University
of
Alberta

InTml Concepts

by

Pablo Figueroa
and J. H. Hoover
and Pierre Boulanger

Technical Report TR 04-06
May 2004

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Abstract

In this document, we describe from different viewpoints our model for Virtual Reality applications. We give first an informal description of the basic concepts, the execution model, and some examples of modeling for devices, behaviors, and media content elements. Second, we give a more in depth description of each concept, the way they are related to each other, and their XML syntax. Finally, we present a more formal description in the Z language [12] that clarifies the semantics of the model and generalizes some of the concepts presented in the two previous presentations.

1 An Informal Introduction to InTml

We consider a VR application a data flow of interconnected filters, described in a language called InTml, the Interaction Techniques Markup Language. Filters are the building blocks that describe the standard connections for any of the following entities: input or output devices, interaction techniques, object behavior, animations, geometric objects, and other media objects. Details about gathering information from devices or about object behavior code are described in a lower level of abstraction through the use of programming languages. Also, geometry or other media types related to VR objects are produced in any of the available tools for that purpose, such as Maya [2], 3D Max [1], or Blender [4]. InTml is then an integration language for all elements involved in VR applications. It enables the designer to concentrate on the architecture of the application, without dealing with too many details. As an example, while dataflow–based languages such as VRML focuses on description of geometry and animation, InTml focuses on the integration of application–specific behavior, object behavior, and events from input devices. Geometry is something that is described at a lower level, in a loadable format, and InTml refers to it as a reference to an object. The same can be applied to sound or haptic content.

A *filter* represents any device, interaction technique, behavior, or content in a VR application. Its interface is defined in terms of input and output ports, which are the type of events it can receive or produce, respectively. Some input ports can be considered parameters, or ports that will receive information only once at application startup. A filter can have an internal state, which is important in order to model complex filters. However, we do not include this description at the architectural level due to its low–level nature. Figure 1 shows a way to represent a filter, `SelectByTouching`, with input ports on the left of a box and output ports on the right. In this

particular example, its output port is a selected object from the scene, and its input ports are the VR object used as hand representation, the current position and orientation of such an object, the scene of objects to pick from, and the events that inform about added or deleted objects from the scene. Note that the input ports for the hand representation and the scene can be considered parameters of such a filter, i.e. they will not change once they are assigned.

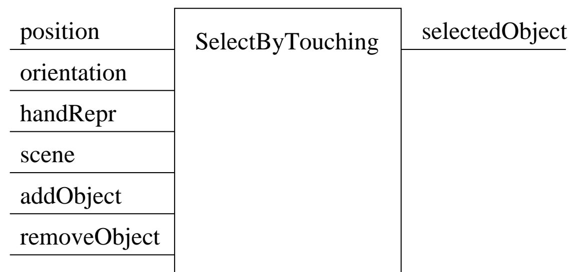

Figure 1: Select by Touching. An Example of a Filter

The computation of a filter is divided in three main stages:

- Data collection. All information generated in a certain time interval is collected. This stage is considered a preprocessing stage, in which filters select and manipulate the information they have received, in order to prepare for the next stage.
- Processing. In this stage a filter executes, given the collected input information and its internal state. Output information is generated, but not propagated
- Output propagation. Output information is propagated to all interested filters.

VR objects represent identifiable pieces of content in the virtual environment: elements that can be seen, heard, or touched by the user. An *object holder* is a filter that associates one object to a set of desired changes. It is drawn with an additional decoration for a special input port, `object`, that receives objects to be hold (a small rectangle between the port and the object

holder) [1]. Once an object **O** is associated to an object holder, by sending such an object through the port `object`, all information coming to the object holder is redirected to **O**. In the same way, outputs from **O** are sent to the filters connected to the object holder. An *application* is a set of interconnected filters, that meet certain user requirements. Figure 2 shows a simple application, which allows a user to move a virtual hand with a tracker and touch virtual objects. In this example, a device (`handTracker`) gives position and orientation information to a selection technique (`SelectByTouching`) and an object holder (`handRepresentation`). The actual object representing the user's hand (`handRepr`) is given to `SelectByTouching` for collision detection, and to `handRepresentation` for changing the object. Once a collision is detected, the collided object is passed to `Feedback`, which changes the color of the object. Filters and applications are independent of any particular software framework and hardware, so the designer does not have to be limited by platform specific elements, and the developer is free to reorganize the implementation in order to improve the performance of the application in a particular platform.

