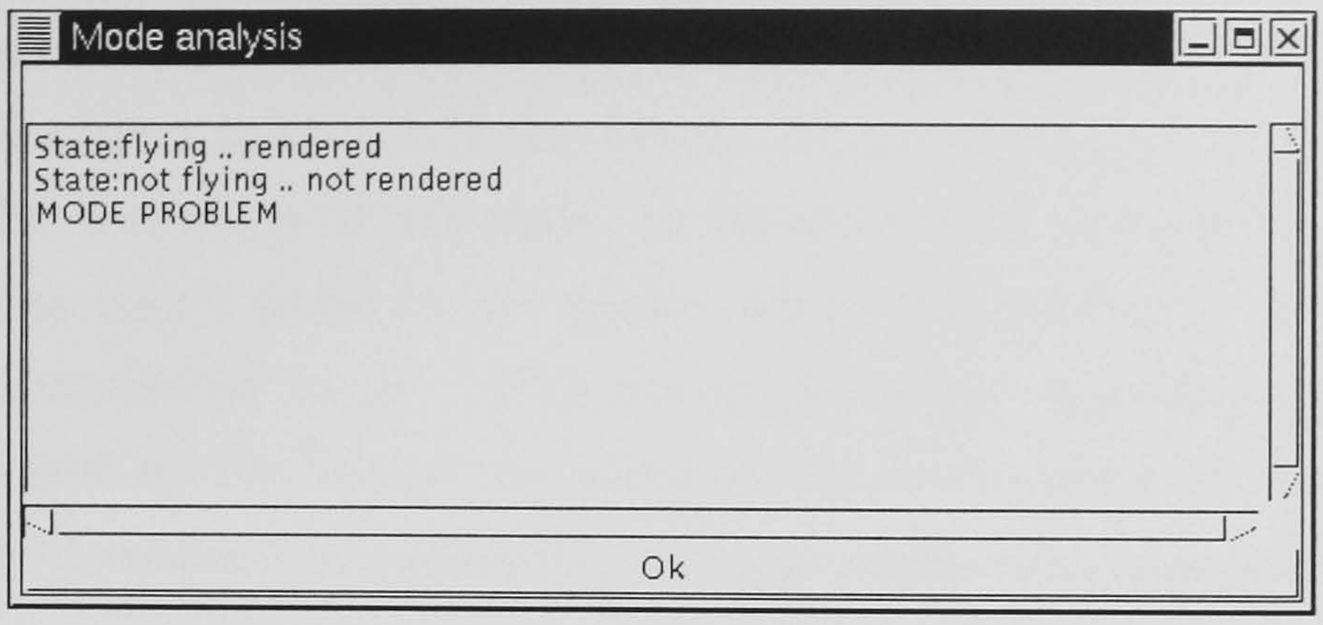

Figure 7.3: The mode confusion analysis result of the two-handed flying interaction technique

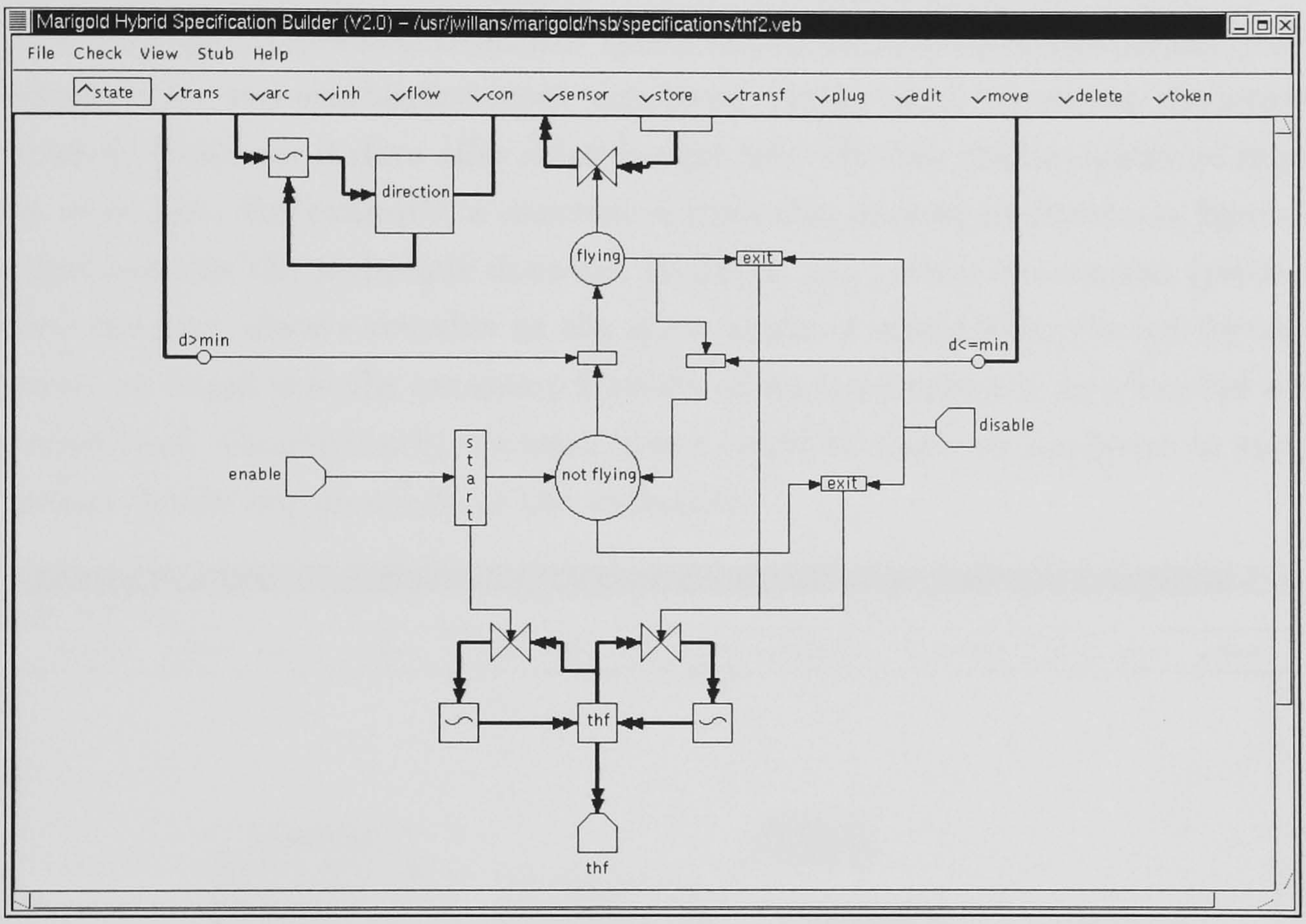

Figure 7.4: Revised Flownet specification for the two-handed flying technique addressing potential mode confusion

7.2.4 Prototyping the design

The next stage was to add code to the nodes constituting design specification of the interaction techniques and to generate a stub which can be inserted into the prototype specification for navigating the landscape. The resulting prototype specification is shown in figure 7.5 using the two-handed flying technique to control navigation. The technique controls the visibility of the *active* world object (which is initially invisible) by changing the state of its *visible* variable. The position of the viewpoint is updated

by the technique as is the position of the *active* world object. This is so that when the *active* world object is visible it can always be perceived within the viewpoint.

Although the technique is intended to be used with trackers to determine the positioning of the user's hand in 3D space, when this prototype was first specified trackers were unavailable to us. This initial prototype specification simulates the hand trackers input by the use of the desktop mouse to represent the left hand, and a mapping of the keyboard to represent the right hand. The position of these devices were also mapped onto cursors so that their relative distance could be perceived in the viewpoint (this would not be necessary with hand trackers because the user has a sense of the relative position of their hands). A further mapping of the keyboard enabled and disabled the technique.

The prototype generated from this specification did not allow the usability of the technique to be well established since simulated, rather than the actual, devices were being used. However, it does offer some insight into whether the navigational requirements were met. For example, a conclusion from this prototype (shown in figure 7.6) was that because the technique does not facilitate any rotary locomotion (pitch, roll or yaw) the user always remains at the same angle of orientation (facing forwards). However, we found that the necessary features of the landscape can be observed within this constraint. Consequently, a commitment could be made to hardware to support the actual device requirements of the technique.

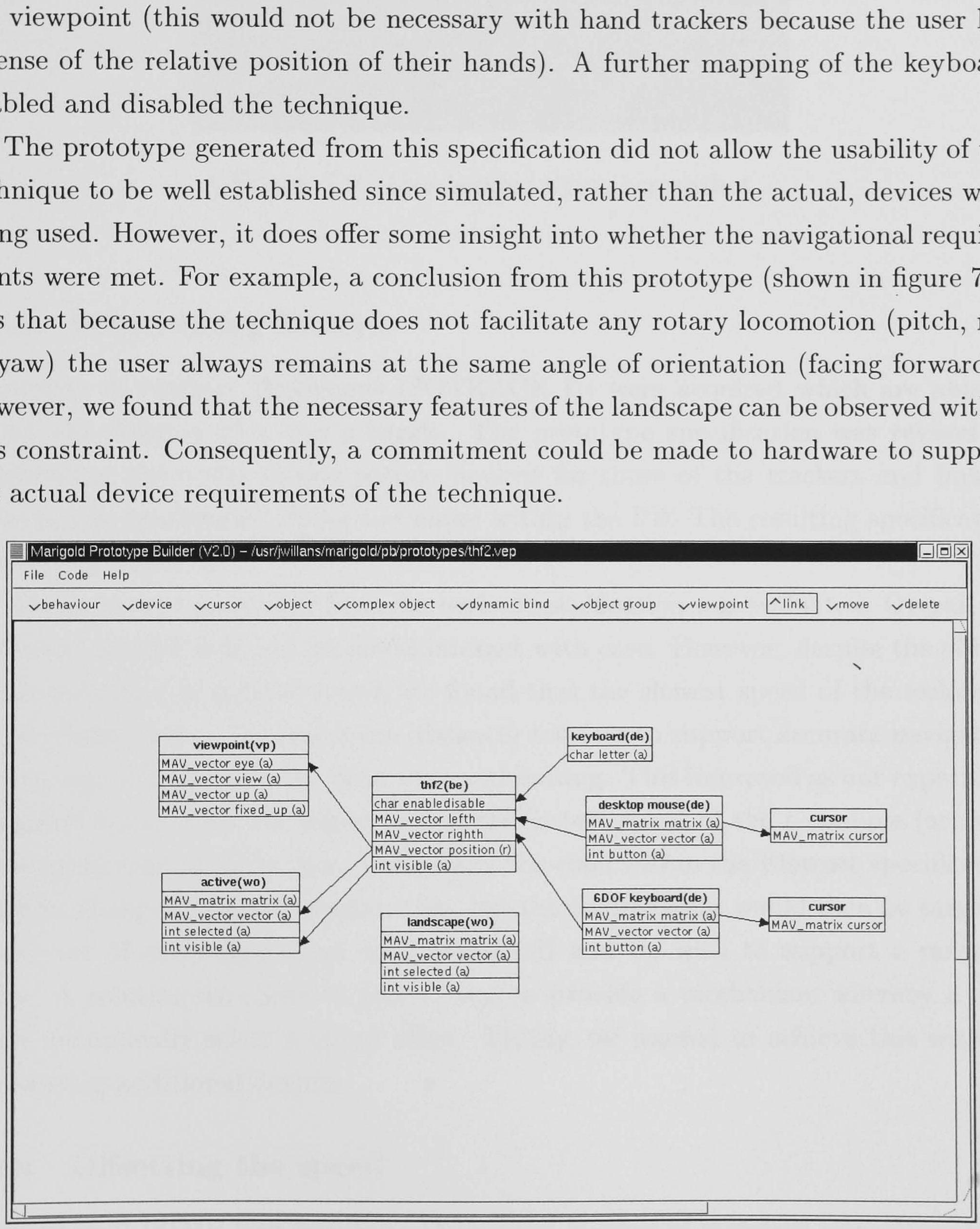

Figure 7.5: Prototype specification with an indicator to avoid mode confusion

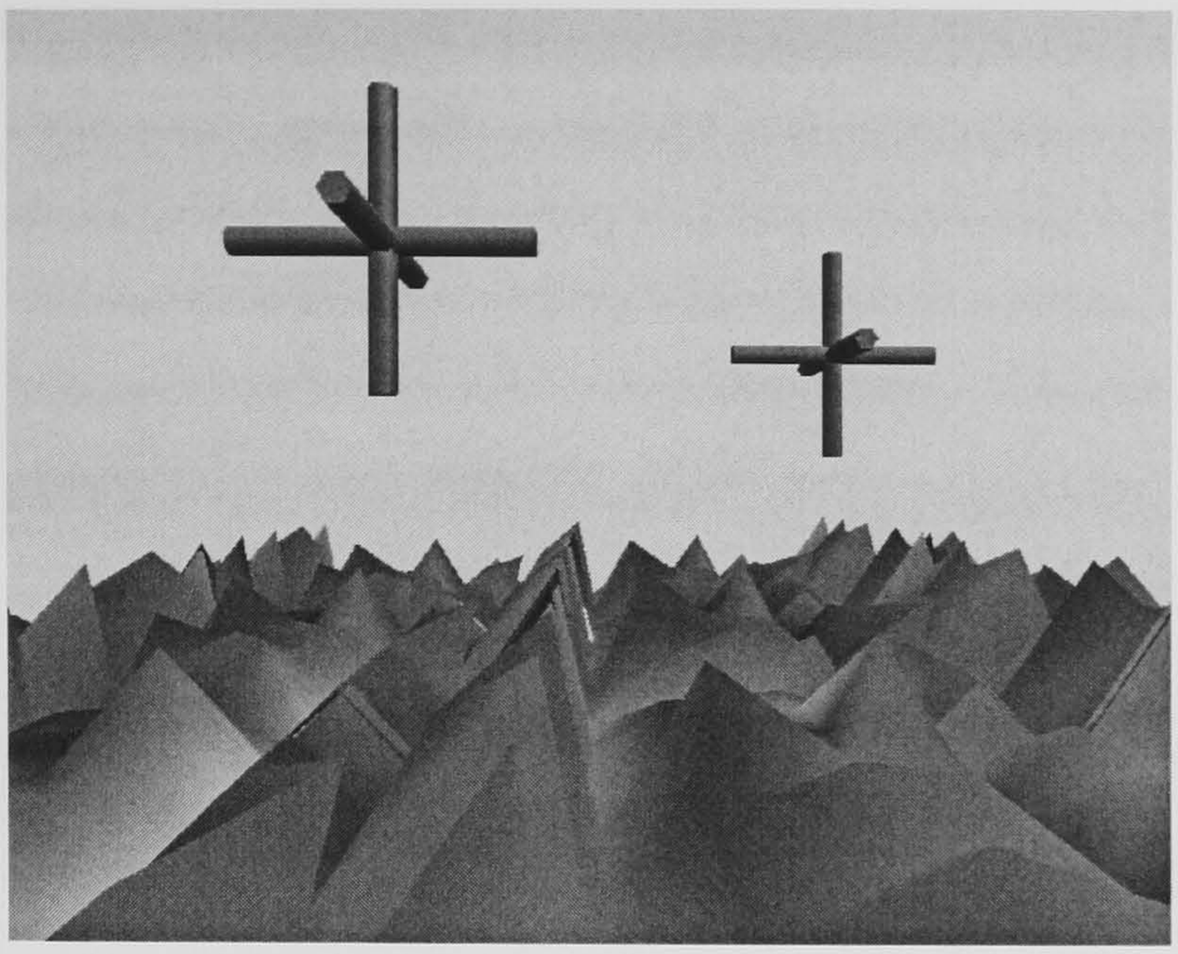

Figure 7.6: Two-handed flying screenshot

7.2.5 Substituting devices

A number of trackers (Polhemus ISOTRACK II) were acquired which are able to detect the position of a user's hands. The prototype specification was revised by substituting the nodes of the pseudo devices for those of the trackers and linking these to the two-handed flying technique within the PB. The resulting specification is shown in figure 7.7.

The prototype generated from the revised specification was evaluated. Overall the technique worked well and we could interact with ease. However, despite the ability of the technique to control speed, we found that the slowest speed of the technique (hand slightly above the minimum distance) too fast to support accurate navigation and we kept slipping into the dead zone and halting. This improved as our experience increased, but quickly we began to find the fastest speed of the technique (arms at full stretch) frustratingly slow. Potentially the code within the Flownet specification could be changed to accommodate this, but the environment would then be targeted to one set of users (amateurs or experienced) and we wish to support a range of users. A solution we chose to pursue was to provide a mechanism whereby a user could dynamically select a speed offset. Ideally, we wanted to achieve this without introducing additional devices.

7.2.6 Offsetting the speed

In traditional WIMP user interfaces there are a number of standard interaction techniques that allow the user to select one of a number of options such as pull-down menus and check-box. One equivalent interaction technique for virtual environments

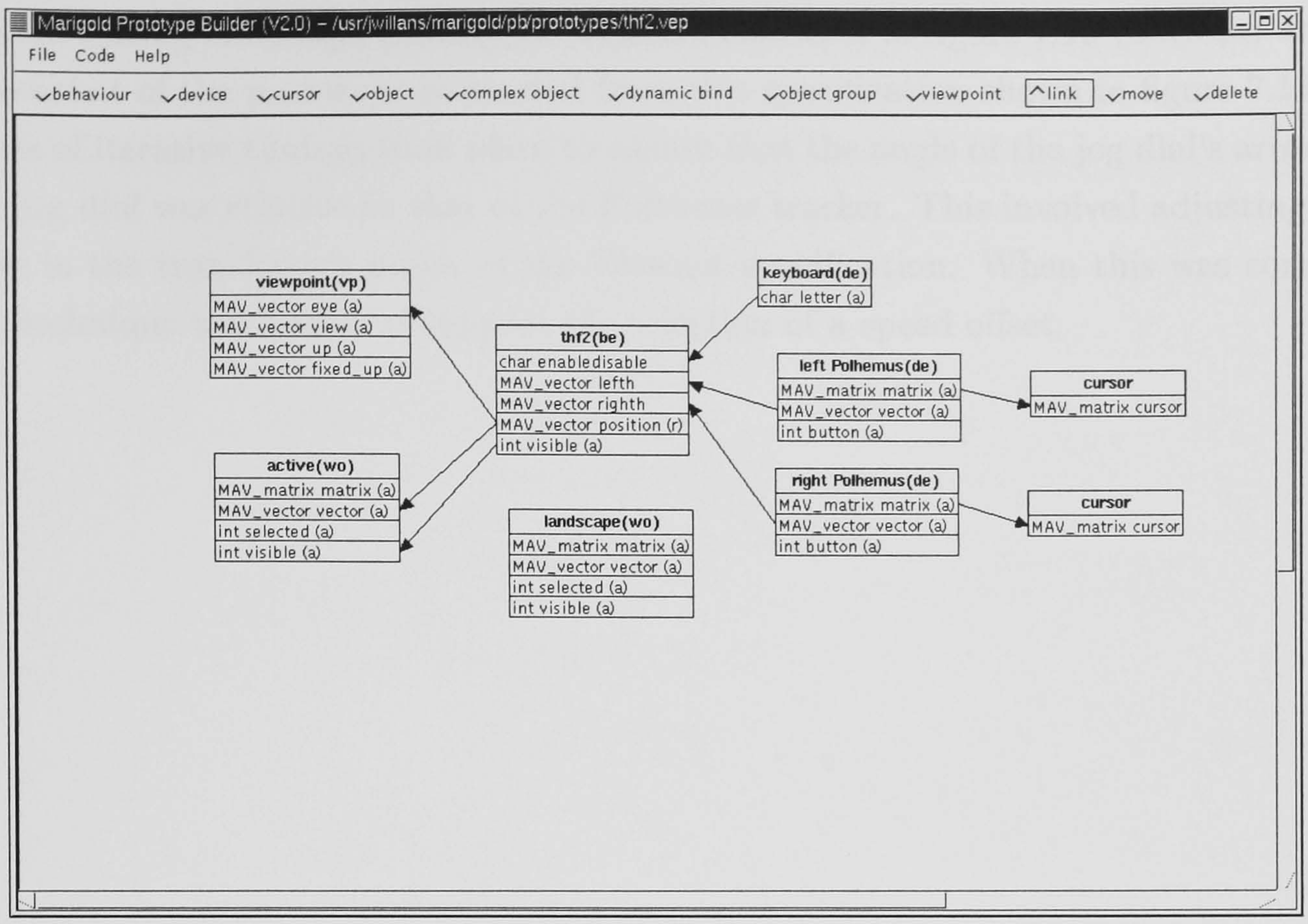

Figure 7.7: Prototype specification using Polhemus trackers

is the virtual jog dial introduced in [Deisinger, Blach, Wesche, Breining, and Simon 2000]. The following description of the technique is taken from [Deisinger, Blach, Wesche, Breining, and Simon 2000]:

> The jog dial for VR is a semi-circle and an indicator. The semi-circle represents the range of possible values and the indicator points to the current value. The indicator rotates around the centre of the semi-circle and is controlled intuitively by rotating the wrist around the axis defined by the forearm. The jog dial is activated through a speech command or menu item. When activated, a button on the device controls the jog dial. As soon as the button is pressed the indicator rotates according to the wrist, until the button is released.

Since the technique uses the rotary movement of the wrist to determine the selection, this technique can utilise one of the two Polhemus trackers which currently exists within the design.

The Flownet specification for this is shown in figure 7.8. Code was added to the nodes constituting the jog dial Flownet and a stub of the technique generated. This was made reusable by encapsulating the behaviour and renderings in a complex object specification shown in figure 7.9. Initially the stub generated from the COB specification was inserted into a new prototype specification for evaluation independent of

the navigating landscape prototype. This is illustrated in figure 7.10 (bottom) and a screenshot of the prototype generated from this specification shown in figure 7.11. A series of iterative tunings took place to ensure that the angle of the jog dial's arrow of the jog dial was relative to that of the Polhemus tracker. This involved adjusting the code in the transformer nodes of the Flownet specification. When this was correct, the technique was found to support the selection of a speed offset.

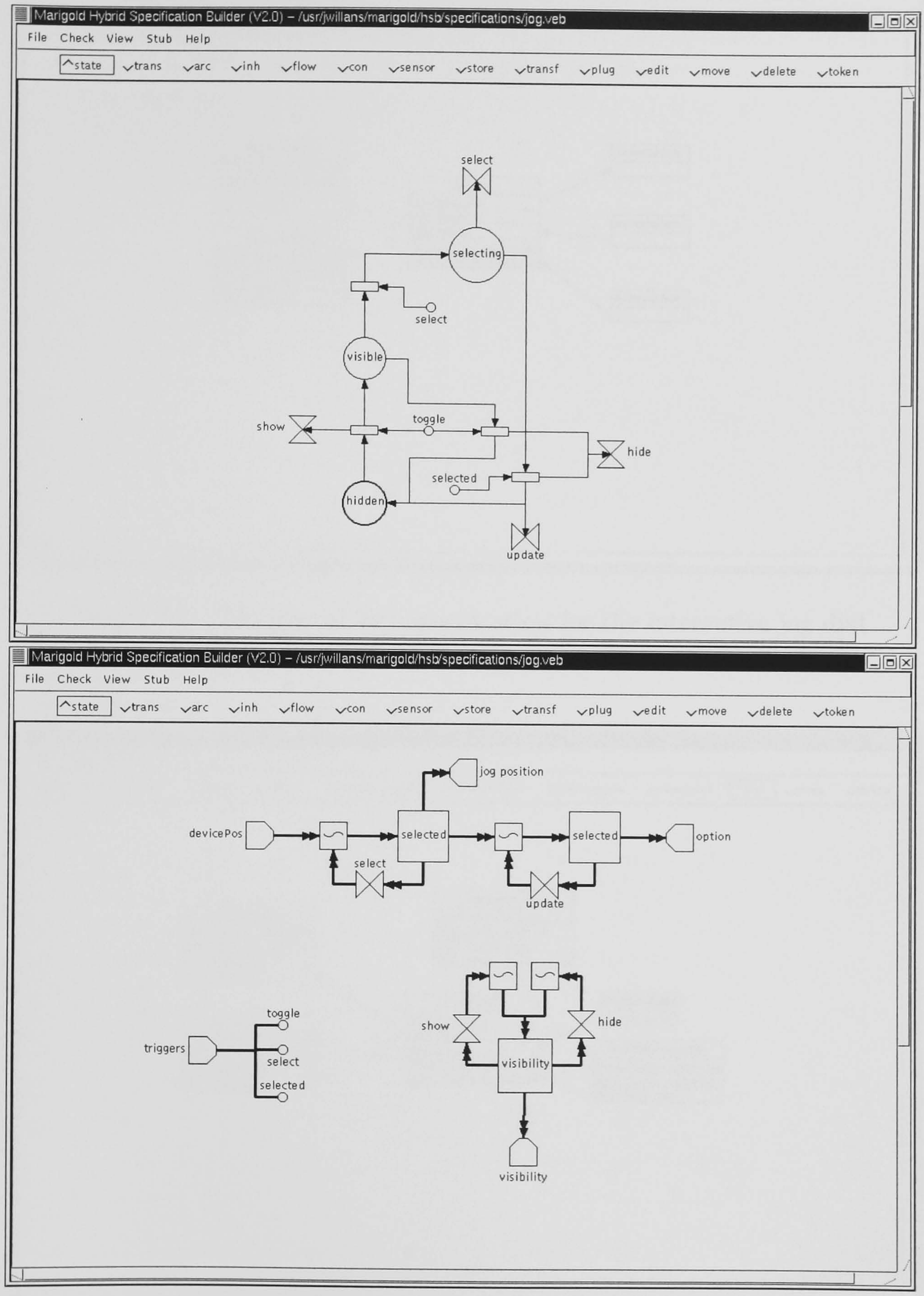

Figure 7.8: Revised Flownet specification for the interactive jog dial

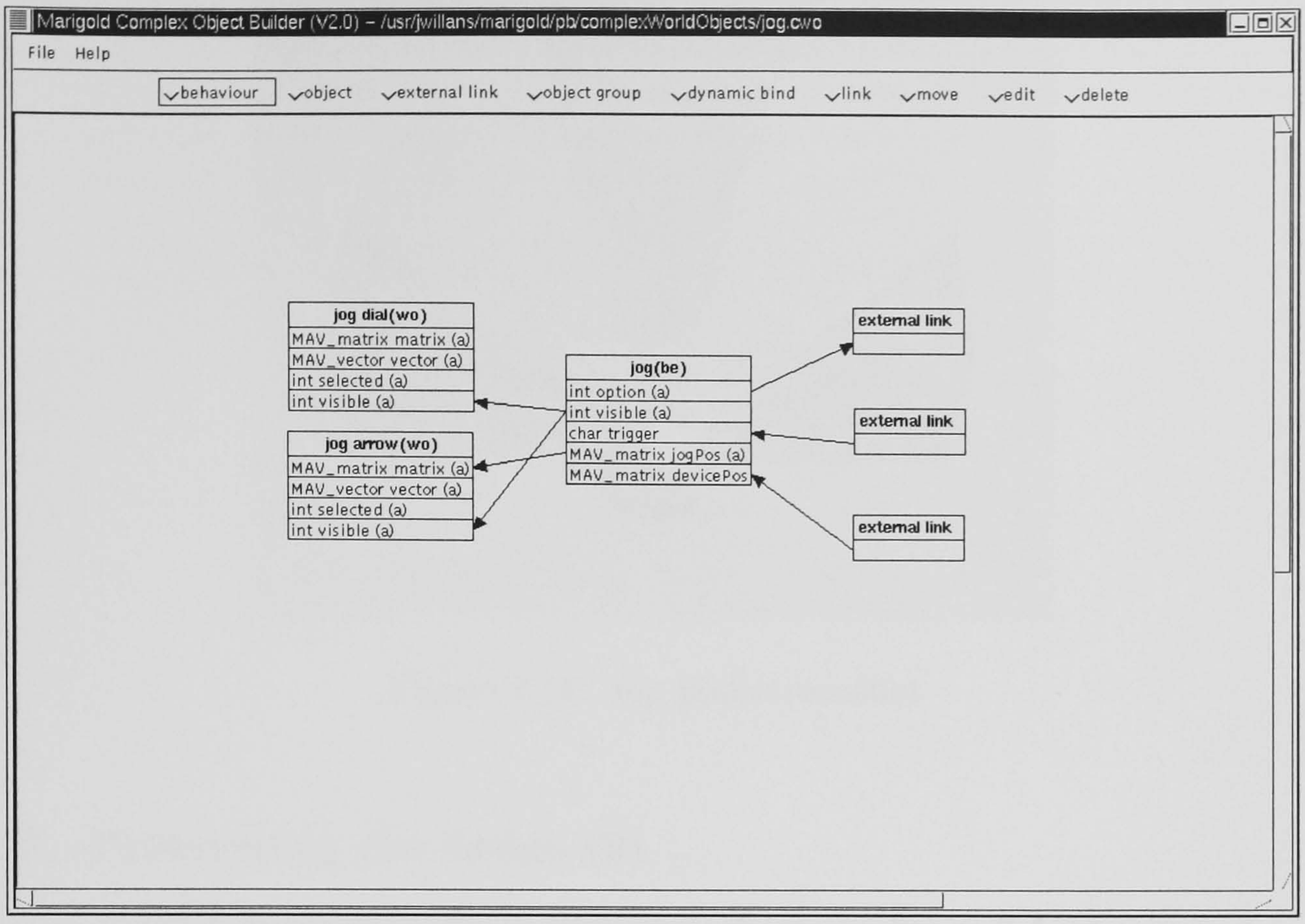

Figure 7.9: Complex object specification for the interactive jog dial

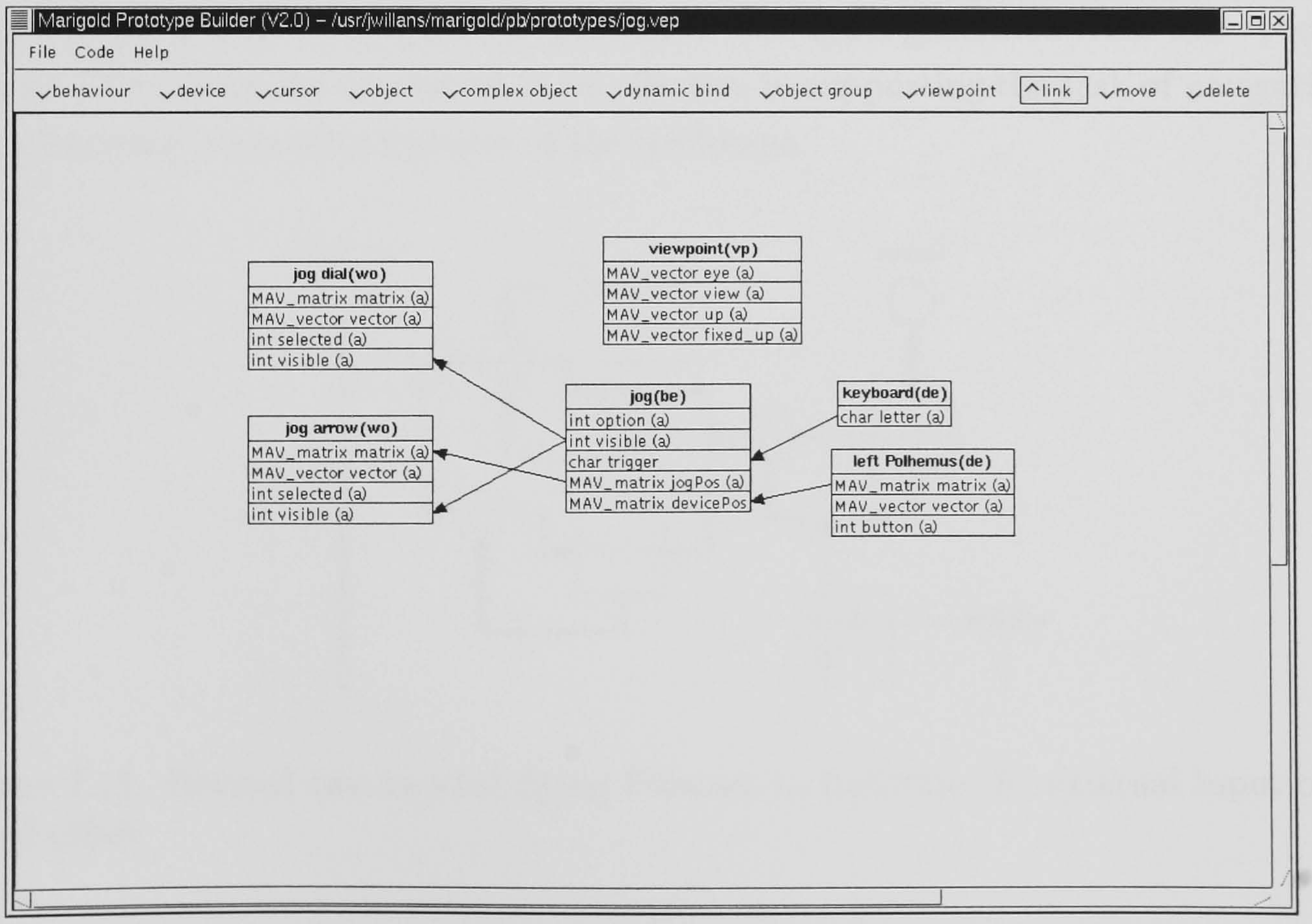

Figure 7.10: A prototype specification constructed to evaluate the jog dial

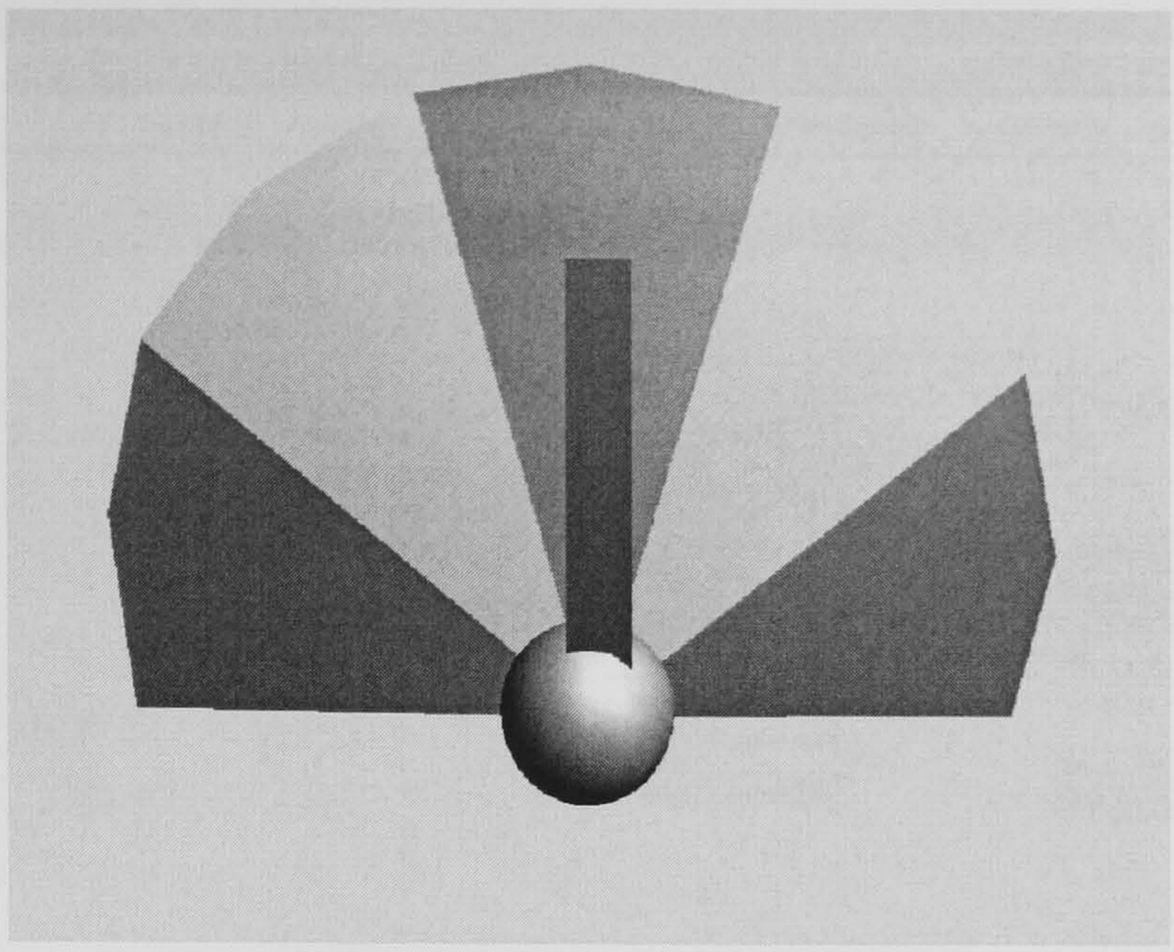

Figure 7.11: Jog dial screenshot

7.2.7 Prototyping the design (2)

The two-handed flying Flownet specification shown in figure 7.4 was revised to facilitate the input of the offset (figure 7.12) and regenerated. The original prototype of figure 7.5 was then augmented to incorporate the complex object for the jog dial. This is shown in figure 7.13. A separate viewpoint was added to the specification to display the jog dial when active. A prototype was again generated from this specification. This prototype was found to be effective in supporting the task of navigating and observing interesting features of the landscape.

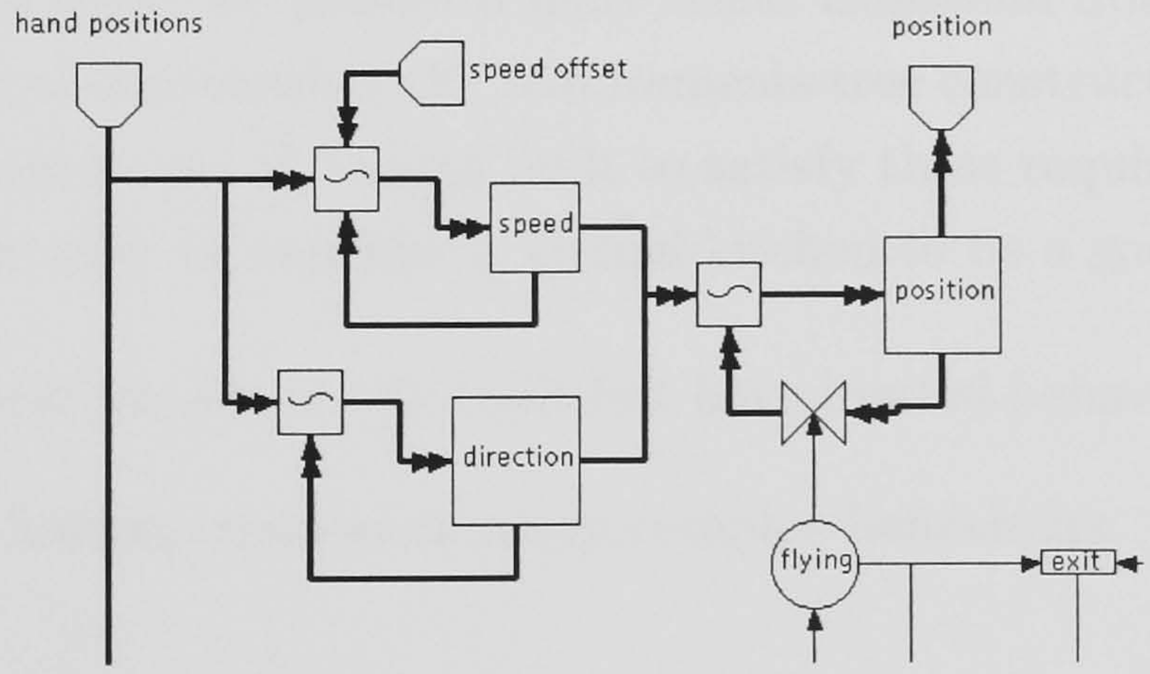

Figure 7.12: Revised two-handed flying Flownet to facilitate the external input of a speed offset

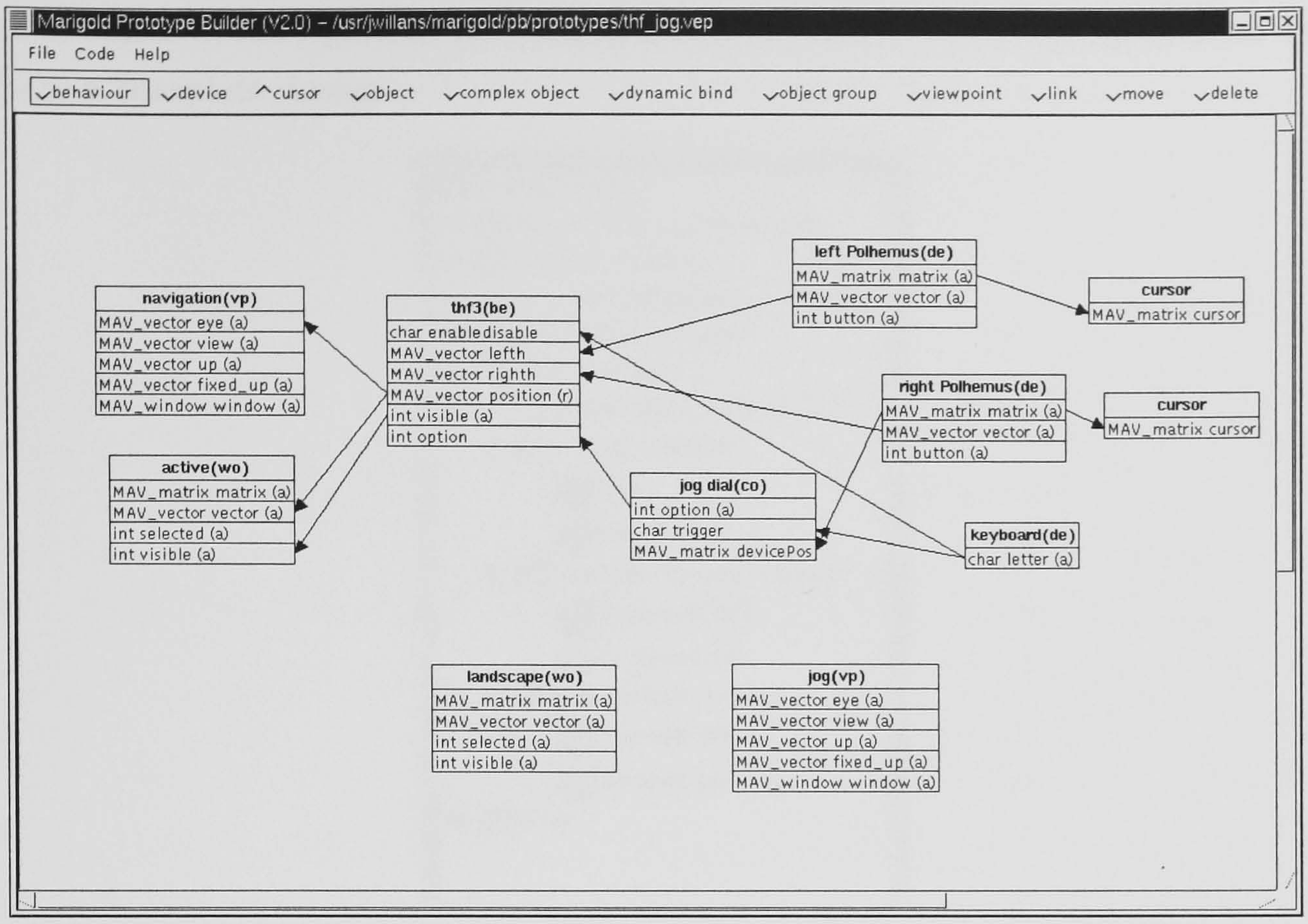

Figure 7.13: Prototype specification for navigating a landscape using the two-handed flying technique with the jog dial technique determining an offset speed

7.3 A virtual kitchen

In the previous case study the precise requirements of the system were initially unclear. In that context, Marigold was able to support the evaluation of alternative designs. In this case study we present a more linear transition from the requirements of a virtual kitchen as expressed in the requirements tree constructed in chapter 6, to a prototype implementation of designs built to satisfy these requirements. There are a number of reasons why we consider a virtual kitchen to be a good case study:

- Virtual kitchens are frequently built but have limited behaviour.
- A real world kitchen consists of many complex behaviours.

7.3.1 Oven

The requirements tree for the virtual kitchen is illustrated in figure 7.14 with the branches unfolded to reveal the requirements for the virtual oven. There are three world objects which exhibit behaviour: the flame, the ignition switch and the gas switch. Initially we built the Flownet design shown in figure 7.15 to satisfy these requirements. Within this design each of the behavioural nodes in the tree became

discrete states in the Flownet, this was subsequently augmented with additional detail to form a complete design.

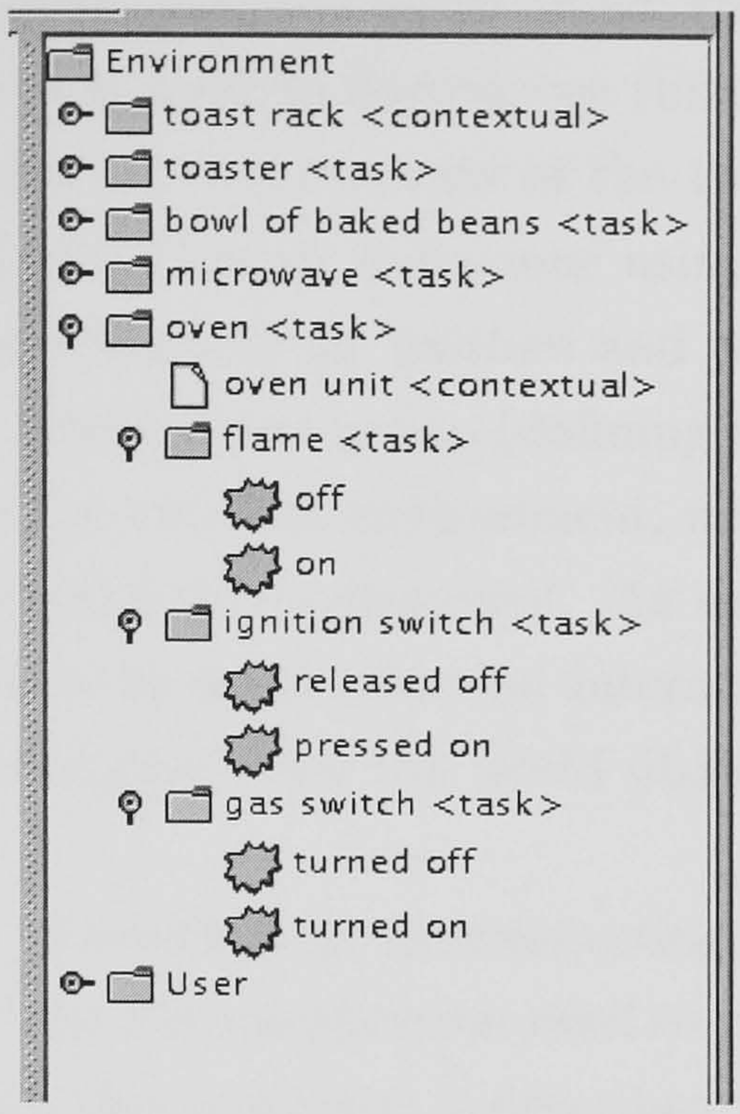

Figure 7.14: Requirements tree exposing those requirements for the oven (within Primrose)

Within the design of the virtual oven we wanted to ensure that the behaviour be realistic. For example, that it not be possible to switch off the flame without turning off the gas switch. Although there may be other ways of extinguishing the flame, such as a cut in the gas supply, this is a rare occurrence and not something we wish to reproduce in our design. We can characterise this aspect of an oven by the following safety property:

- The oven *cannot* have the gas knob turned on and the flame present and *then* have the gas knob turned on and the flame not present.

When this was checked from the Marigold HSB, it was found that this property failed to hold. The trace of the behaviour reported by the tool (figure 7.16) demonstrated that the *gas* token is consumed in order to fire the transition to switch the *flame on*. The consequence of this is that the token in the *flame on* state can be returned to the *flame off* state (since it no longer inhibited by the *gas* state) and the flame is extinguished without turning off the gas knob. The revised Flownet design is shown in figure 7.17 which ensures that when the *gas* token is removed it is immediately replaced before any further behaviour can take place. The property was rechecked and found to hold.

Code was added to some of the nodes of the Flownet specification and a stub of the behaviour was generated. The next stage involved integrating this stub into the world objects of the virtual oven. An oven world object was bought from a third party supplier and a 3D modeller was used to decompose this into separate world objects according to the decompositional requirements of the tree (figure 7.14). The world objects were then linked to the Flownet behaviour using the COB (figure 7.18). A world object group was placed around the *ignition* and *gas switch* object renderings. The matrix variable of this world object group (defining its position and orientation) was linked as an output to the external environment, and the selected variable was linked as an input from the external environment. As we shall demonstrate later in the case study, this allowed us to use a selection interaction technique within a PB specification in order to determine when the world objects within the world object group are selected.