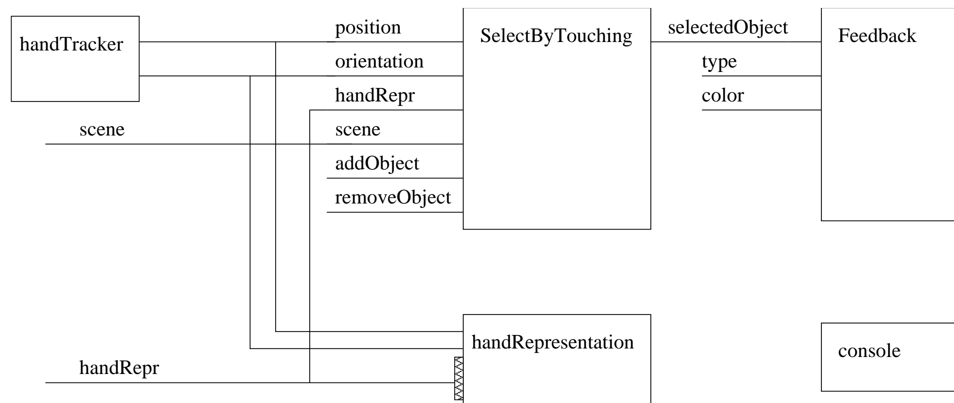

Figure 2: Simple Application. Touching Objects With a Virtual Hand.

An *input device* is a filter with just output ports that sends events of a certain type to the dataflow. An *output device* is a placeholder that describes where the output of the application will be displayed – it is internally related to the VR objects, but the details are hidden to the VR designer.

In order to reduce the complexity of an application, subsets of interconnected filters can be encapsulated in a *composed filter*. A composed filter

[1] For more details about object holders, see Section 4.11

represents a complex behavior in an application, that might be treated as a unit and reused in new applications. Composed filters can be used to encapsulate all necessary details of an interaction technique. As an example, Figure 3 shows two views of the Go-Go interaction technique [9] – an interaction technique to lengthen the user's virtual arm for reaching distant objects. The left image shows enough detail to allow VR designers to use such an interaction technique, while the image at the right shows all the filters and objects involved.

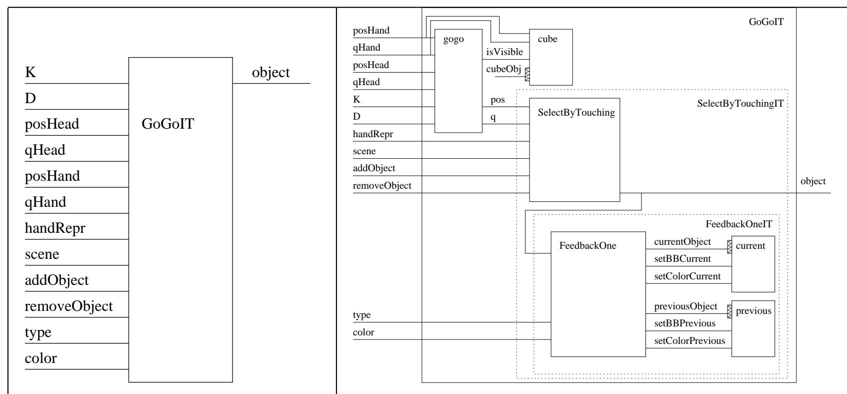

Figure 3: The Go-Go interaction technique. General and detailed views.

2 Execution Model of an InTml Application

An InTml application is executed as a sequence of identical steps, each one composed of four stages. The actual execution time of each step and the resulting application frame rate are considered implementation dependant, since they vary according to the computational power of the particular hardware platform and the particular method for translating InTml to such a target environment. We assume that a minimum frame rate per device is known, and that the translation process from InTml to code takes into account such rates in order to provide a usable virtual experience. The semantic model of InTml defines which filters are executed in each step, no matter their order, and which results are expected. It is up to the InTml implementation to execute such filters fast enough to reach the target speed of the VR application.

Figure 4 shows the stages in an InTml execution step. The activities performed during each stage are the following:

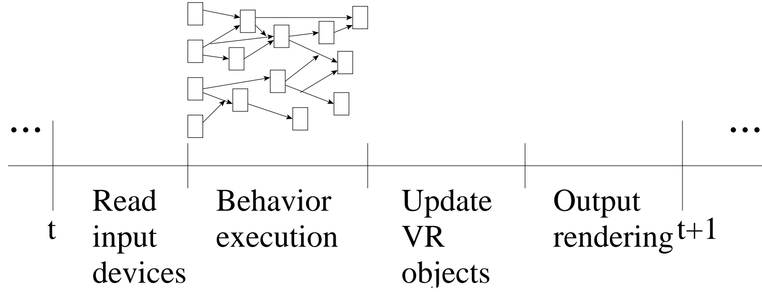

Figure 4: Execution Model of InTml Applications between two rendering steps.

- Device reading: Events from active devices are read during this period of time. All events during such a period are considered simultaneous.
- Data flow execution: Events received in the previous stage are fed to the dataflow, which propagates the events throughout reachable filters. All changes to objects requested by filters are queued, and they will be executed in the following stage.
- Object updating: All changes requested in the previous stage are considered, and the selected ones are executed. Each object type should implement its own conflict resolution policy, when several conflicting changes are requested. For example, if an object receives several position changes, it might decide to either execute any of the changes at random, or execute the average of the requested changes. This objects' feature is important due to the fact that the order of execution of filters behind any object is unknown, since InTml implementations with more computational power might execute filters in parallel, and each object might receive changes from more than one filter. However, a selection policy defines if the dataflow is deterministic or not [2].
- Object rendering: Changes to objects are shown to users, in each one of the output devices available. This stage is usually transparent to users of modern graphic APIs, such as Performer and Java3D, and usually supported by dedicated hardware.