In figure 7.19 a number of screenshots demonstrating interaction with the virtual oven prototype are shown. The PB specification used to integrate the complex object of figure 7.18 and to generate this prototype is described later in the chapter (section 7.3.4). Within the screenshots, the gas switch is on the left and the ignition switch on the right. In figure 7.19 (top left) the oven is in its initial state. In figure 7.19 (top right) the gas is on and the user is pressing the ignition switch. Finally, 7.19 (bottom) shows the oven frying the eggs (i.e. with a flame).

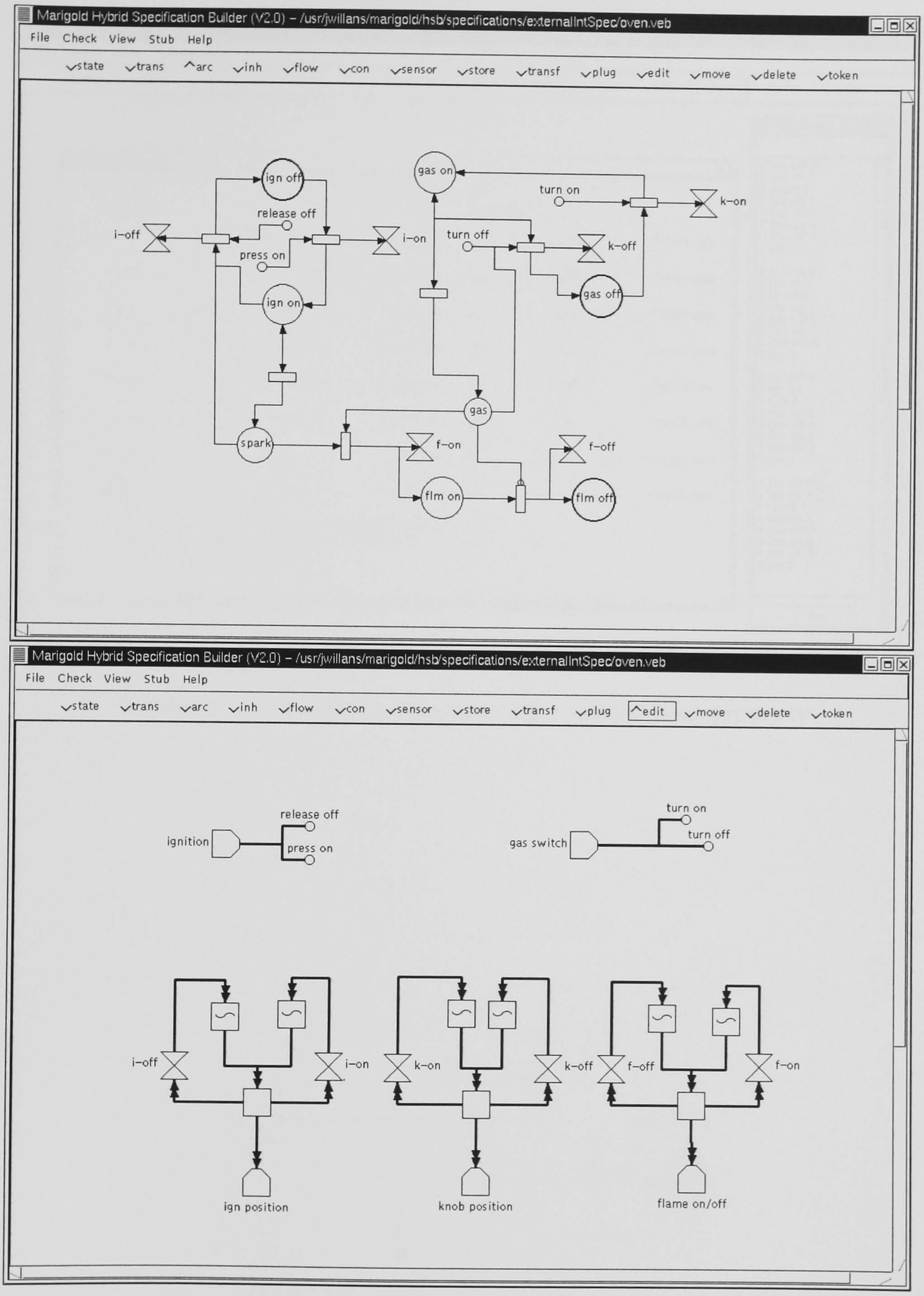

Figure 7.15: Flownet specification for the oven world object

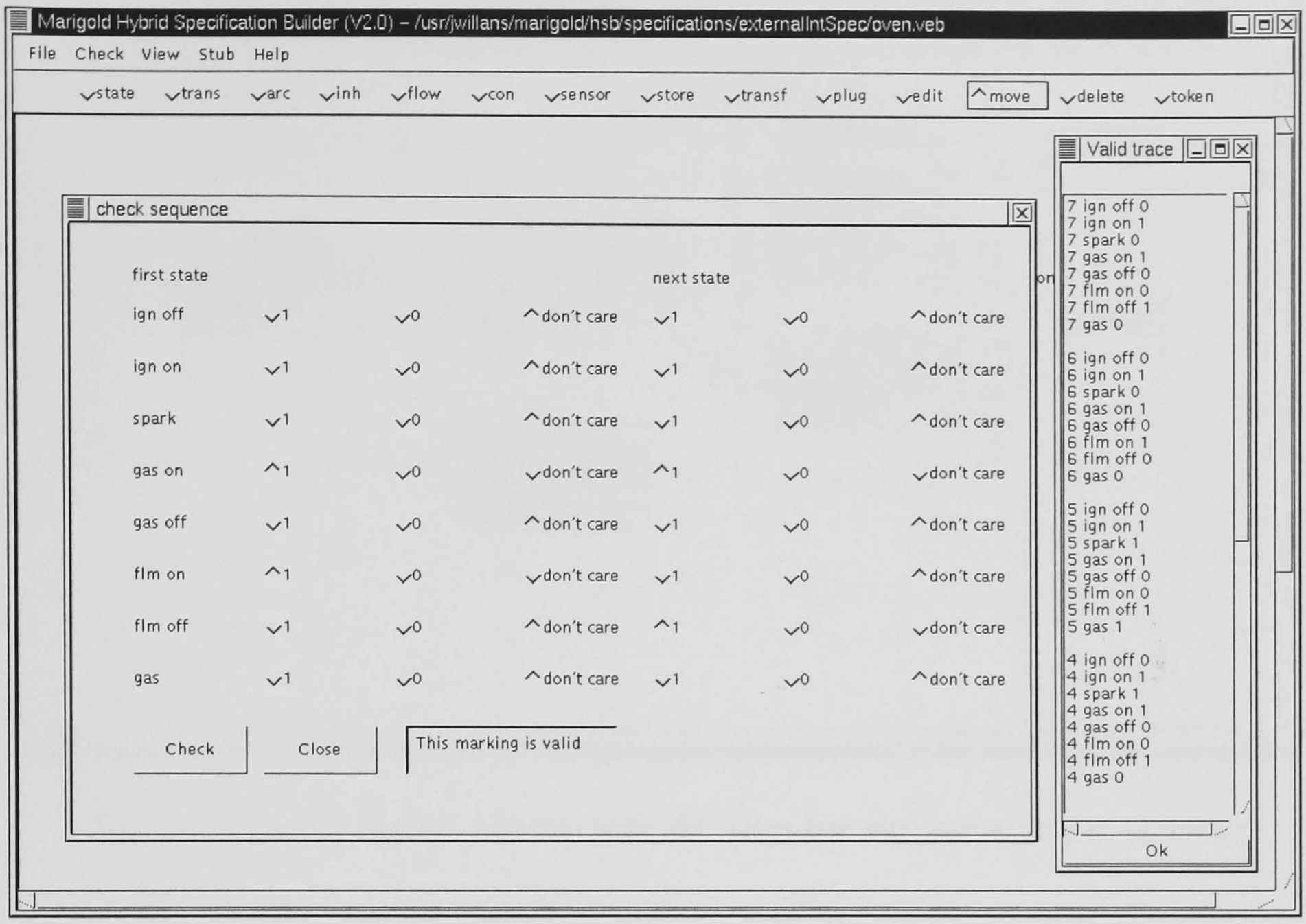

Figure 7.16: Analysing the oven for a correctness property

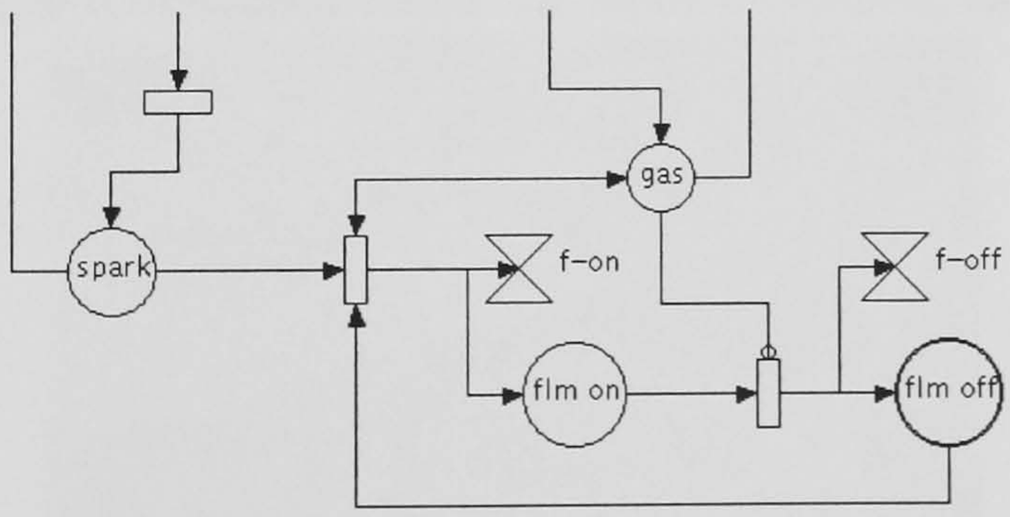

Figure 7.17: Revised Flownet specification for the oven world object

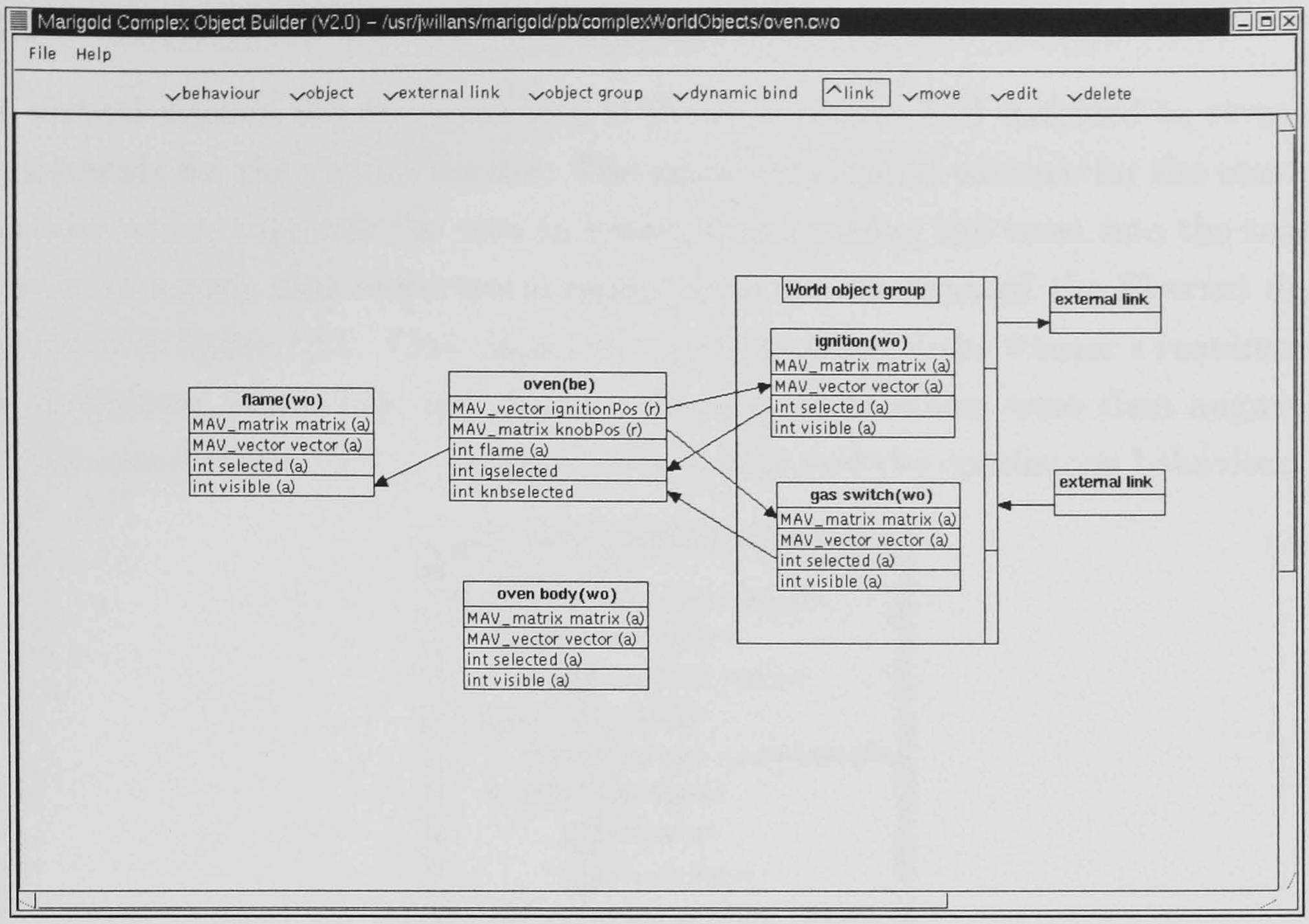

Figure 7.18: Complex object specification for the oven world object

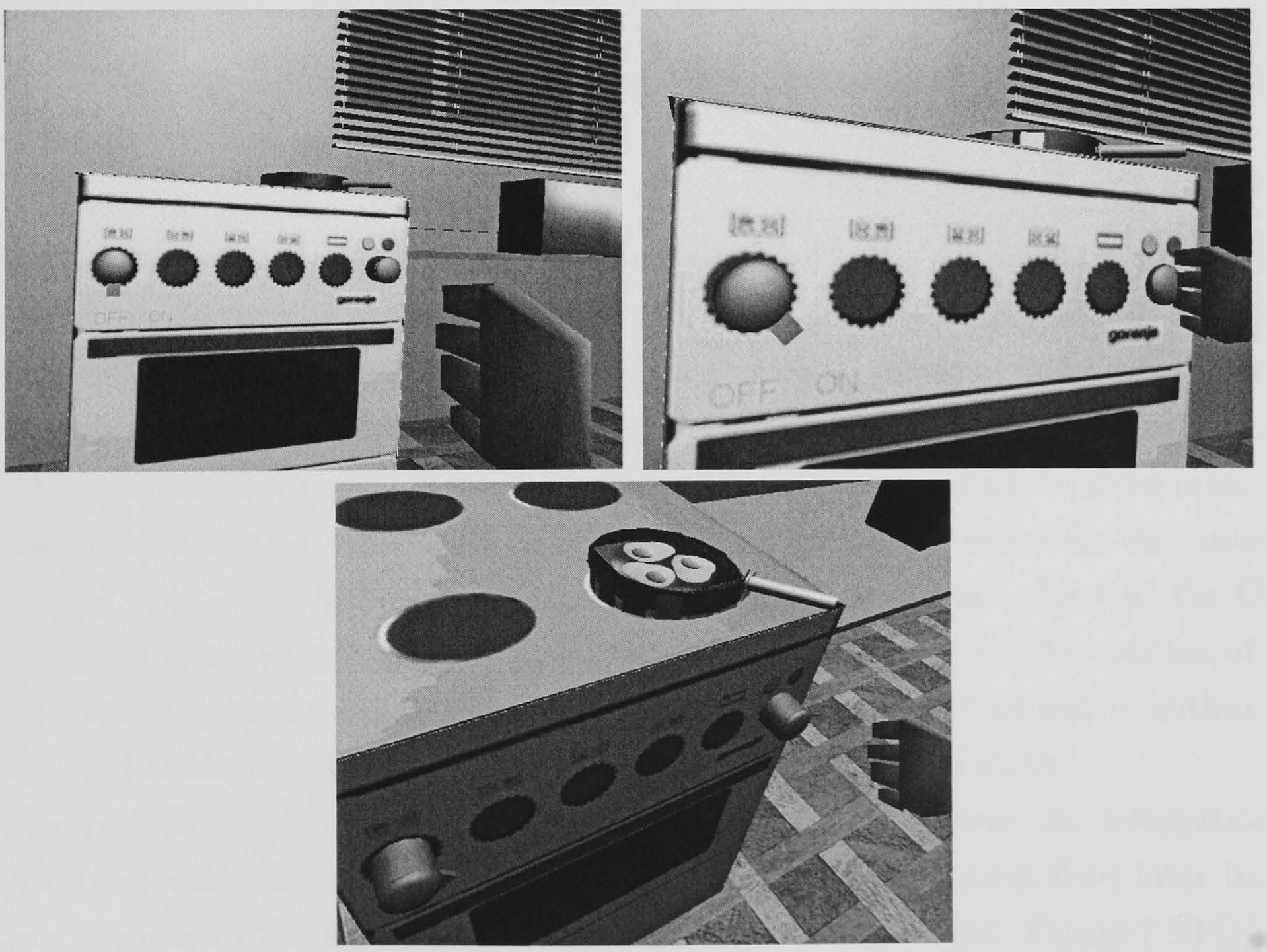

Figure 7.19: Oven in its initial state (top left), oven with gas switched on and ignition switch being pressed (top right), frying the eggs (bottom)

7.3.2 Toaster

The virtual kitchen requirements tree is shown in figure 7.20 unfolded to reveal the requirements for the virtual toaster. The main behavioural concern for the toaster is the slider which supports the user in lowering and raising the toast into the toaster. In order to satisfy this behavioural requirement we constructed the Flownet design illustrated in figure 7.21. This maps the two behaviours of the toaster's requirements tree to discrete states (*up* and *down*). These discrete states were then augmented with intermediate states (*to top* and *follow hand*) and the continuous behaviour.

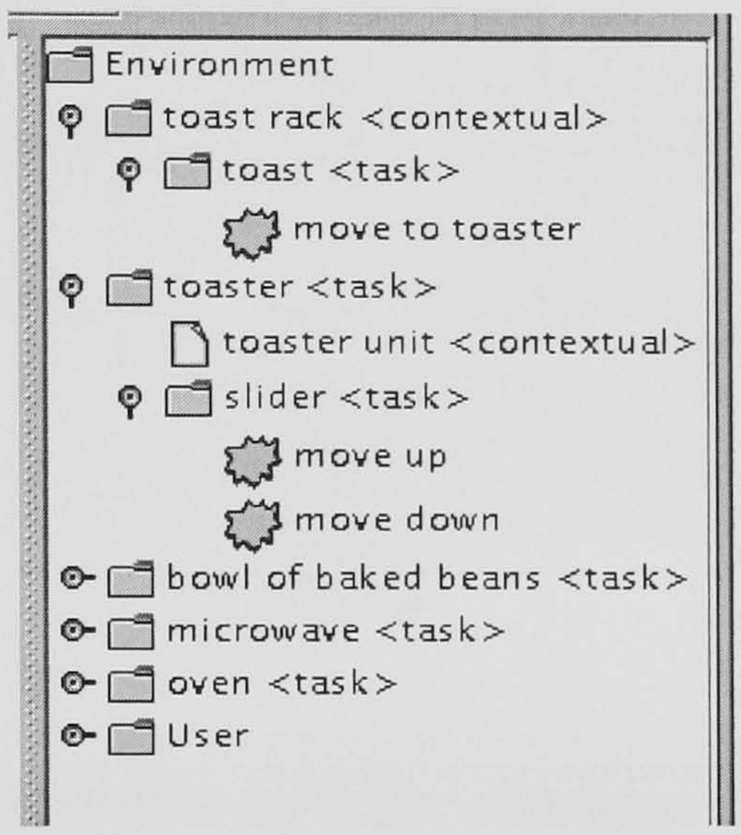

Figure 7.20: Requirements tree exposing those requirements for the gas toaster (within Primrose)

The stub of the toaster's Flownet was integrated with the visual renderings of the world object (decomposed according to the requirements tree) within the COB specification illustrated in figure 7.22. As in the oven, external links were created to the environment outside the world object to determine when components of the object are being interacted with. Additionally, an external link to determine the position of the virtual hand (or selector) was incorporated in order for the slider to follow the position of the hand. Two dynamic binds were also added to the COB specification *bread in left* and *bread in right* which were linked to the position of the slider. These dynamic binds determine when the bread world object is within the toaster and subsequently bind the bread to the position of the slider.

Figure 7.23 shows a screenshot of the toaster resulting from the integration of the COB specification of figure 7.22 into a PB specification (described later in the chapter). Figure 7.23 (left) shows the toaster in its original state. Figure 7.23 (right) shows the user pulling the slider down to begin toasting the bread.

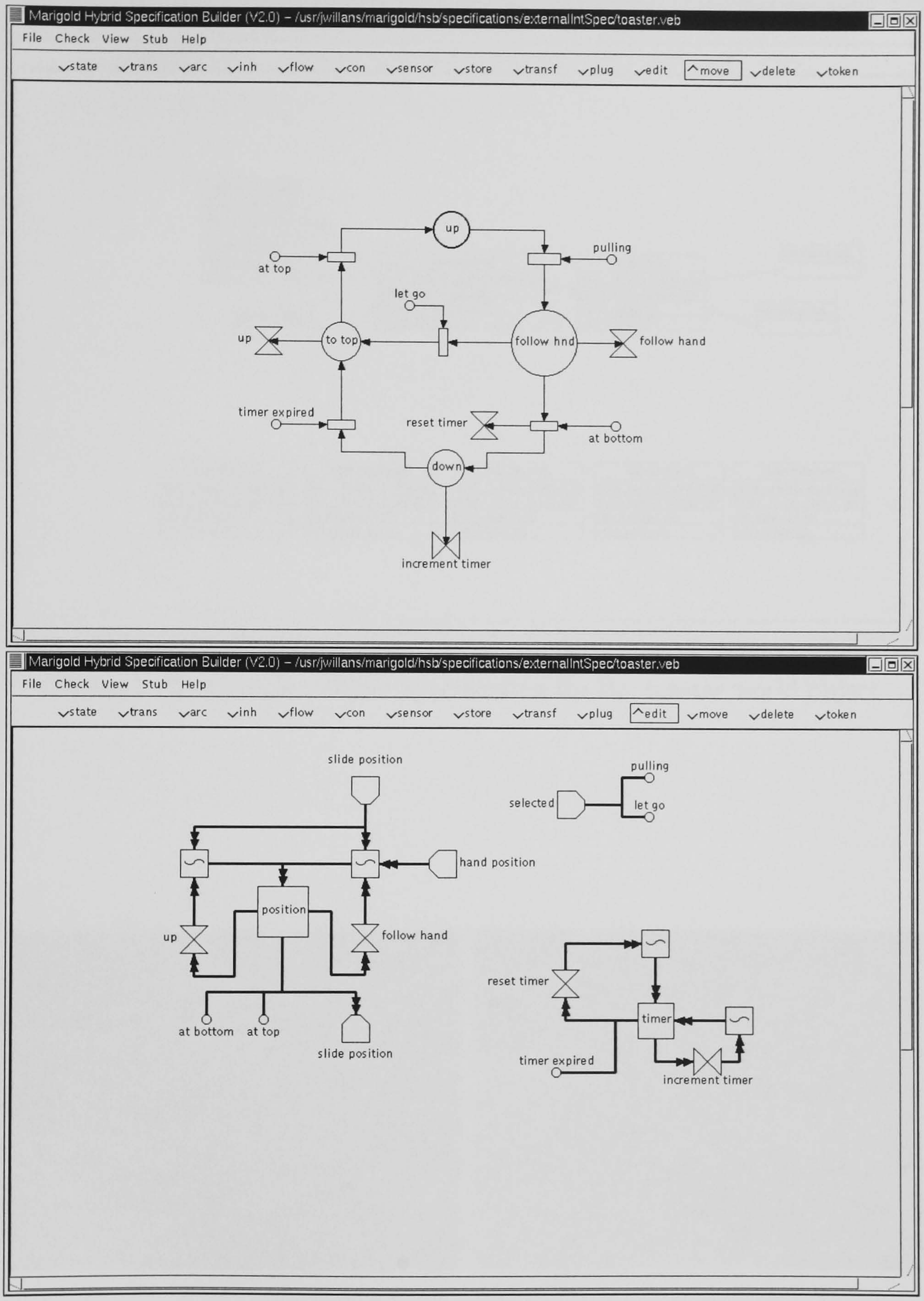

Figure 7.21: Flownet specification for the toaster world object

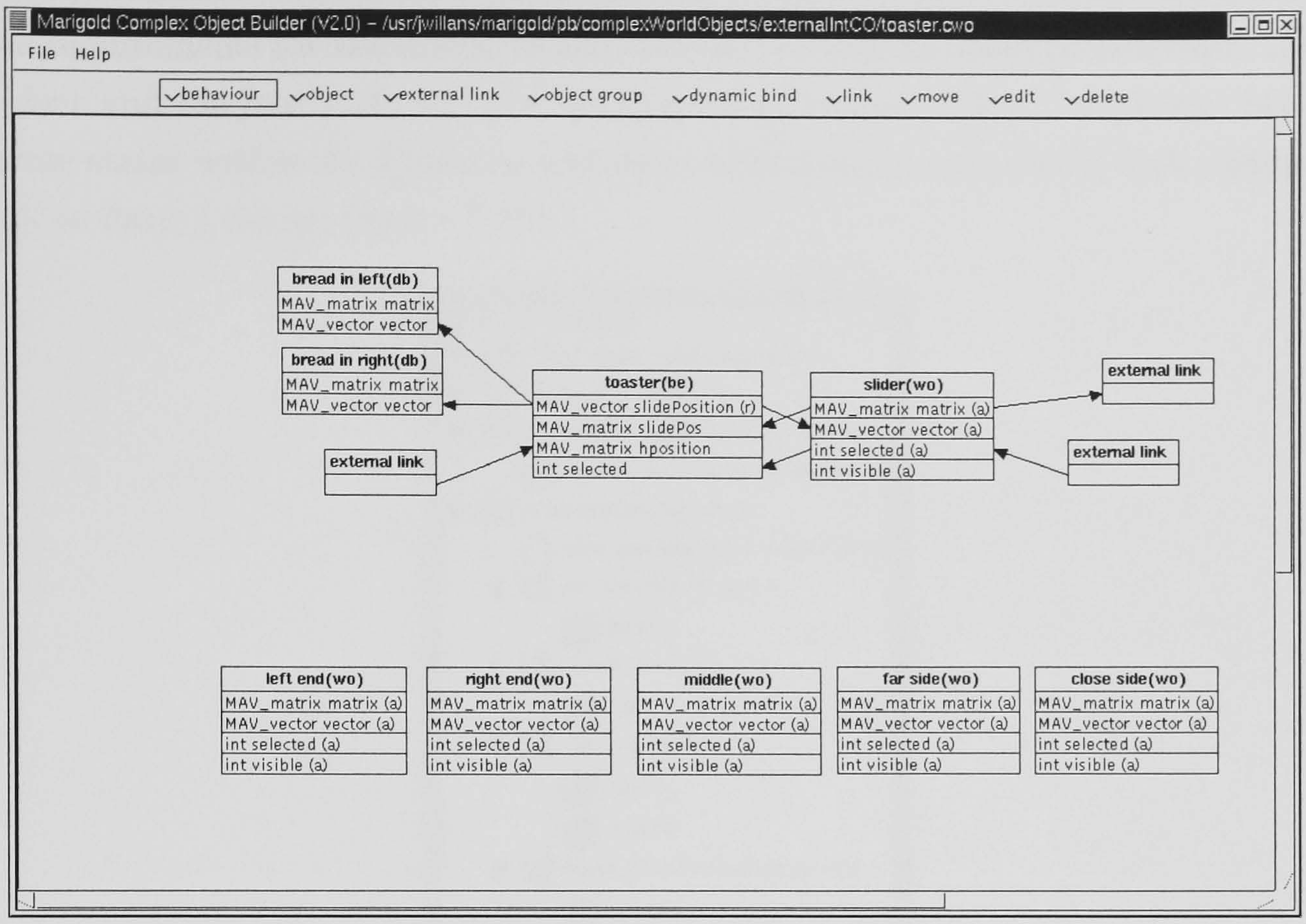

Figure 7.22: Complex object specification for the toaster world object

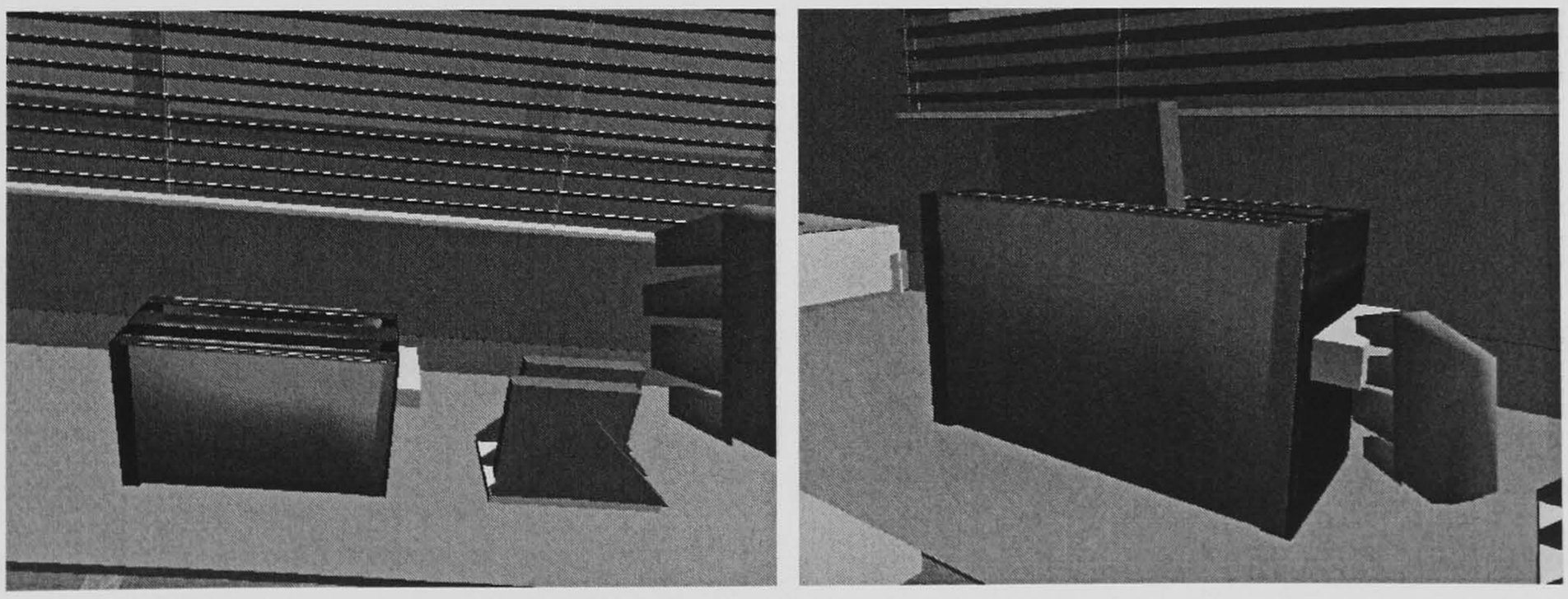

Figure 7.23: Toaster in its initial state (left), pulling the toasters slider to begin toasting the bread (right)

7.3.3 Microwave

The requirements tree for the virtual kitchen is shown again in figure 7.24 exposing those requirements for the design of the microwave. Within this the *on switch*, *timer*, the *door* and the *food plate* all have associated behaviours. These behaviours became discrete states within the Flownet, and were subsequently augmented with additional detail to form a design (figure 7.25).

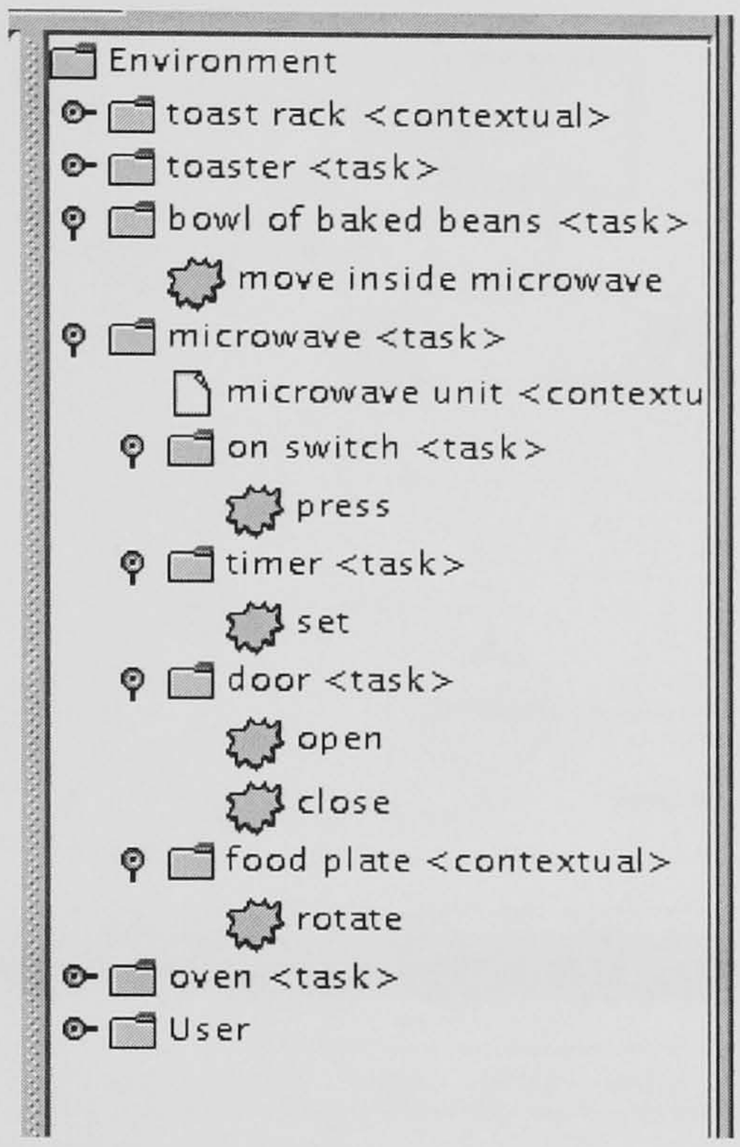

Figure 7.24: Requirements tree exposing those requirements for the microwave (within Primrose)

The stub of the microwave's Flownet was integrated with the visual renderings of the microwave (decomposed according to the requirements tree) using the COB. The resulting specification is illustrated in figure 7.26. As with the oven and the toaster, this behaviour was linked to the external environment to determine when the object is been interacted with. A dynamic bind node was also added to the COB specification to link any world object placed in the microwave to the rotating food plate (*in microwave*).

Screenshots of the microwave in an environment generated from a PB specification (described later) are shown in figure 7.27. In figure 7.27 (top left) the microwave is in its initial state. In figure 7.27 (top right) the microwave is open and the user has placed a bowl of food into the microwave. In figure 7.27 (bottom) the user has closed the door of the microwave and is setting the time in preparation for cooking by pressing the on switch.

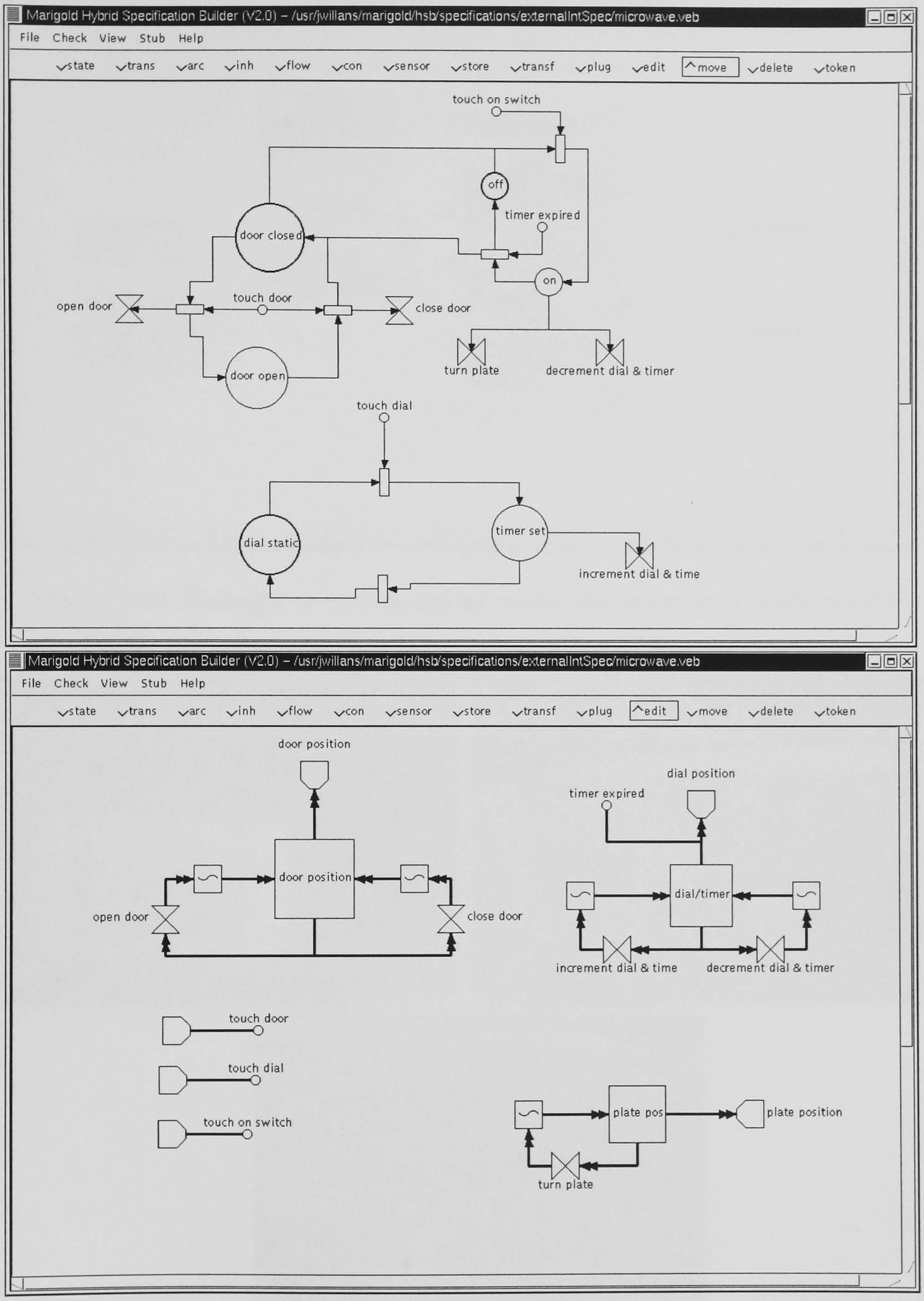

Figure 7.25: Flownet specification for the microwave world object

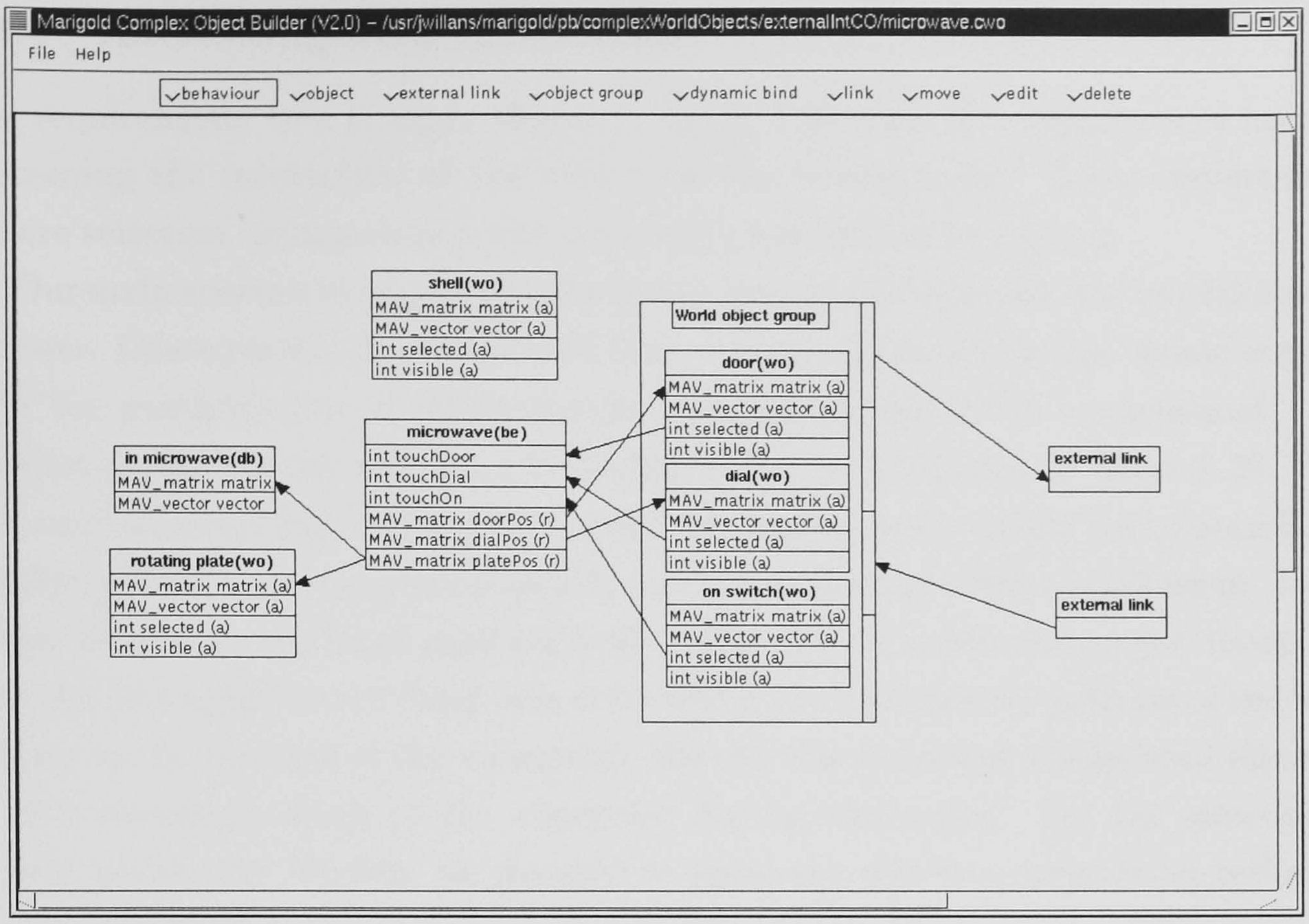

Figure 7.26: Complex object specification for the microwave world object

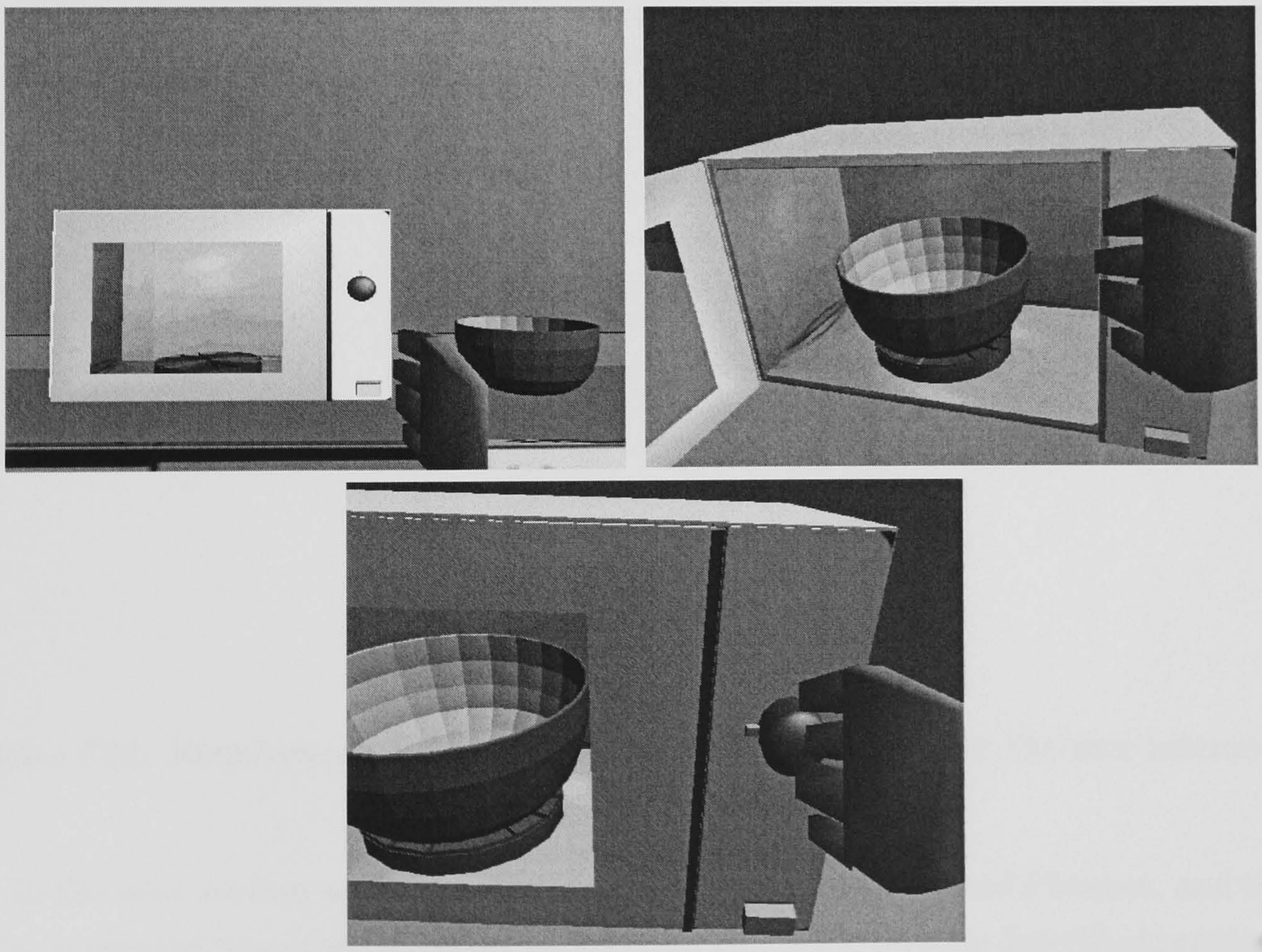

Figure 7.27: Microwave in its initial state (top left), placing food into the microwave (top right), setting the timer (bottom) before pressing the on switch

7.3.4 Interacting with the kitchen

The requirements tree is again shown in figure 7.28 with the requirements exposed concerning the interaction of the user with the environment. These requirements require selection, manipulation and navigation interaction techniques.

Our main concern in supporting the interaction was to minimise the mental load on the user. Consequently it was decided that one interaction technique should support both the manipulation of world objects, and navigation of the environment. The Flownet design for the technique to facilitate this is illustrated in figure 7.29. This Flownet has two main interaction states. The initial state *update both* controls the position of both a virtual pointer (hand) and the navigation of the environment. In the *update both* state the hand position is always projected in relation to the viewpoint. This can be toggled to the *hand only* state which just updates the position of the hand relative to the position of the viewpoint. We call this technique *sticky-hand* since the hand is seemingly stuck to the viewpoint during interaction. For the selection of objects within the kitchen, we decided to reuse the selection interaction technique used in chapter 4.

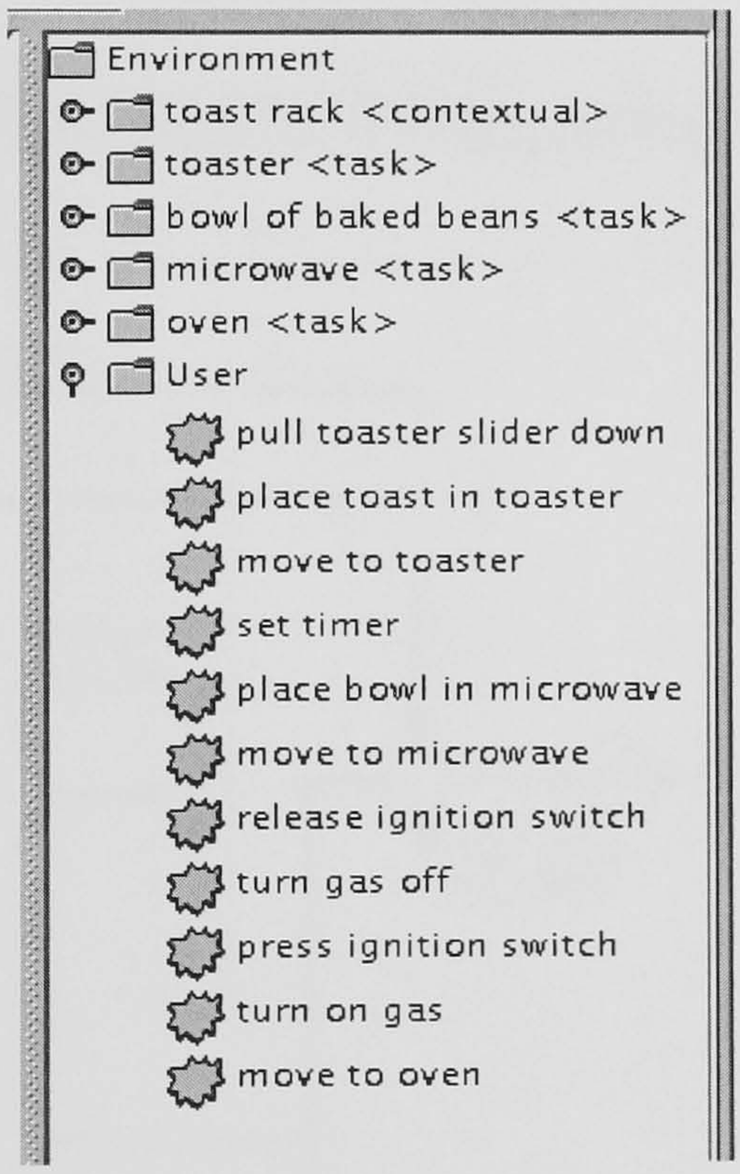

Figure 7.28: Requirements tree exposing those requirements for the user interaction

In the next section we describe how the stub of the sticky-hand Flownet, and those stubs generated from the COB specification for the world objects described previously, were incorporated into a PB specification and then generated into a prototype.

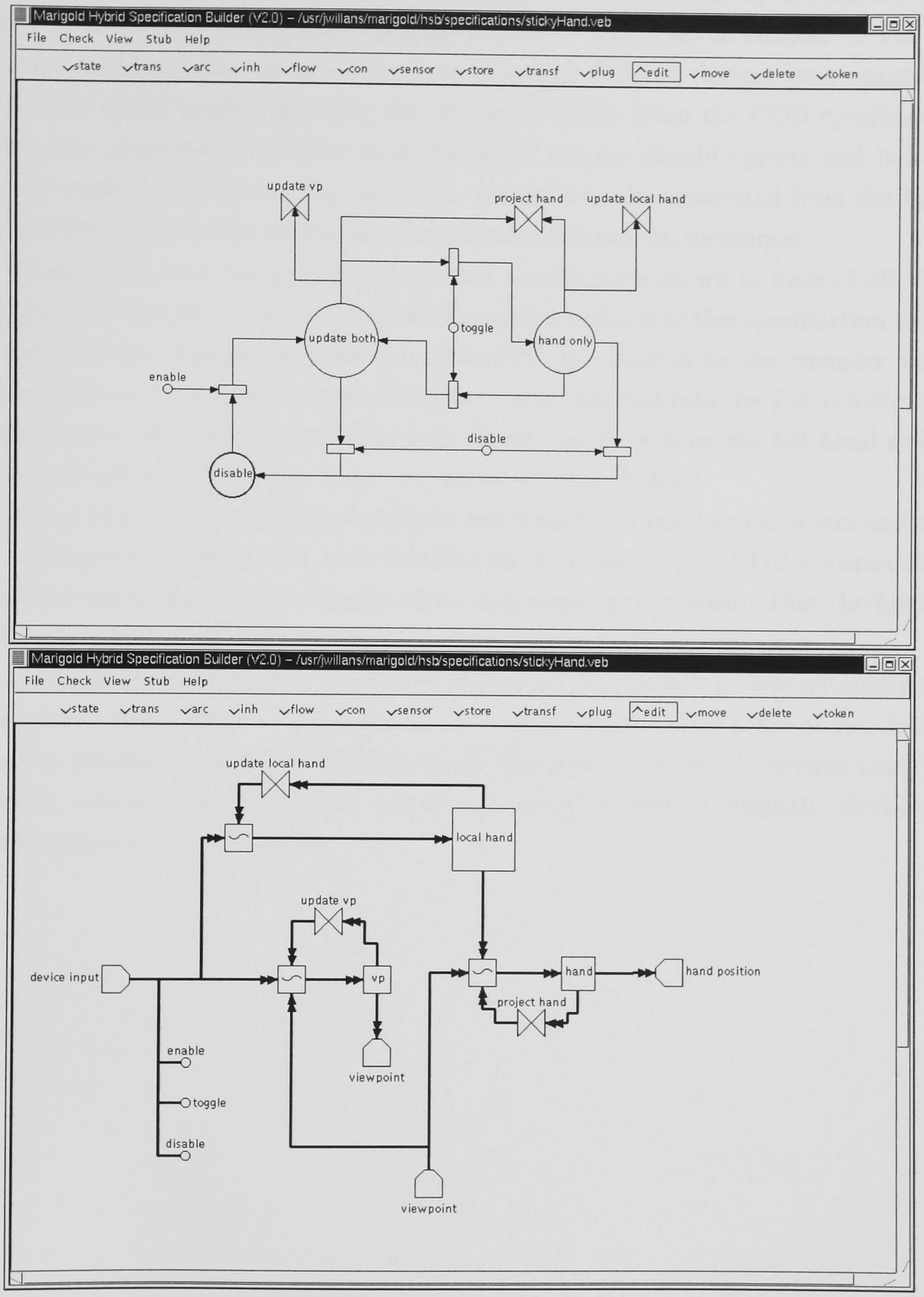

Figure 7.29: Flownet specification for the sticky-hand interaction technique

7.3.5 Kitchen prototype

In the previous sections we have described the Flownet designs specification which were constructed to satisfy the requirements tree constructed in chapter 6. For the complex world objects, we have described how their Flownet designs were integrated with their visual renderings using the Marigold COB. From the COB specification stubs were generated describing how the world objects should appear and behave. For the sticky-hand interaction technique, a stub was also generated from the HSB, and a HSB stub already existed for the selection interaction technique.

These stubs were integrated into the PB specification shown in figure 7.30. The complex world objects can be seen down the right hand side of this specification linked to the selection interaction technique (select3). In addition to the complex world objects, a food bowl and toast world objects were inserted into the PB specification in accordance with the requirements tree. These can be seen on the left hand side of the specification and are also linked to the selection technique.

The sticky-hand interaction technique was linked to a hand world object and also to a viewpoint. Finally, the node labelled *kitchen objects* is a COB encapsulation of all the background objects (walls, floor and worktops) to ensure that the kitchen looks like a real world kitchen. None of these have associated behaviour.

From the PB specification illustrated in figure 7.30 a prototype was automatically generated. Screenshots of the individual complex world objects are shown in the previous sections, and figure 7.31 shows an overview of the whole environment. In order to evaluate the kitchen we tested the scenarios used to originally devise the requirements tree in chapter 6.

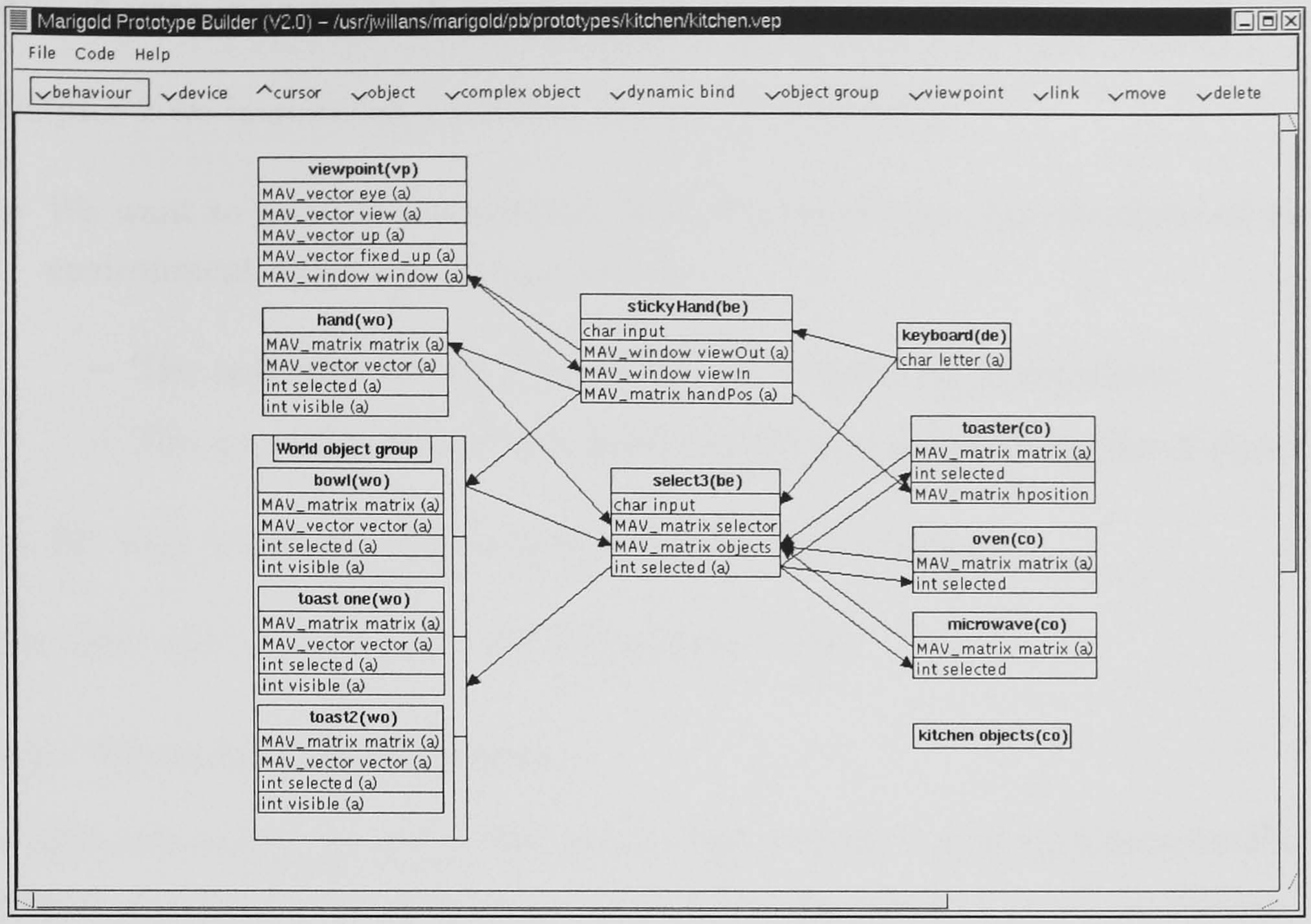

Figure 7.30: Prototype specification for the virtual kitchen

Figure 7.31: Kitchen virtual environment

7.4 Aims revisited

In chapter 2 we introduced a number of aims for the thesis:

- We want to provide a translation from Flownet design specifications of virtual environment behaviour to a prototype.
  - The approach should support the exploration of presentations.
  - The approach should hide implementation concerns from the designer.
- We want to explore the analysis of design specifications.

These aims will be reviewed in the following sections.

7.4.1 Prototyping Flownets

The approach supported by the Marigold toolset provides a transition between Flownet designs of virtual environment behaviour to a prototype realisation of the design. This approach involves adding a small amount of code to some of the Flownet nodes, and then 'plugging' the Flownet into a presentation. An important characteristic of the Marigold toolset is that it not only *enables* the prototyping process, but it also *supports* the prototyping process.

A Flownet is specified using the Marigold HSB. The HSB ensures that the Flownet being constructed is syntactically correct by not allowing connections between incompatible components. When process and conditional code is added to the nodes of the HSB, the tool informs the designer of the data flowing in and out of the nodes. This support ensures a Flownet can be specified rapidly and accurately.

The PB supports the integration of Flownet nodes into a presentation. In the Marigold PB it is not possible to instantiate behaviours or presentation components that do not exist because these must be chosen via a dialogue. The PB ensures that variables being related are of the same type and are of the form input to output. These characteristics of the PB mean that a specification cannot be constructed which will not compile and execute (providing the Flownets have been refined correctly).

The reuse mechanism supported by the Marigold COB further supports the prototyping of Flownet designs. By encapsulating behaviours and the rendering of world objects into a reusable node, the designer does not need to redefine the relation every time it is required. In the first case study the use of the COB was illustrated for the jog dial (section 7.2.6), and in the second case study for the oven (section 7.3.1), toaster (section 7.3.2) and microwave (section 7.3.3).

Exploring presentations

Marigold supports a 'plug and play' approach to relating presentation components to Flownets using data flow networks. The use of data flow networks enables rapid coupling and decoupling of presentations to behaviours and, consequently, enables the exploration of alternative presentations and the evaluation of whether a presentation is suitable for a behavioural design (or *vice versa*). In the first case study (section 7.2) this style of exploratory prototyping was illustrated. The initial prototype demonstrated that the mouse-based flying interaction technique did not allow important parts of the landscape to be viewed because it only supported navigation on the x and z axis (section 7.2). Similarly, a further prototype demonstrated the inability of the Polhemus tracking devices to adequately control speed when used with the two-handed flying interaction technique (section 7.2.5).

Flownet nodes are not concerned with the presentation they interface to. In the second case study (section 7.3) we illustrated the use of the dynamic bind construct for the toaster and microwave (sections 7.3.2 and 7.3.3) which enables decisions about binding behaviour to world objects to be delayed until runtime. For example, the Flownet defining the microwave behaviour is not concerned with whether the behaviour is propagated to the rotating plate or to the bowl of beans (or to the toast, if it is placed in the microwave!).

The visual nature of the Marigold PB and COB enables the relation between behaviour(s) and presentation components to be readily perceived by the designer. For instance, in the first case study (section 7.2) the revised PB specification incorporating the jog dial (figure 7.13) clearly shows that the *right polhemus* is an input device for both the jog dial interaction technique (*jog dial)* and the two-handed flying interaction technique (*thf3*).

Hiding implementation concerns

The use of data flow networks enables Marigold to hide the low-level detail of the implementation from the designer. Nodes encapsulate the underlying complexity of the presentation components that they represent. For instance, an input device node knows about when to poll the device and how to pass data for the device to the variables in the nodes. Similarly, a world object node knows how to place a world object in a data structure that will ensure that it is rendered to the user. The designer does not have to worry about these low-level implementation details.

It is necessary for the designer to add code to some of the nodes of the Flownet specification. This involves some knowledge about data types (vectors and matrices) and the Maverik [Hubbold, Dongbo, and Gibson 1996] functions to transform these data types. However, the code required is minimal compared to the code required to

implement the prototype using a programming language such as Maverik. To give some indication of this, the final prototype developed for the first case study (section 7.2) contained approximately 50 lines of code. The finished generated prototype from the Marigold PB was 600 lines of code. Although, it should be noted that the generated prototype code is not optimal.

7.4.2 Analysing Flownets

The second form of evaluation supported by the Marigold toolset is the automatic analysis of Flownet specifications. The major strength of automated analysis compared to prototyping is that it is exhaustive [Campos 2000]. Two forms of analysis can take place on a Flownet design. Usability characteristics of the design can be checked. This was exemplified in the first case study (section 7.2) when the potential for mode confusion was identified for the two-handed flying interaction technique (section 7.4). The HSB automatically checked the design and specified precisely which part of the design was deficient. The second form of analysis is for correctness of the design according to the functional requirements. This was demonstrated in the second case study when the behaviour of the gas oven was found to be flawed (section 7.3.1). The HSB automatically checked the property and a trace was returned which illustrated a behaviour which would cause the property to fail. This trace informed the revision of the design enabling the behaviour to be consistent with the requirements.

7.4.3 Guiding design using Primrose

In the second case study (section 7.3) the approach supported by the Primrose tool was employed to guide the construction of the virtual environment designs. For the construction of world objects, there is a small step from the behavioural requirements, as expressed in the requirements tree, to Flownet designs. Often this is achieved by simply mapping the requirements to discrete states within the Flownet. This was the case for the oven (section 7.3.1), toaster (section 7.3.2) and microwave (section 7.3.3). Similarly the decompositional requirements precisely describe how the world objects should be decomposed. For the kitchen case study, these requirements allowed us to decompose the third party kitchen world objects in order to accommodate the behavioural requirements.

The behavioural requirements, as expressed in the requirements tree, guides the design of the interaction techniques to a lesser extent. As demonstrated in the kitchen case study (section 7.29), there is not a straightforward mapping between requirements of this type and Flownet designs. However, the behavioural requirements document what the user should be able to achieve in the virtual environment. This

offered some indication of the requirements the design should satisfy in the case of the kitchen.

Marigold and Primrose (along with the use of a 3D modeller) can be seen as supporting a complete requirements and design process for virtual environments. An overview of this process is shown in figure 7.32.

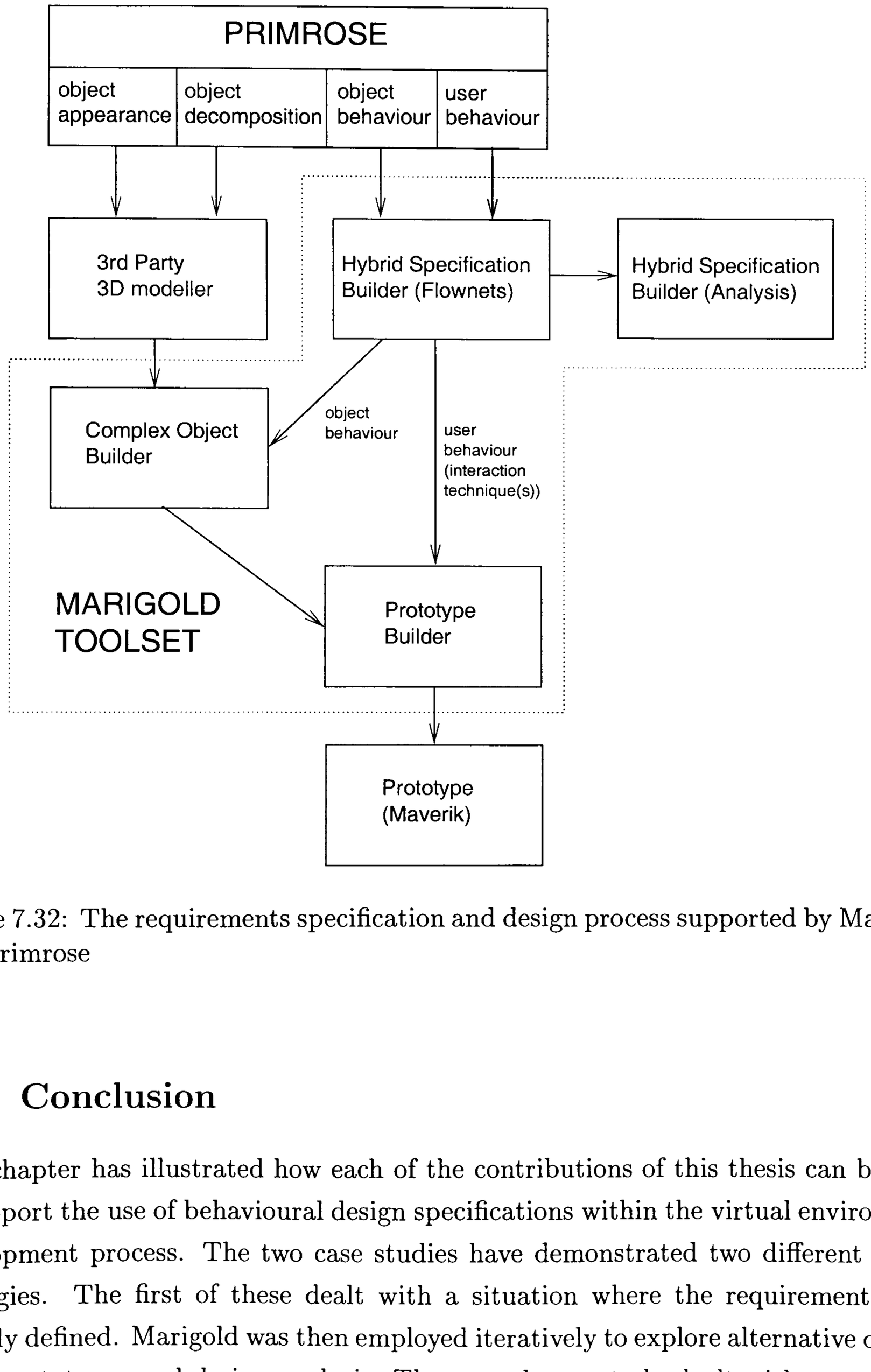

Figure 7.32: The requirements specification and design process supported by Marigold and Primrose

7.5 Conclusion

This chapter has illustrated how each of the contributions of this thesis can be used to support the use of behavioural design specifications within the virtual environment development process. The two case studies have demonstrated two different design strategies. The first of these dealt with a situation where the requirements were roughly defined. Marigold was then employed iteratively to explore alternative designs using prototypes and design analysis. The second case study dealt with prototyping

a design based on more concrete requirements defined using the approach supported by the Primrose tool.

In addition to specifying the individual usefulness of each of the approaches, these case studies have illustrated how the approaches supported by Marigold for the evaluation of behavioural designs, and the requirements elicitation and specification supported by Primrose, can be used as a complete development approach prior to the implementation of an environment. The Primrose approach focusses on the user and their requirements, Marigold then refines these to designs which can be evaluated against the user's requirements.

Chapter 8

Conclusion

8.1 Summary of the thesis

It has been argued that the disadvantage of using implementation code for designing virtual environment behaviour is that there is little correspondence to the language of the requirements. Although this issue is addressed by behavioural design formalisms which support the design of behaviour at an abstract level, a major disadvantage of these formalisms is that they cannot be directly executed and evaluated in the same manner as implementation code. This is a serious obstacle to their application in the virtual environment development process.

This thesis presents two approaches to evaluating behavioural designs which are supported by the Marigold toolset. Firstly, with refining designs to prototypes, so that they can be explored by users. Secondly, with automatically analysing the designs, so that characteristics of the design can be evaluated without prototyping. The Marigold toolset provides usable support for both approaches. The use of data flow networks for prototyping behavioural designs allows the developer to plug the presentation of the environment into the behaviour and to visualise the configuration. The automated nature of the analysis approach, and its informative feedback, supports the developer in applying analysing and understanding the result.

Notwithstanding the level of support and ease of use of Marigold, evaluation of designs will always be an expensive process. The need to apply this type of evaluation can be reduced by being more certain about the requirements that the designs are addressing. This thesis therefore also presents an approach to eliciting and specifying virtual environment requirements when the requirements are based on the real world. Like Marigold, the emphasis within this approach is ensuring that it is easy to apply. This is realised by the Primrose tool which supports the application of the approach.

8.2 Contribution

The thesis contributes to supporting the integration of behavioural design into the virtual environment development process by enabling the evaluation of virtual environment behavioural designs described using Flownets. Evaluation of designs using prototypes has many advantages, the most salient of these is that the user can be involved in the evaluation process. The work presented in chapter 4 describes how this is achieved using the Marigold toolset which facilitates a semi-automatic transition between the design and the prototype implementation.

Despite the advantages of prototypes, they cannot be used to evaluate some properties of the system exhaustively. A common approach to addressing this problem in software engineering is to evaluate the design directly using specification analysis. The approach presented in chapter 5 supports the automatic analysis of Flownet designs. An insight discussed in that chapter is that it is necessary to analyse both usability requirements of the environment, as with traditional interfaces, and the correctness requirements of the environment. The presented approach demonstrates the analysis of both types of requirements and is also supported by the Marigold toolset.

There is a greater chance of producing a satisfactory design if the requirements (which the design is based upon) are an accurate reflection of the user's 'real' requirements. Requirement specification for virtual environments is not an area that has been explored in the past, but given the complex and inconsistent nature of virtual environments, it can be seen as an important part of ensuring their successful development. An approach, supported by the Primrose tool, to eliciting and specifying virtual environment requirements has been provided in chapter 6.

8.3 Designing virtual environments

The central motivation for this thesis is that virtual environments must be designed in a requirements-oriented manner. However, it is unavoidable that the limitations of current technology will influence designs to a certain extent. The tension between requirements at one end of the spectrum, and technology at the other can be addressed by design processes that combine top-down (from the requirements) and bottom-up approaches (from the technology) [Bryson 1995].

Reflecting on the contribution of this thesis we observe that the specification approaches supported by Primrose and Flownets are top-down because they are refining an informal understanding of requirements to a more concrete implementable form. At the same time, the prototyping ability of the Marigold toolset is addressing the bottom-up concerns of designing virtual environments. We observe then a symbiosis between implementation independent specification (not necessarily formal) and

prototyping as a means of addressing the design of virtual environments. This is illustrated in figure 8.1 annotated with the kinds of questions being asked by each part of the process.

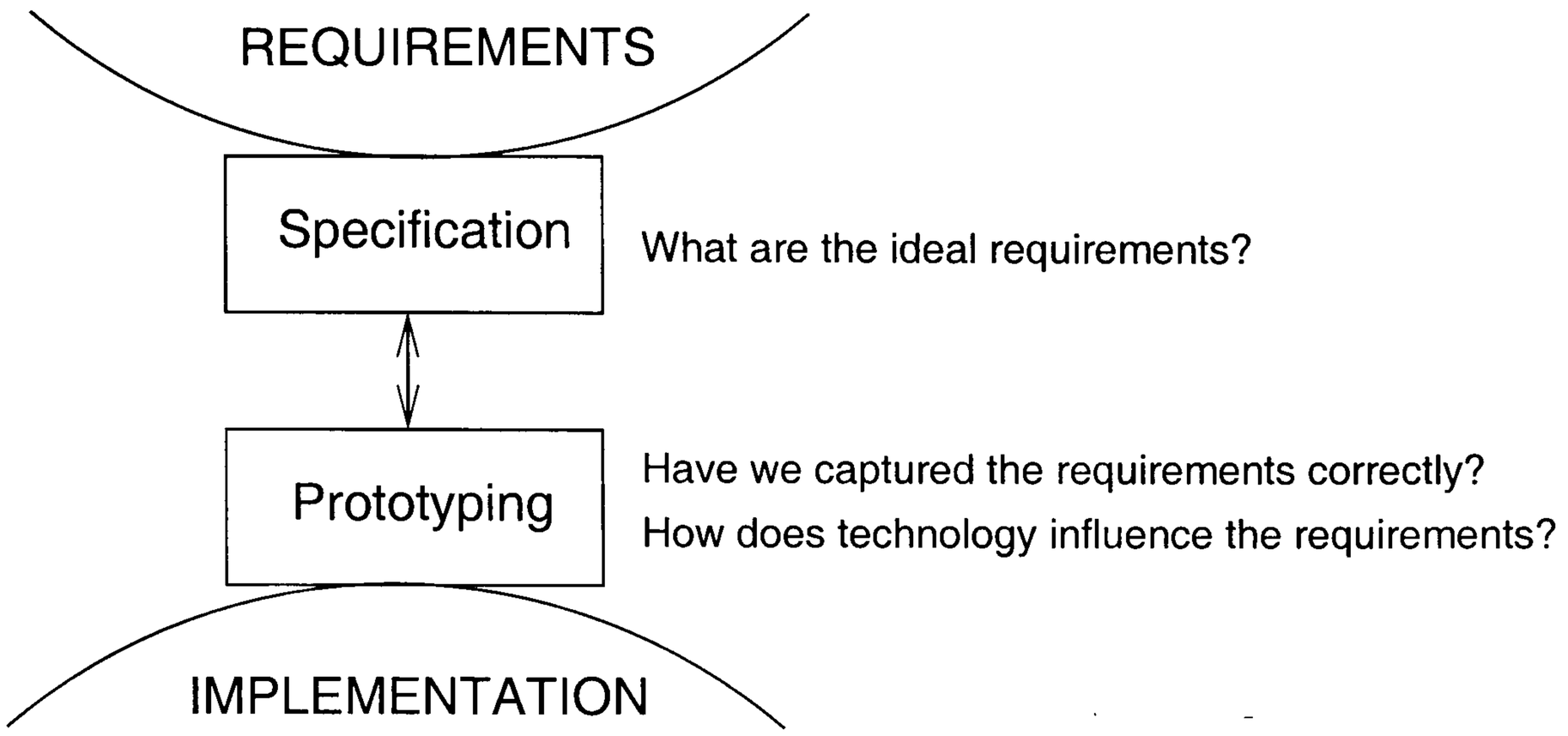

Figure 8.1: Supporting the top-down and bottom-up design of virtual environments using specification and prototyping

8.4 Recent work (revisited)

We have recently become aware of a further work which should be mentioned here. The Shadow system presented in [Morrison 1998] extends the Tufts formalism (described in chapter 2) by introducing hierarchy to the specification formalism. These graphical specifications are manually translated to a textual language. This textual language reproduces the syntax of the graphical specification using tags, within which the developer embeds C++ code segments. From this specification, program code can be generated, compiled and executed.

There are a number of differences between the prototyping approach supported by the Shadow system and that presented in this thesis. Firstly, like the Tufts formalism, discrete behaviour is described using state transition diagrams, in chapter 2 we argued that state transition diagrams do not adequately model the discrete behaviour of virtual environments. Secondly, the granularity of the specification used by the Shadow system is lower level than both Flownets and the original Tufts formalism. Concepts used within these descriptions relate primarily to the implementation rather than the requirements. For instance, variable names and the notion of instantiation of behaviours are incorporated at the highest level of abstraction. Thirdly, the Marigold toolset supports the refinement of the graphical specification to the prototype whereas

with the Shadow system it is necessary to map the graphical representation to the textual equivalent. Finally, the Shadow system offers advantages over Marigold in that large scale virtual environments can be specified because of its use of hierarchy within the specifications.

Marigold and the Shadow system can be seen as complementary approaches to the design and implementation of virtual environments. Marigold could be used at the earlier design stage of development to evaluate alternative designs. The Shadow system could then be used to implement the whole environment once designs have been established.

8.5 Future work

In this section we discuss future directions for the research described in this thesis.

8.5.1 Prototyping

The Marigold toolset is composed of three separable tools so that we could isolate and explore the issues each individual tool is addressing. In practice, it would be desirable that they were integrated so that a transition could be more rapidly made between each of the representations. Indeed, the Primrose tool could also be integrated so that behavioural and world objects specifications are associated with nodes in the tree. This would have the desirable effect that the behavioural and world object specifications can be traced back to the scenario they are addressing with a few mouse clicks.

One of the weaknesses of the prototyping approach supported by Marigold is that it is necessary to place code in some of the nodes of a Flownet specification. This means that its usage remains in the domain of the programmer rather than the designer. A number of strategies could be explored to address this issue. Firstly, data flow networks could also be used to specify the code using low level nodes such as matrix and vector transformations. Secondly, techniques could be borrowed from animation where behaviour (code) is inferred from the specification of a number of key frames (important states of the behaviour). This type of approach has been applied to traditional WIMP interaction in [Frank and Foley 1993; Frank, Sukaviriya, and Foley 1995]. For trivial world object behaviour, this has also been explored for virtual environment in [Yamamoto 1996; Kim and Lee 1998].

The dynamic bind concept proved a particularly interesting and useful abstraction. As discussed in chapter 4, there are parallels between this type of binding and that which was investigated initially in the Aviary system [Snowden 1996], and more recently the Deva system [Pettifer 1999]. Within both these systems laws of physics

are bound to spaces within the environments. In this thesis, we have not explored how laws of physics might be specified and prototyped. Potentially they can be specified using Flownets and linked to an environment using dynamic binds. This is an area worthy of investigation.

8.5.2 Analysis

The analysis of virtual environment specifications has the potential to offer very useful insights into the designs. This thesis has just scratched the surface. Particularly, we have dealt only with discrete properties and, as mentioned in chapter 5, there are characteristics of the continuous behaviour which it would be useful to analyse. In this context, time is an interesting issue which is being dealt with in interactive systems generally [Kutar, Britton, and Nehaniv 2000].

Combined analysis of the specifications of the behaviour and the (appearance of) world objects is also a potential interesting research area. This dual reasoning approach can lead to insights about whether the user will be able to see interesting (important) parts of the environment or whether the user will be able to reach certain world objects, for instance. In this respect, it would also be interesting to see if any of the virtual environment usability guidelines proposed by Kaur [Kaur 1998] can be automatically checked. This is a difficult proposition given the abstract nature of the guidelines and the richness of the specifications required to support such analysis. Potentially this research could draw upon the work examining automatic scene construction and camera placement within computer graphics ([Fleishman, Cohen-Or, and Lischinski 2000], for instance) which deal with similar issues.

8.5.3 Requirements specification

The requirements specification approach presented supports requirements elicitation and specification when the requirements are based on the real world. Often it is the case that the requirements have no real world counterpart. An obvious example of this is the requirements specification for games. In this context, it is less clear how well the approach supported by Primrose might work. For the scenarios, this depends to a large extent on whether it is suitable to conceptualise less novel environments as a series of scenarios. For the requirements tree, this depends whether it is a rich enough description to be interpreted without any real world references. This is probably not the case since it does not include details about behavioural dependencies. Consequently, extensions to the approach supported by Primrose for environments based on non-real world phenomena would be an interesting avenue of research.

As well as addressing the functional and the usability requirements of an environment, there are further requirements that can be seen as important. One example

of this is maintaining the interest of the user while they are interacting with the environment. If a greater insight was gained into what such requirements mean in practice (maybe from cognitive theory) then these could be formalised and applied as templates to the requirements specification approach. Once again, this has great potential in the context of games where there seems to be some informal understanding of these kinds of issues [Kanev and Sugiyama 1998].

Appendix A

A semantics for Flownets

We will use the Z specification language to structure a formal definition of a semantics for Flownet. Z has previously been used for describing the semantics of other formalisms similar to Flownets (Statecharts [Harel 1987] in [Mikk, Lakhench, Petersohn, and Siegel 1997], for instance). This specification has been checked using the *f*UZZ typechecker [Spivey 2000].

A.1 Overview

In this semantics we define the transformation of a Flownet configuration in two stages. Firstly, by describing the configuration of a Flownet: the meaning of components (nodes), the state of components and the relation between the components (arcs). Secondly, we define operations on a Flownet which results in the transformation of its configuration.

A.2 Flownet configuration

First we introduce a basic type to define the name of variables:

$$[NAME]$$

A sensor places one or more thresholds on variables which must be met in order for it to fire. These are characterised by a function mapping variable names to threshold values:

$$Sensor \mathrel{\widehat=} [\, NameValue : NAME \longrightarrow \mathbb{N} \,]$$

A store holds data. This data is characterised by a function mapping variable names to values:

$$Store \mathrel{\widehat=} [\, NameValue : NAME \longrightarrow \mathbb{N} \,]$$

A transformer takes some input value(s) and transforms this to some output value(s). In this definition we are not concerned with the precise nature of the transformation. We abstract from this by considering the transformation as a mapping of variables names:

$$Transformer \mathrel{\widehat=} [\, Transformation : NAME \longrightarrow NAME \,]$$

A plug is a source or destination of data. This data is characterised by a function mapping variables names to values:

$$Plug \mathrel{\widehat=} [\, NameValue : NAME \longrightarrow \mathbb{N} \,]$$

Usually the firing of a transition is wholly determined by the distribution of tokens in the Petri-net and the state of sensors. The exception to this is when a plug is related to a transition via a discrete arc. When this is the case, the data in the plug must meet some threshold value(s) in order for the transition to fire. These thresholds are characterised by a function mapping variable names to values:

$$Transition \mathrel{\widehat=} [\, NameValue : NAME \longrightarrow \mathbb{N} \,]$$

A flow control enables the flowing of data when enabled. The data which is flowing through a flow control is characterised by a function mapping variable names to values:

$$FlowControl \mathrel{\widehat=} [\, NameValue : NAME \longrightarrow \mathbb{N} \,]$$

A place component is defined as a basic type:

$$[Place]$$

A Flownet component can be one of the six types defined above. It is convenient to be able to identify these more abstractly as component types:

$$\begin{aligned} COMPONENT ::= {} & flowControl\langle\langle FlowControl\rangle\rangle \mid \\ & sensor\langle\langle Sensor\rangle\rangle \mid \\ & store\langle\langle Store\rangle\rangle \mid \\ & transformer\langle\langle Transformer\rangle\rangle \mid \\ & plug\langle\langle Plug\rangle\rangle \mid \\ & place\langle\langle Place\rangle\rangle \mid \\ & transition\langle\langle Transition\rangle\rangle \end{aligned}$$

Six operations can be executed upon a Flownet which transform its configuration. We define a type to record each of these:

$$OPERATION ::= sensors \mid iTransitions \mid transitions \mid places \mid flowControls \mid transformers$$

Informally a Flownet configuration consists of:

- A set of components that form the Flownet.
- A set of arcs relating components. These can be standard Petri-net arcs, inhibitor arcs and continuous arcs.
- A set for each of the component groups that can be considered as being active. These can be sensors, transitions, places, transformers and flow controls. For instance, a place can be considered active when it has a token, and a flow control can be considered active when its threshold values have been met.
- A record of which transformation operation has been applied in order to know which operation to next apply.

Within this configuration it is important to preserve the following characteristics:

- The set of arc relating components are restricted to which components they can relate. For instance, an inhibitor arc can only relate places to transitions.
- The sets of active components can only contain components that are in the Flownet.
- If a plug is mapped to a store via a continuous arc, then the data in the plug is also available in the store.
- If a store is mapped to a flow control via a continuous arc, then the data in the store is also available in the flow control.

We formally define a Flownet configuration as:

Flownet

$$
\begin{array}{l}
Components : \mathbb{P}\ COMPONENT \\
Arcs, Inhibitors, Continuous : COMPONENT \leftrightarrow COMPONENT \\
activeSensors : \mathbb{F}\ COMPONENT \\
activeTransitions : \mathbb{F}\ COMPONENT \\
activePlaces : \mathbb{F}\ COMPONENT \\
activeTransformers : \mathbb{F}\ COMPONENT \\
activeFlowControls : \mathbb{F}\ COMPONENT \\
nextOperation : OPERATION \\
\hline
Arcs \subseteq (\mathrm{ran}\ sensor \times \mathrm{ran}\ transition) \cup \\
\qquad (\mathrm{ran}\ transition \times \mathrm{ran}\ place) \cup \\
\qquad (\mathrm{ran}\ transition \times \mathrm{ran}\ flowControl) \cup \\
\qquad (\mathrm{ran}\ place \times \mathrm{ran}\ flowControl) \cup \\
\qquad (\mathrm{ran}\ sensor \times \mathrm{ran}\ transition) \cup \\
\qquad (\mathrm{ran}\ plug \times \mathrm{ran}\ transition) \\
Inhibitors \subseteq (\mathrm{ran}\ place \times \mathrm{ran}\ transition) \\
Continuous \subseteq (\mathrm{ran}\ plug \times \mathrm{ran}\ sensor) \cup \\
\qquad (\mathrm{ran}\ plug \times \mathrm{ran}\ transformer) \cup \\
\qquad (\mathrm{ran}\ transformer \times \mathrm{ran}\ store) \cup \\
\qquad (\mathrm{ran}\ store \times \mathrm{ran}\ transformer) \cup \\
\qquad (\mathrm{ran}\ store \times \mathrm{ran}\ flowControl) \cup \\
\qquad (\mathrm{ran}\ store \times \mathrm{ran}\ sensor) \cup \\
\qquad (\mathrm{ran}\ store \times \mathrm{ran}\ plug) \cup \\
\qquad (\mathrm{ran}\ flowControl \times \mathrm{ran}\ transformer) \\
activeSensors \subseteq Components \\
activeTransitions \subseteq Components \\
activePlaces \subseteq Components \\
activeTransformers \subseteq Components \\
activeFlowControls \subseteq Components \\
\forall p, s : Components; P : Plug; S : Store \mid (P, p) \in plug \wedge (S, s) \in store \\
\qquad \wedge (s, p) \in Continuous \bullet P.NameValue \subseteq S.NameValue \\
\forall s, f : Components; S : Store; F : FlowControl \mid (S, s) \in store \\
\qquad \wedge (F, f) \in flowControl \wedge (s, f) \in Continuous \\
\qquad\qquad \bullet S.NameValue \subseteq F.NameValue
\end{array}
$$

A.3 Transformation operations

In this section we describe the operations that transform the Flownet configuration.

A.3.1 Sensor

A sensor is targeted by either a *Plug* or a *Store* component. In order for a sensor to become active, every threshold *NameValue* in the sensor must find a matching *NameValue* in a *Plug* or *Store* which targets the sensor via. a continuous arc. When a sensor becomes active it is either in, or is added to, *activeSensors*. Otherwise it is not in, or removed from, *activeSensors*:

evaluateSensors

$$
\begin{array}{l}
\Delta Flownet \\
\hline
\forall s : Components; S : Sensor \mid (S, s) \in sensor \bullet \\
\quad (\forall nv : S.NameValue \bullet \\
\qquad \exists c : Continuous \mid second\ c = s \bullet \\
\qquad\quad ((\exists p : Plug \mid (p, first\ c) \in plug \bullet nv \in p.NameValue) \\
\qquad\quad \lor \\
\qquad\quad (\exists st : Store \mid (st, first\ c) \in store \bullet nv \in st.NameValue)) \\
\qquad \land (s \in activeSensors \lor \\
\qquad\quad (s \notin activeSensors \land activeSensors' = activeSensors \cup \{s\}))) \\
\quad \lor \\
\quad (s \notin activeSensors \lor \\
\qquad (s \in activeSensors \land activeSensors' = activeSensors \setminus \{s\}))
\end{array}
$$

A.3.2 Place

The only component that can target a *Place* is a *Transition* component via a Petri-net arc. A place becomes active when at least one targeting *Transition* component is active (they are a member of *activeTransitions*). When a *Place* is evaluated active it is either already a member of, or becomes a member of, *activePlaces*. If a place is not evaluated active, *activePlaces* remains unchanged.

evaluatePlaces

$\Delta Flownet$

$$\forall p : Components \mid p \in \operatorname{ran} place \bullet$$
$$((\exists a : Arcs \mid first\ a = p \bullet$$
$$first\ a \in activeTransitions)$$
$$\wedge ((p \in activePlaces) \vee$$
$$(p \notin activePlaces \wedge activePlaces' = activePlaces \cup \{p\})))$$
$$\vee activePlaces' = activePlaces$$

A.3.3 Transition

For the reasons explained in chapter 3, transitions needs to be considered as two groups. Firstly, interaction transitions whose firing is governed by the external environment. Secondly, non-interaction transitions whose firing is wholly determined by the state of places within the Petri-net.

We will first informally describe interaction transitions followed by the formal schema. An interaction transition is targeted by at least one *Sensor* or *Plug* component via an *Arc* relation, but potentially also by *Place* via an *Arc* relation or by a *Inhibitors* relation. In order for the transition to be, or become a member of the *activeTransitions* set the following must be satisfied:

1. The *Transition* component must be the target of at least one *Plug* component or at least one *Sensor* component. This determines that it is an interaction transition.

2. Every *Sensor* component, targeting the *Transition*, must be a member of *activeSensors*.

3. For every *NameValue* function within the *Transition* there must be a matching *NameValue* in a *Plug* component which targets the *Transition*.

4. Every *Place* component targeting the *Transition* within *Arcs* must be a member of *activePlaces*. After the operation, every *Place* component relating to the *Transition* within *Arcs* must not be a member of *activePlaces* (the tokens are consumed).

5. Every *Place* component targeting the *Transition* within *Inhibitors* must not be a member of *activePlaces*.

6. The *Transition* component is a member of *activeTransition*.

Otherwise the transition is not in, or should be removed from *activeTransitions*:

evaluateITransitions

$\Delta Flownet$

$$
\begin{aligned}
&\forall t : Components;\ T : Transition \mid (T, t) \in transition \bullet \\
&\quad ((\exists c : Continuous \mid second\ c = t \bullet \\
&\qquad first\ c \in \operatorname{ran} plug \lor first\ c \in \operatorname{ran} sensor) \\
&\quad \land \\
&\quad (\forall c1 : Continuous \mid second\ c1 = t \land first\ c1 \in \operatorname{ran} sensor \bullet \\
&\qquad first\ c1 \in activeSensors) \\
&\land \\
&(\forall nv : T.NameValue \bullet \\
&\quad \exists c2 : Continuous \mid second\ c2 = t \bullet \\
&\qquad \exists p : Plug \mid (p, first\ c2) \in plug \bullet nv \in p.NameValue) \\
&\land \\
&(\forall a : Arcs \mid second\ a = t \land first\ a \in \operatorname{ran} place \bullet \\
&\quad first\ a \in activePlaces \land \\
&\qquad activePlaces' = activePlaces \setminus \{first\ a\}) \\
&\land \\
&(\forall i : Inhibitors \mid second\ i = t \bullet \\
&\quad first\ i \notin activePlaces) \\
&\land \\
&(t \in activeTransitions \lor (t \notin activeTransitions \land \\
&\quad activeTransitions' = activeTransitions \cup \{t\}))) \\
&\lor \\
&(t \notin activeTransitions \lor (t \in activeTransitions \land \\
&\quad activeTransitions' = activeTransitions \setminus \{t\}))
\end{aligned}
$$

The distinction of a non-interactive transition is that it becomes active based only on the relation with places linked via Petri-net arcs and inhibitors. Again, we give an informal description followed by the formal schema:

1. The *Transition* component must not be related by any *Plug* or *Sensor* component. This determines that it is a non-interaction transition.

2. Every *Place* component targeting the *Transition* within *Arcs* must be a member of *activePlaces*. After the operation, every *Place* component relating to the *Transition* within *Arcs* must not be a member of *activePlaces* (the tokens are consumed).

3. Each *Place* component targeting the *Transition* within *Inhibitors* must not be a member of *activePlaces*.

4. The *Transition* component is a member of *activeTransition*.

Otherwise the transition is not in, or should be removed from *activeTransitions*:

evaluateTransitions

$\Delta Flownet$

$$
\begin{aligned}
&\forall\, t : Components \mid t \in \operatorname{ran} transition \bullet \\
&\quad (\neg\, (\exists\, c : Continuous \mid second\ c = t \bullet \\
&\qquad first\ c \in \operatorname{ran} plug \lor first\ c \in \operatorname{ran} sensor) \\
&\quad \land \\
&\quad (\forall\, a : Arcs \mid second\ a = t \land first\ a \in \operatorname{ran} place \bullet \\
&\qquad first\ a \in activePlaces \land \\
&\qquad\quad activePlaces' = activePlaces \setminus \{first\ a\}) \\
&\quad \land \\
&\quad (\forall\, i : Inhibitors \mid second\ i = t \bullet \\
&\qquad first\ i \notin activePlaces) \\
&\quad \land \\
&\quad (t \in activeTransitions \lor (t \notin activeTransitions \land \\
&\qquad activeTransitions' = activeTransitions \cup \{t\}))) \\
&\lor (t \notin activeTransitions \lor (t \in activeTransitions \land \\
&\quad activeTransitions' = activeTransitions \setminus \{t\}))
\end{aligned}
$$

A.3.4 Flow control

A flow control is can be targeted by a *Place* or *Transition* component via a Petri-net arc, or a *Store* component via a continuous arc. For a *FlowControl* component to become a member of *activeFlowControls*, one of the *Place* components must be a member of *activePlaces* or one of the *Transition* components must be a member of *activeTransitions*. Otherwise the flow control is not in, or should be removed from *activeFlowControls*:

evaluateFlowControls

$\Delta Flownet$

$$
\begin{aligned}
&\forall\, fc : Components \mid fc \in \operatorname{ran} flowControl \bullet \\
&\quad (\exists\, a : Arcs \mid second\ a = fc \bullet \\
&\qquad first\ a \in activePlaces \lor first\ a \in activeTransitions \\
&\quad \land (fc \in activeFlowControls \lor (fc \notin activeFlowControls \\
&\qquad \land\ activeFlowControls' = activeFlowControls \cup \{fc\}))) \\
&\lor \\
&(fc \notin activeFlowControls \lor (fc \in activeFlowControls \\
&\quad \land\ activeFlowControls' = activeFlowControls \setminus \{fc\}))
\end{aligned}
$$

A.3.5 Transformer

A transformer can be targeted either by a *Plug*, *Store* or *FlowControl* component and can target a *Store* component. Again, we will informally describe the predicate followed by the formal schema. In order for a transformer to be, or become, a member of the *activeTransformers* set, the following must be satisfied:

1. For each of the *Transformations* functions in the *Transformer* component, there must be a targeting *FlowControl*, *Plug* or *Store* with an element of their *NameValue* function with the same domain as the domain of *Transformations*. This ensures that the variable to be transformed is flowing into the *Transformer* component. Additionally, if the data originates from a *FlowControl* component, this component must be active (i.e. a member of *activeFlowControls*).

2. For each of the *Transformations* functions in the *Transformer* component, there must be a targeting *FlowControl*,*Plug* or *Store* with an element of their *NameValue* function with the same domain as the domain of *Transformations*. This ensures that the variable to place the result is flowing into the *Transformer* component. Additionally, if the data originates from a *FlowControl* component, this component must be active (i.e. a member of *activeFlowControls*).

3. For each of the *Transformations* functions in the *Transformer* component, the *Transformer* component must target a *Store* with an element of their *NameValue* function with the same domain as the range of *Transformations*. This checks to ensure that the variable to place the result is a member of a targeting store.

4. The *Transformer* component is a member of *activeTransformers*

Otherwise the transformer is not in, or should be removed from *activeTransformer*:

evaluateTransformers

$\Delta Flownet$

$$
\begin{aligned}
&\forall\, t : Components;\ T : Transformer \mid (T, t) \in transformer \bullet \\
&\quad ((\forall\, nv : T.Transformation \bullet \\
&\qquad \exists\, c : Continuous \mid second\ c = t \bullet \\
&\qquad\quad ((\exists\, f : FlowControl \mid (f, first\ c) \in flowControl \\
&\qquad\qquad \bullet\ first\ nv \in \mathrm{dom}\, f.NameValue \\
&\qquad\qquad\quad \wedge\ first\ c \in activeFlowControls) \\
&\qquad\quad \vee \\
&\qquad\quad (\exists\, p : Plug \mid (p, first\ c) \in plug \\
&\qquad\qquad \bullet\ first\ nv \in \mathrm{dom}\, p.NameValue) \\
&\qquad\quad \vee \\
&\qquad\quad (\exists\, s : Store \mid (s, first\ c) \in store \\
&\qquad\qquad \bullet\ first\ nv \in \mathrm{dom}\, s.NameValue))) \\
\wedge\ &(\forall\, nv : T.Transformation \bullet \\
&\quad \exists\, c : Continuous \mid second\ c = t \bullet \\
&\qquad ((\exists\, f : FlowControl \mid (f, first\ c) \in flowControl \\
&\qquad\quad \bullet\ second\ nv \in \mathrm{dom}\, f.NameValue \\
&\qquad\qquad \wedge\ first\ c \in activeFlowControls) \\
&\qquad \vee \\
&\qquad (\exists\, p : Plug \mid (p, first\ c) \in plug \\
&\qquad\quad \bullet\ second\ nv \in \mathrm{dom}\, p.NameValue) \\
&\qquad \vee \\
&\qquad (\exists\, s : Store \mid (s, first\ c) \in store \\
&\qquad\quad \bullet\ second\ nv \in \mathrm{dom}\, s.NameValue))) \\
&\quad \wedge \\
&\quad (\forall\, nv : T.Transformation \bullet \\
&\qquad \exists\, c : Continuous \mid first\ c = t \bullet \\
&\qquad\quad \exists\, s : Store \mid (s, second\ c) \in store \\
&\qquad\quad\quad \bullet\ second\ nv \in \mathrm{dom}\, s.NameValue) \\
&\quad \wedge\ (t \in activeFlowControls \vee (t \notin activeTransformers \\
&\qquad \wedge\ activeTransformers' = activeTransformers \cup \{t\}))) \\
\vee\ &(t \notin activeFlowControls \vee (t \in activeTransformers \\
&\quad \wedge\ activeTransformers' = activeTransformers \setminus \{t\}))
\end{aligned}
$$

A.3.6 Operation ordering

The operation described above are executed upon a Flownet as illustrated in figure A.1 beginning with the evaluation of sensors. The variable *nextOperation* in the *Flownet* schema records which operation should be executed in the next step. Below we define a precondition schema for each of the operations which checks to ensure

that they should be executed and sets the variable to the value of the next operation to be executed.

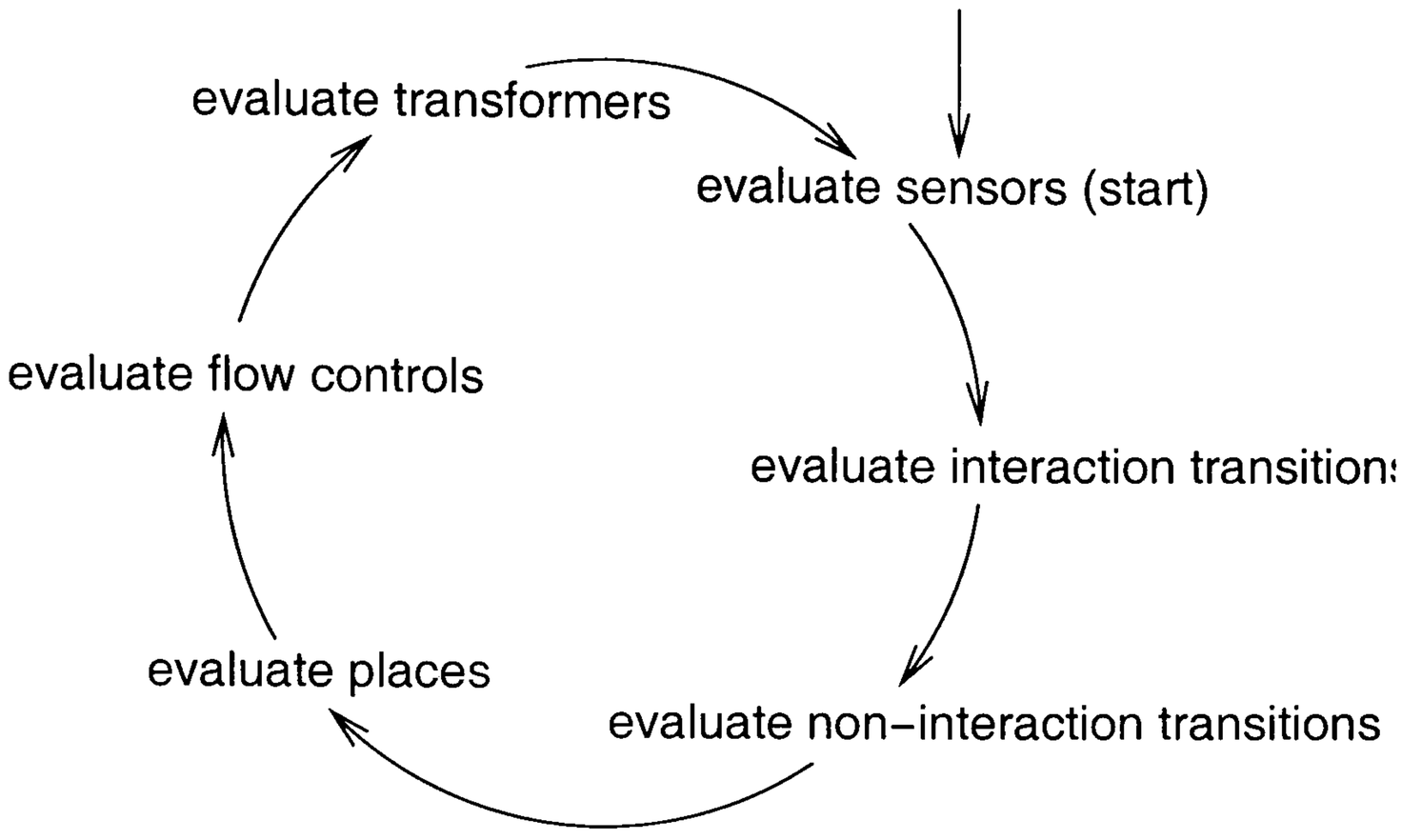

Figure A.1: The execution cycle of operations on a Flownet

For the sensor operation:

sensorsPrecondition

$\Delta Flownet$

$nextOperation = sensors$

$nextOperation' = iTransitions$

$$checkSensors \mathrel{\widehat{=}} sensorsPrecondition \wedge evaluateSensors$$

For the interaction transition operation:

iTransitionsPrecondition

$\Delta Flownet$

$nextOperation = iTransitions$

$nextOperation' = transitions$

$$checkITransitions \mathrel{\widehat{=}} iTransitionsPrecondition \wedge evaluateITransitions$$

For the non-interaction transition operation:

transitionsPrecondition

$\Delta Flownet$

$nextOperation = transitions$

$nextOperation' = places$

$$checkTransitions \mathrel{\widehat=} transitionsPrecondition \land evaluateTransitions$$

For the place operation:

placesPrecondition

$\Delta Flownet$

$nextOperation = places$

$nextOperation' = flowControls$

$$checkPlaces \mathrel{\widehat=} placesPrecondition \land evaluatePlaces$$

For the flow control operation:

flowControlsPrecondition

$\Delta Flownet$

$nextOperation = flowControls$

$nextOperation' = transformers$

$$checkFlowControls \mathrel{\widehat=} flowControlsPrecondition \land evaluateFlowControls$$

For the transformer operation:

transformersPrecondition

$\Delta Flownet$

$nextOperation = transformers$

$nextOperation' = sensors$

$$checkTransformers \mathrel{\widehat=} transformersPrecondition \land evaluateTransformers$$

We also define an operation to initialise the *nextOperation* of a *Flownet* to be the sensor operation.

initFlownet

$\Delta Flownet$

$nextOperation' = sensors$

It is now necessary to ensure that a Flownet only exhibits valid behavioural traces. This is achieved by utilising the approach presented in [Evans 1996]. First, we introduce a new data type mapping numbers to states to record computational traces.

$$\text{comp}\, X == \mathbb{N}_1 \longrightarrow X$$

We then introduce the generic schema of [Evans 1996]. This schema defines that the computation σ is a valid behaviour of a system if its first step belongs to the initial state (I) and subsequent steps belong to the set of possible states of the system. This schema also facilitates an 'idling operation' where the state of the system is not changed (i.e. the next state is the current state).

$$STATE\_ \text{ validcomp } \_ : \text{comp}\, STATE \leftrightarrow (\mathbb{P}\, STATE \times (STATE \leftrightarrow STATE))$$

$$\forall \sigma : \text{comp}\, STATE;\ I : \mathbb{P}\, STATE;\ R : STATE \leftrightarrow STATE \bullet$$
$$\sigma \text{ validcomp } (I, R) \Leftrightarrow$$
$$\sigma(1) \in I \wedge$$
$$(\forall n : \mathbb{N}_1 \bullet \sigma(n)\ \underline{R}\ \sigma(n+1) \vee \sigma(n+1) = \sigma(n))$$

The state Flownet of may be transformed by any of the operations described above. We describe the possible new state of the Flownet by the disjunction of its operations:

$$FlownetTransform \mathrel{\widehat=}$$
$$checkSensors \vee checkITransitions \vee checkTransitions \vee$$
$$checkPlaces \vee checkFlowControls \vee checkTransformers$$

We can now ensure the valid transformation of a Flownet using the *validcomp* schema beginning with the initial state *initFlownet*:

FlownetTransformation

σ : comp *Flownet*

$$\sigma \text{ validcomp } (\{initFlownet \bullet \theta Flownet\},$$
$$\{FlownetTransform \bullet \theta Flownet \mapsto \theta Flownet'\})$$

Appendix B

Marigold details

B.1 Code generation

In this section we describe informally the details of how Marigold achieves the implementation refinement of Flownet designs. Specifically we are concerned with the code generation module of the PB which produces Maverik/C code. This module is separated from the rest of the prototype builder so that new modules for different virtual environment implementations can be written and plugged into the PB.

The first step in the generation process is the unfolding of complex world objects within the PB specification. This involves removing external link nodes, such that there is a direct, rather than a two step link between nodes within a complex object and nodes within the PB specification. This is illustrated in figure B.1.

The second step of the Marigold generation process is to generate Maverik functions for each of the nodes within the PB specification. Each of these nodes, apart from the Flownets, has a corresponding file which defines the Maverik code required to implement their concept. For world objects, viewpoints and dynamic binds, information added via dialogue boxes in the PB and COB such as the initial positioning of world objects (see section B.2) are appended to this code. The generated functions expose input variables to communicate data and/or output variables to pass data.

A Flownet is an augmented condition-event Petri-net. Algorithms for translating such Petri-nets to code are widely available and adopt similar strategies (for instance, [Valette 1986; Bruno and Marchetto 1986; Nelson, Haibt, and Sheridan 1983; Sibertin-Blanc, Hameurlain, and Touzeau 1995]). These need to be extended to support the implementation of Flownets as described in appendix A. For each of the sensors, transitions and transformers within the Flownets, a C function is generated which contains the code added within the HSB. For any functions and transitions which contain no code (unconditional), these are appended to return true. When the sensor and transition functions are called, they update an associated boolean vari-

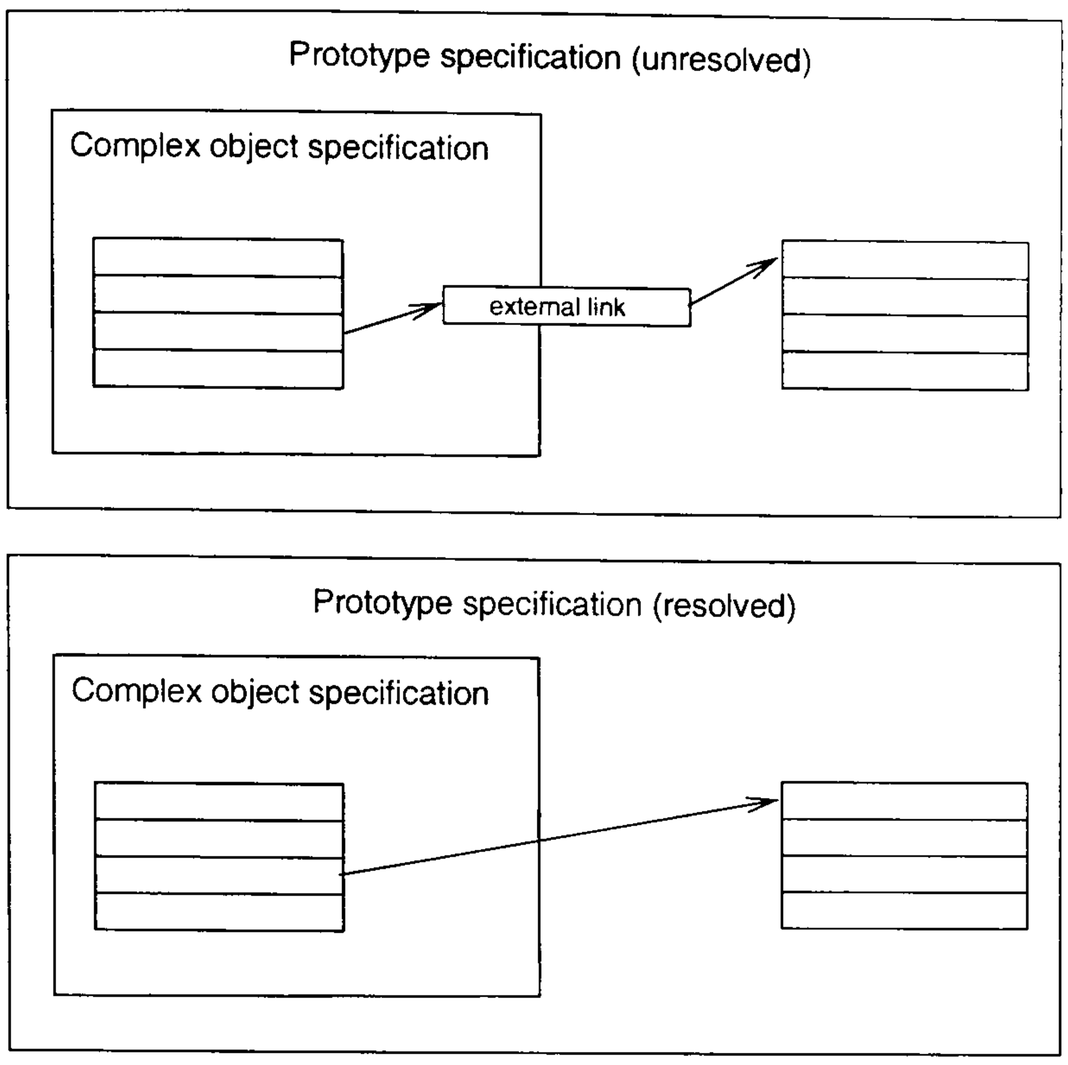

Figure B.1: Resolving complex object specification links during code generation

able indicating whether they have fired. When the transformer functions are called, they perform a single transformation on their local data (as defined in the HSB). The Flownets are then reduced to a standard condition-event net (retaining any inhibitors) which are converted into a single C function using an algorithm similar to that presented in [Bruno and Marchetto 1986] (adopting the semantics described in appendix A). The sensor, transition and transformer functions are called from this function and the resulting state of their variables analysed to determine whether a transition should fire. The pseudo code for this function is given in figure B.2.

In this algorithm, those C functions associated with sensors are called prior to the parsing of the net and their variables evaluated by interaction transitions (interaction transitions being those which are directly influenced by the interaction of the user) during a parse. The C functions associated with interaction transitions are called and evaluated as necessary during the parsing of the Flownet. Those functions associated with transformers are called (depending on the state of the net) subsequent to a parsing of the Flownet. This algorithm checks interaction transitions prior to non-interaction transitions . These details are in accordance with the semantics of Flownets given in Appendix A although we refrain from proving this.

Consequently, there are a number of functions defining each of the nodes (concepts) within the PB specification including the Flownets. A function is then generated which implements the link between the nodes of the unfolded PB specification (mapFunction). This is illustrated in figure B.3.

```
1  parseFlownets()
2  {
3
4    // check all sensors S
5
6    S = all sensors
7    S.updateState()
8
9    // check all interaction transitions T
10   // note: t.associatedFunction is function
11   // associated with transition T
12
13   T  = all interaction transitions
14   S  = all sensors targeting T via arc
15   P  = all places targeting T via an arc
16   PP = all places targeting T via an inhibitor
17
18   if(S.enabled() && P.enabled() &&
19      PP.disabled() && T.associatedFunction())
20   {
21      disable(P)
22      enable(T)
23   }
24
25   // check all non-interaction transitions T
26
27   T = all non-interaction transitions
28
29   if(P.enabled() && PP.disabled())
30   {
31      disable(P)
32      enable(T)
33   }
34
35   // check all place instances 'p'
36
37   T = all transitions targeting 'p'
38
39   if(T.enabled())
40   {
41      disable(T)
42      enable(p)
43   }
44
45   // activate all transformers 'TF' which are
46   // related to an enabled place to update
47
48   AP = P.enabled()
49   TF = all transformers related to AP via a continuous arc
50
51   update(TF)
52 }
```

Figure B.2: Algorithm for executing a Flownet

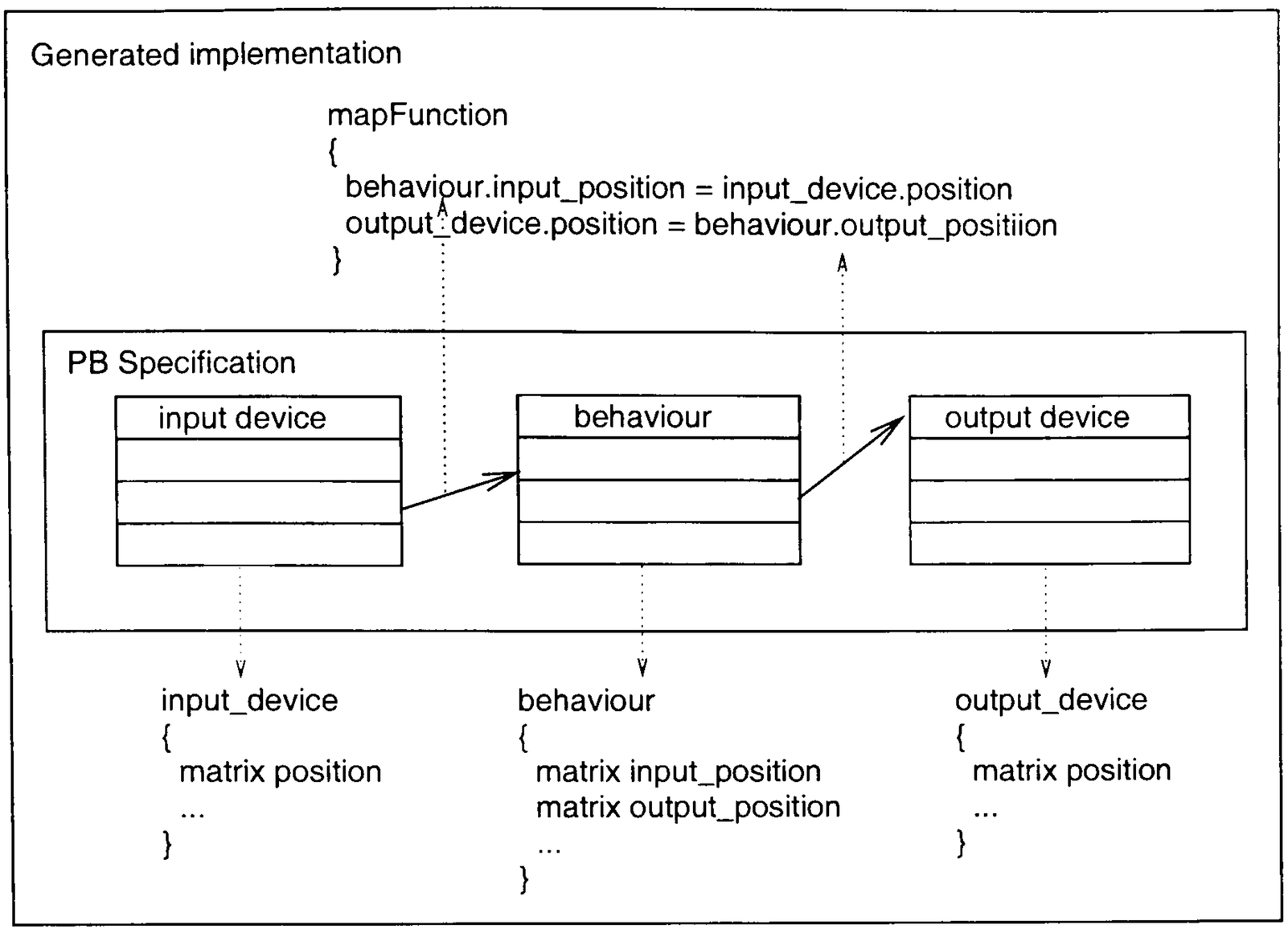

Figure B.3: Mapping from PB specifications to implementation code

The final step of the code generation process is defining when those functions described previously should be called. Within Maverik (and most virtual environment languages) there are two important parts to the program code. Firstly, the initialisation which happens only once during the lifetime of an environment. Secondly, the rendering loop which continually iterates during the lifetime of the environment. Some of the functions generated by Marigold are called in the former such as those that instantiate world objects and viewpoints, others are called in the latter such as those functions which poll devices and parse Flownets. The pseudo-code for this algorithm is given in figure B.4.

```
// initalisation

createDevices ()
createWorldObject ()
createDynamicBindWorldObjects ()
createViewpoints ()

// rendering loop

while ()
{
   checkDevices ()
   mapFunction ()
   parseFlownets ()
}
```

Figure B.4: Main algorithm for executing Flownet specifications

B.2 Editing node properties

Illustrated in figure B.5 are the dialogue boxes which support the editing of node properties within the PB and COB. In figure B.5 (a) the dialogue shown supports editing the initial position and direction of view of a viewpoint.

In figure B.5 (b) the dialogue box shown supports editing the initial positioning and orientation of a dynamic bind within the virtual environment. In addition, this dialogue allows the specification of whether any behaviour should also update the position of the bind (*update bind*) and whether the intersection of the bind with other world objects should be complete or partial (*intersection condition*).

In figure B.5 (c) the dialogue box shown supports editing the initial positioning and orientation of a world object within the virtual environment, and whether the world object should be initially visible and/or selected. In addition, the dialogue supports the specification of whether the selected variable determines whether the world object should be updated, and whether the world object should participate with dynamic binds.

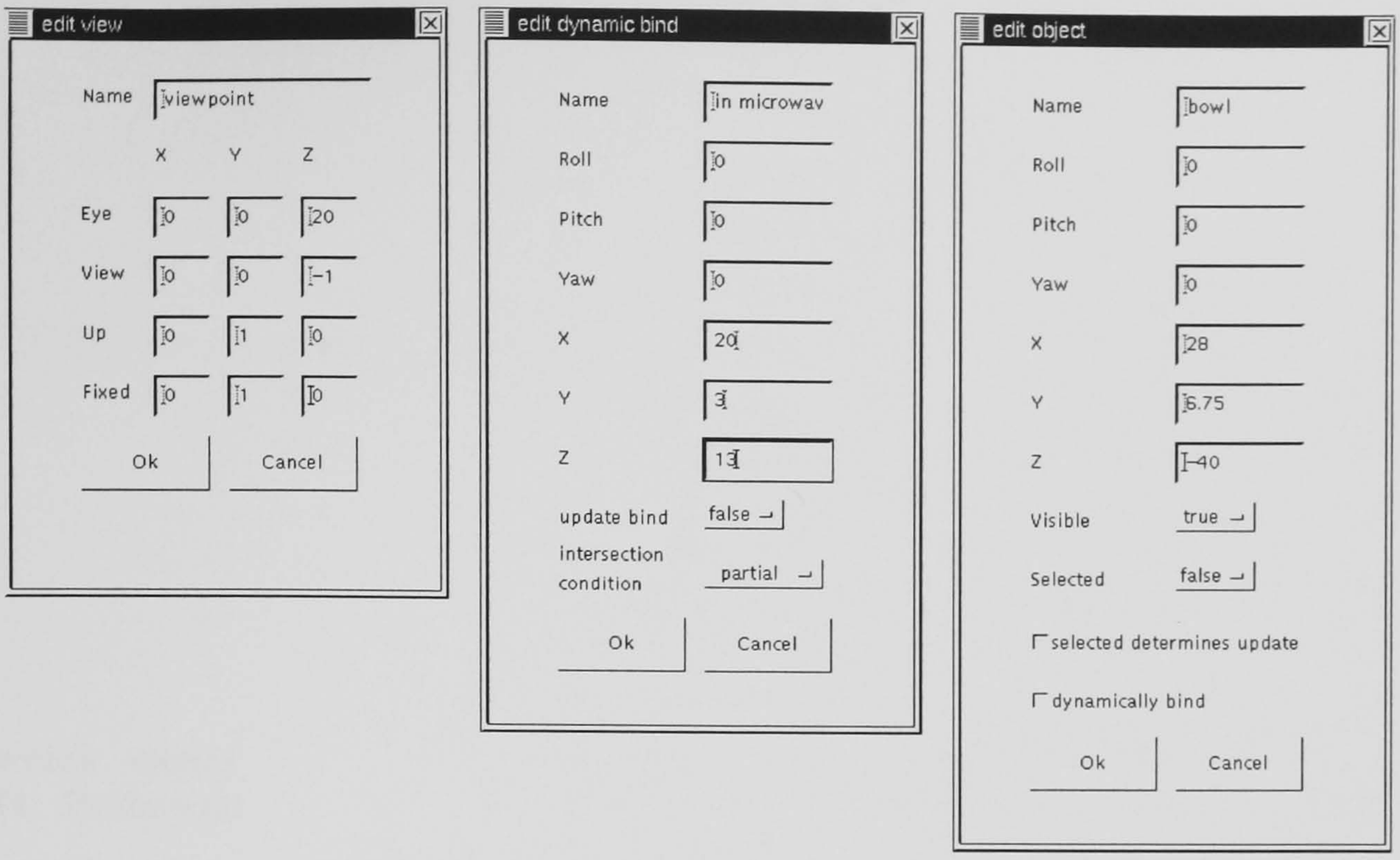

Figure B.5: (a) Editing the properties of a viewpoint node (b) editing the properties of a dynamic bind node (c) editing properties of a world object rendering node

B.3 Device stubs

Marigold is configured to use a number of interaction devices which are defined using device stubs. New devices can be added by creating an appropriate stub for that device, although this involves some knowledge of how Maverik deals with devices. Illustrated in figure B.6 is the stub for a Polhemus tracker. A device stub consists of four parts, terminated by the *end* tag:

- *device name* specifies the name of the device being defined. This is the name which will appear in the node of the PB specification.
- *device variables* specifies the C/Maverik variables that the device updates. These are the variables which will appear in the node of the PB specification.
- *device function* specifies a C/Maverik function that may be required to be iteratively called.
- *device main* specifies C/Maverik code that is always called by the rendering loop of the virtual environment kernel. This should update the variables specified in *device variables* either directly and/or by calling the function in *device function*.

```
{device name}
left Polhemus

{device variables}
MAV_matrix polhemusl_matrix;
MAV_vector polhemusl_vector;
int keyboard_button;

{device function}

{device loop}
polhemusl_matrix = mav_TDM_matrix[3];
polhemusl_vector = mav_matrixXYZGet(mav_TDM_matrix[3]);

{device main}

{end}
```

Figure B.6: The device stub for a Polhemus tracker

Bibliography

Abowd, G. D. (1991). Formal aspects of human-computer interaction. DPhil thesis, University of Oxford.

Alexander, H. (1990). Structuring dialogues using CSP. In M. Harrison and H. Thimbleby (Eds.), *Formal Methods in Human-Computer Interaction*, pp. 273–295. Cambidge University Press.

Autodesk-corporation (1997). 3DStudio. 111 McInnis Parkway, San Rafael, California, 94903, USA.

Bastide, R. and P. Palanque (1990). Petri net objects for the design, validation and prototyping of user-driven interfaces. In *Human-Computer Interaction - INTERACT'90*, pp. 625–631. Elsevier Science Publisher, Amsterdam.

BBC/Colt International (1997). Virtual production planner.

Beaudouin-Lafon, M. (1994). User interface management systems: Present and future. In S. Coquillart, W. Staßer, and P. Stucki (Eds.), *From Object Modelling to Advanced Visual Communication*, pp. 197–223. Springer-Verlag.

Ben, C., A. M. Tawbi, and C. Souveyet (1999). Bridging the gap between users and requirements engineering: the scenario-based approach. Technical Report 99-07, CREWS Report Series, CRI Université Paris.

Berry, D. M. and J. M. Wing (1985). Specifying and prototyping: some thoughts on why they are successful. In *Proceedings of the Internation Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Volume 2, pp. 117–128. Lecture notes in Computer Science.

Bruno, G. and G. Marchetto (1986, February). Process-translatable petri nets for the rapid prototyping of process control systems. *IEEE Transactions on Software Engineering 12*(2), 346–357.

Bryson, S. (1995). Approaches to the successful design and implementation of VR applications. London Academic Press.

Campos, J. (2000). Automated deduction and usability reasoning. DPhil thesis, University of York.

Campos, J. and M. Harrison (1998). The role of verification in interactive systems design. In *Design, Specification and Verification of Interactive Systems'98*. Springer-Verlag. Abingdon, UK, June 3-5.

Carey, R. and G. Bell (1997). *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley.

Carr, D. A. (1996). Towards more understandable user interface specifications. In F. Bodart and J. Vanderdonkt (Eds.), *Design, specification and verification of interactive systems'96*, pp. 141–161. Springer-Verlag.

Clarke Jr., E. M., O. Grumberg, and D. A. Peled (1999). *Model Checking*. MIT Press.

Colebourne, A. (2001). AC3D modeller. `http://www.comp.lancs.ac.uk/computing/users/andy/ac3d.html`.

Cybulski, J. L. and K. Reed (1999). Automating requirements refinement with cross-domain requirements classification. *Australian Journal of Information Systems* (7), 131–145.

Deering, M. F. (1996, May). The holosketch VR sketching system. *Communications of the ACM 39*(5), 54–61.

Degani, A. (1996). *On Modes, Error, and Patterns of Interaction*. Ph. D. thesis, Georgia Institue of Technology, USA.

Deisinger, J., R. Blach, G. Wesche, R. Breining, and A. Simon (2000). Towards immersive modeling - challenges and recommendations: A workshop analyzing the needs of designers. In *Virtual Environments 2000*, pp. 145–146. Springer-Verlag. Amsterdam, 1-2 June.

Denert, E. (1977). Specification and design of dialogue systems with state diagrams. *International Computing Symposium*, 417–424.

Dix, A. and G. Abowd (1996, November). Modelling status and event behaviour of interactive systems. *Software Engineering Journal 11*, 334–346.

Doherty, G., J. Campos, and M. Harrison (2000). Representational reasoning and verification. *Formal Aspects of Computing* (12), 260–277.

Esteban, O., S. Chatty, and P. Palanque (1995). Whizz'ed: a visual environment for building highly interactive software. In *Proceeding of Interact'95*, pp. 121–126.

Evans, A. S. (1996). *Z For Concurrent Systems*. Ph. D. thesis, Leeds Metropolitan University.

Faisstnauer, C., D. Schmalstieg, and Z. Szalavári (1997, August). Device-independent navigation and interaction in virtual environments. Technical Report TR-186-2-97-15, Vienna University of Technology, Austria.

Fields, B., N. Merriam, and A. Dearden (1997). DMVIS: Design, modelling and validation of interactive systems. In *Design, Specification and Verification of Interactive Systems conference proceedings*, pp. 29–45. Springer-Verlag. 4-6 June, Granada, Spain.

Flaus, J.-M. and G. Ollagnon (1997). Hybrid flow nets for hybrid processes modelling and control. In O. Maler (Ed.), *Hybrid and Real-Time Systems International Workshop'97*, pp. 213–227. Spinger Verlag.

Fleishman, S., D. Cohen-Or, and D. Lischinski (2000, June). Automatic camera placement for image-based modeling. *Computer Graphics Forum 19*(2), 101–110.

Foley, J., A. van Dam, S. Feiner, and J. Hughes (1990). *Computer Graphics Principles and Practice.* The Systems Programming Series. Addison-Wesley.

Forrester, J. W. (1961). *Industrial Dynamics.* MIT Press.

Frank, M. R. and J. D. Foley (1993). Model-based user interfaces design by example and interview. In *Proceedings of UIST'93 ACM Symposium on User Interface Software and Technology*, pp. 128–137. ACM press.

Frank, M. R., P. N. Sukaviriya, and J. D. Foley (1995). Inference bear: Designing interactive interface through before and after snapshots. In *Proceedings of the ACM Symposium on Designing Interactive Systems.*

Gibson, S. and T. Howard (2000). Interactive reconstruction of virtual environments from photographs with application to scene-of-crime analysis. In *Proceedings of ACM Symposium in Virtual Reality Software and Technology*, pp. 41–48. ACM Press.

Hack, M. (1975). Petri-net languages. Memo 124, Massachusetts Institute of Technology. Computation Structures Group, Cambridge, Massachusetts.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 231–274.

Harel, D., H. Lachover, A. Naaad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot (1990, July). STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering 16*(4), 403–413.

Harrison, M. and H. Thimbleby (Eds.) (1990). *Formal Methods in Human-Computer Interaction.* Cambridge University Press.

Higgett, N. and S. Bhullar (1998). An investigation into the application of a virtual environment for fire evacuation mission rehearsal training. In P. Hatton (Ed.), *Eurographics 16th Annual UK Conference*, pp. 87–96.

Hoare, C. A. R. (1978, August). Communicating sequential processes. *Communications of the ACM 28*(8), 666–677.

Hodges, L. F., B. A. Watson, B. O. Rothbaum, and D. Opdyke (1996, November). Virtually conquering fear of flying. *IEEE Computer Graphics*, 42–49.

Horrocks, I. (1999). *Constructing the User Interface with Statecharts.* Addison-Wesley.

Hubbold, R. J., X. Dongbo, and S. Gibson (1996). MAVERIK - the Manchester virtual environment interface kernel. In M. Goebel and J. David (Eds.), *Proceedings of 3rd Eurographics Workshop on Virtual Environments.* Springer-Verlag.

Ingalls, D., S. Wallance, Y.-Y. Chow, F. Ludolph, and K. Doyle (1988). The fabrik programming environment. In *IEEE Workshop on Visual Languages*, pp. 222–230.

Jacob, R. J. K. (1986). A specification language for direct-manipulation user interfaces. *ACM Transactions on Computer Graphics 5*, 283–317.

Jacob, R. J. K. (1995). Specifying non-WIMP interfaces. In *CHI'95 Workshop on the Formal Specification of User Interfaces Position Papers.*

Jacob, R. J. K. (1996). A visual language for non-WIMP user interfaces. In *Proceedings IEEE Symposium on Visual Languages*, pp. 231–238. IEEE Computer Science Press.

Jacob, R. J. K., L. Deligiannidis, and S. Morrison (1999, March). A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction 6*(1), 1–46.

Kalawsky, R. S. (1993). *The science of virtual reality and virtual environments.* Addison-Wesley.

Kanev, K. and T. Sugiyama (1998). Design and simulation of interactive 3D computer games. *Computer and Graphics 22*(2), 281–300.

Kaur, K. (1998). *Designing virtual environments for usability.* Ph. D. thesis, City University, London.

Kaur, K., N. Maiden, and A. Sutcliffe (1996). Design practice and usability problems with virtual environments. In *Proceedings of Virtual Reality World '96, Stuttgart, Germany.*

Kim, G. J., K. C. Kang, H. Kim, and J. Lee (1998). Software engineering of virtual worlds. In *ACM Virtual Reality Systems and Technology Conference (VRST'98)*, pp. 131–138.

Kim, M. and E.-T. Lee (1998). A visual interface for scripting virtual behaviors. In *Asia-Pacific Computer Human Interaction*, pp. 165–168. IEEE Press.

Kutar, M., C. Britton, and C. Nehaniv (2000). Specifying multiple time granularities in interactive systems. In P. Palanque and F. Paternó (Eds.), *Interactive systems design, specification and verification*, pp. 51–63. Springer-Verlag LNCS 1946.

Kutti, K. (1995). Work processes: Scenarios as a preliminary vocabulary. In J. M. Carroll (Ed.), *Scenario-Based design*, pp. 19–36. John Wiley and Son.

Lucena, F. N. and H. K. E. Liesenberg (1994). A statechart engine to support implementation of complex behaviour. In *Proceedings of the 21st Semish Conference*, pp. 177–191.

Massink, M., D. Duke, and S. Smith (1999). Towards hybrid interface specification for virtual environments. In D. Duke and A. Puerta (Eds.), *Design, Specification and Verification of Interactive Systems '99*, pp. 30–51. Springer-Verlag.

Mikk, E., Y. Lakhench, C. Petersohn, and M. Siegel (1997). On formal semantics of statecharts as supported by STATEMATE. In D. J. Duke and A. S. Evans (Eds.), *2nd BCS-FACS Northern Formal Methods Workshop*. Springer-Verlag.

Mine, M. R., F. P. Brooks Jr, and C. H. Sequin (1997). Moving objects in space: Exploiting proprioception in virtual-environment interaction. In *SIGGRAPH 97*, pp. 19–26. ACM Press.

Morrey, I., J. Siddiqi, R. Hibberd, and G. Buckberry (1998). A toolset to support the construction and animation of formal specifications. *The Journal of Systems and Software* (41), 147–160.

Morrison, S. A. (1998). *A specification paradigm for design and implementation of non-WIMP human-computer interactions*. Ph. D. thesis, Tufts University.

Morrison, S. A. and R. J. K. Jacob (1998). A specification paradigm for design and implementation of non-WIMP human-computer interaction. In *ACM CHI'98 Human Factors in Computing Systems Conference*, pp. 357–358. Addison-Wesley/ACM Press.

Myers, B. A. (1989). User-interface tools: Introduction and survey. *IEEE Software 6*(1), 15–23.

Naumovich, G. and L. A. Clarke (2000, March). Classifying properties: An alternative to the safety-liveness classification. In *Proceedings of the 8th ACM*

Symposium on the Foundations of Software Engineering (FSE 8), pp. 159–168. ACM Press.

Nelson, R. A., L. M. Haibt, and P. B. Sheridan (1983, September). Casting petri nets into programs. *IEEE Transactions on Software Engineering 9*(5), 590–602.

Newman, W. M. and M. G. Lamming (1995). *Interactive System Design.* Addison-Wesley.

Olson, J. D. R. (1992). *User Interface Management Systems: models and algorithms.* Morgan Kaufmann.

Palanque, P. A., R. Bastide, L. Dourte, and C. Silbertin-Blane (1993). Design of user-driven interfaces using petri nets and objects. In C. Rolland, F. Bodart, and C. Cauvet (Eds.), *Proceedings of CAISE'93 (Conference on advance information system engineering), Springer-Verlag LNCS*, Volume 685.

Patch, K. (2001, November). Virtual reality gets easier. *Technology Research News.* `http://www.trnmag.com/Stories/2001/110701/Virtual_reality_gets_easier_110701.html`.

Paternó, F. (1995). *A Method for Formal Specification and Verification of Interactive Systems.* Ph. D. thesis, University of York.

Pausch, R. (1995, May). Alice: Rapid prototyping for virtual reality. *IEEE Computer Graphics 15*(3).

Petri, C. A. (1962). Kommunikation mit automaten. Schriften des iim nr. 2, Institut für Instrumentelle Mathematic. English translation: Technical Report RADC-TR-65-377, Griffiths Air Base, New York, Vol. 1, Suppl. 1, 1966.

Pettifer, S. R. (1999). *An operating environment for large scale virtual reality.* Ph. D. thesis, Manchester University.

Pfaff, G. E. (Ed.) (1985). *User Interface Management Systems.* Eurographic Seminars Series. Springer-Verlag.

Reisig, W. (1982). *Petri Nets.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

Roch, S. and P. H. Starke (1999, April). *INA Integrated Net Analyser (Version 2.2).* Humboldt-Universität zu Berlin.

Sastry, L., D. R. S. Boyd, R. F. Fowler, and V. V. S. S. Sastry (1998). Numerical flow visualization using virtual reality techniques. In *8th International Symposium on Flow Visualisation*, pp. 235.1–235.9.

Sherman, W. R. (1993). Integrating virtual environments into the dataflow paradigm. In *4th Eurographics workshop on ViSC.*

Sibertin-Blanc, C., N. Hameurlain, and P. Touzeau (1995). Syroco: A C++ implementation of cooperative objects. In G. Agha and F. DeCindio (Eds.), *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency*, pp. 51–62.

Smith, D. N. (1990). The interface construction set. In *Visual Languages and Applications*. Plenum Pub.

Smith, S. and D. Duke (1999a). Using CSP to specify interaction in virtual environments. Technical Report YCS 321, University of York - Department of Computer Science.

Smith, S. and D. Duke (1999b). Virtual environments as hybrid systems. In N. Dodgson and M. Austen (Eds.), *Eurographics UK 17th Annual Conference*, pp. 113–128. Eurographics.

Smith, S., D. Duke, and M. Massink (1999). The hybrid world of virtual environments. *Computer Graphics Forum 18*(3), C297–C307.

Smith, S. P., D. J. Duke, and J. S. Willans (2000). Designing world objects for usable virtual environments. In P. Palanque and F. Paternó (Eds.), *Workshop on design, specification and verification of interactive systems*, pp. 309–319.

Snowden, D. N. (1996). *AVIARY: A model for a general purpose virtual environment*. Ph. D. thesis, Manchester University.

Sommerville, I. (1996). *Software Engineering* (Fifth ed.). Addison-Wesley.

Spivey, M. (2000). *The fuzz manual* (second ed.). The Spivey Partnership.

Steed, A. J. (1996). *Defining Interaction within Immersive Virtual Environments*. Ph. D. thesis, Queen Mary and Westfield College, UK.

Sufrin, B. and J. He (1990). Specification, analysis and refinement of interactive processes. In M. Harrison and H. Thimbleby (Eds.), *Formal Methods in Human-Computer Interaction*, pp. 153–200. Cambidge University Press.

Systä, K. (1995). *A Specification Method for Interactive Systems*. Ph. D. thesis, Tampere University of Technology.

Thompson, M. R., J. D. Maxfield, and P. M. Dew (1999). Interactive virtual prototyping. In P. Hatton (Ed.), *Eurographics UK 16th Annual Conference*, pp. 107–120. Eurographics.

Upson, C., T. F. Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam (1989, July). The application visualization system: A computational environment for scentific visualization. *IEEE Computer Graphics and Applications*, 30–42.

Valette, R. (1986). Nets in production systems. In W. Brauer, W. Reisig, and G. Rozenbiz (Eds.), *Petri-nets: Applications and relationship to other models of concurrency, Advances in Petri-nets (part 2)*, Volume LNCS 255, pp. 191–217. Springer-Verlag.

van Bilion, W. R. (1988). Extending petri-nets for specifying man-machine dialogue. *International Journal of Man-machine studies 28*, 437–455.

van Schooten, B., O. Donk, and J. Zwiers (1999). Modelling interaction in virtual environments using process algerbra. In A. Nijholt, O. Donk, and B. van Oijt (Eds.), *12th Workshop on Language techology: Interaction in virtual worlds*, pp. 195–212. Twente University.

van Zijl, L. and D. Mitton (1991). Using statecharts to design and specify a direct-manipulation user interface. In *Proceedings of the Southern African Computing Symposium*, pp. 51–68.

VPL Research (1991). Virtual reality data-flow language and runtime system, body electric manual 3.0. Redwood City, CA.

Wasserman, A. I. (1985, August). Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering 11*(8).

Wellner, P. D. (1990). Statemaster: A UIMS based on statecharts for prototyping and target implementation. In *Human Factors in Computing Systems 1990 proceedings*, pp. 177–182. ACM Press.

Wieting, R. (1996). Hybrid high-level nets. In J. M. Charnes, D. J. Morrice, and D. T. Brunner (Eds.), *Proceedings of the 1996 Winter Simulation Conference*, pp. 848–855. ACM Press.

Willans, J. S. and M. D. Harrison (1999). Requirements for prototyping the behaviour of virtual environments. In M. D. Harrison and S. P. Smith (Eds.), *User-centered Design and Implementation of Virtual Environments Workshop*, pp. 7–13.

Willans, J. S. and M. D. Harrison (2000a). A 'plug and play' approach to testing virtual environment interaction techniques. In R. van Liere and J. Mulder (Eds.), *6th Eurographics Workshop on Virtual Environments*, pp. 33–42. Springer-Verlag.

Willans, J. S. and M. D. Harrison (2000b). Verifying the behaviour of virtual environment world objects. In P. Palanque and F. Paternó (Eds.), *Interactive systems design, specification and verification*, pp. 65–77. Springer-Verlag LNCS 1946.

Willans, J. S. and M. D. Harrison (2001a). Prototyping pre-implementation designs of virtual environment behaviour. In R. Little and L. Nigay (Eds.), *Eighth IFIP Working conference on Engineering for Human Computer Interaction (EHCI'01)*, pp. 91–113. Springer-Verlag LNCS 2254.

Willans, J. S. and M. D. Harrison (2001b, August). A toolset supported approach for designing and testing virtual environment interaction techniques. *International Journal of Human-Computer Studies 55*(2), 145–165.

Willans, J. S., M. D. Harrison, and S. P. Smith (2000). Implementing virtual environment object behaviour from a specification. In V. Paelke and S. Volbracht (Eds.), *User Guidance in Virtual Environments*, pp. 87–97. Shaker Verlag, Aachen, Germany.

Willans, J. S., S. P. Smith, and M. D. Harrison (2001a, April). Modelling and verifying virtual environment behaviour. In G. J. Doherty, M. Massink, and M. D. Wilson (Eds.), *Continuity in Future Computing Systems*, pp. 75–79. Technical report RAL-CONF-2001-001.

Willans, J. S., S. P. Smith, and M. D. Harrison (2001b). Using scenarios to identify the design requirements of virtual environments. Technical Report YCS 333, University of York.

Wüthrich, C. A. (1999). An analysis and a model of 3D interaction methods and devices for virtual reality. In D. Duke and A. Puerta (Eds.), *Design, Specification and Verification of Interactive Systems '99*, pp. 18–29. Springer-Verlag.

Yamamoto, K. (1996). 3D-visulan: A 3D programming language for 3D applications. In *Pacific Workshop on Distributed Multimedia Systems (DMS96)*, pp. 199–206.

Zeleznik, R. C., K. P. Herndon, and J. F. Hughes (1996). SKETCH: An interface for sketching 3D scenes. In *Computer Graphics Proceedings SIGGRAPH'96*, pp. 163–169.