[2]The average policy results in a deterministic execution, whereas a random selection policy will produce a non-deterministic result.

The non-deterministic feature of some object updating policies deserves one more comment. While this is in general an undesirable characteristic, created by the availability of more than one input event of a certain type, in practice it is not very noticeable. If we assume that all input events have been generated from a certain device, or by several filters that indirectly received such events, and that filters compute “smooth” functions, the spatial and temporal coherence of the input events will guarantee certain proximity of the values computed from each input event. In practice, when input values are close enough, users will not notice differences related to the chosen value.

Stages can be parallelized or pipelined, so it is possible to get the best performance from each platform, with only one application description. An example is shown in Figure 5. Additional threads are dedicated to device event gathering, filter execution, or object updating. There are some points where threads should be synchronized, but in general this approach permits a better application throughput and more complex computations.

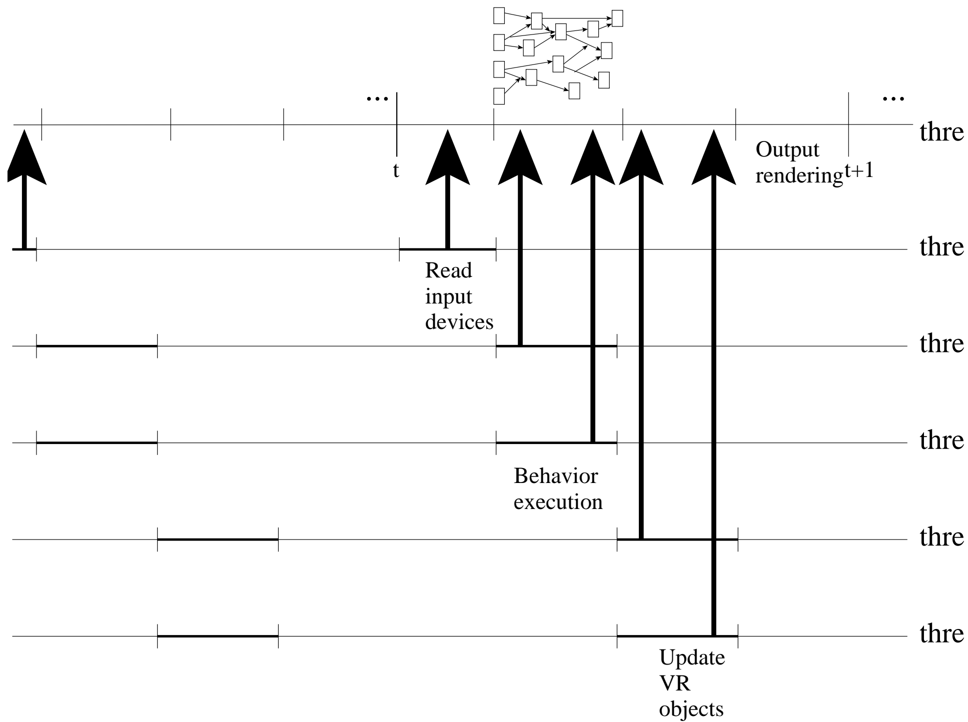

Figure 5: Extended InTml Execution Model.

Information in the dataflow and the state of VR objects are considered immutable in a particular time frame. For this reason all filters will see the same state of an object inside a computation frame. An InTml application describes the first two stages — devices to read and data flow —, and the other two are hidden at this level from the designer. We assume an implicit connection between media objects and output devices, in order to allow devices to render the entire state of the world.

From the point of view of designers of VR applications, an InTml application is a set of modules that has to be implemented on top of a foundation framework, and certain rules of execution have to be taken into account. The designer's work is divided between the definition of new filters, reuse of previously defined ones, and definition of applications. Designers collaborate with developers of VR applications, whose main job in an InTml–based environment is to develop the inner code in the filters.

2.1 Implementation issues of the Execution Model

As we have mentioned, the actual execution time of a particular InTml application depends on the implementation in which the code is running on, since all concepts related to time are abstracted from the general description we have presented. In particular, we assume that all information generated at a particular time simultaneously arrives to interested filters. The purpose of this abstraction is to hide complexity to designers, whose work is to describe a solution no matter the hardware characteristics of the platform they have available. An implementation of InTml has to address the following issues related with time:

- Network delay. This delay is due to the physical characteristics of the network technology used between the computers of a specific platform. Once it is measured, it can be considered constant, no matter the network load.
- Processing delay. This delay is due to the inherent time required to compute results inside a filter. It depends on the amount of input at a particular time, and the algorithms used in the implementation of a filter. It is possible to create models for particular filters in order to predict this delay and take decisions regarding frame execution.
- Latency. This time is caused by the load in the network. It is difficult to predict, but it is possible to define a maximum waiting time, used to discard messages that arrive late.

A simple implementation of InTml, in which there is only one computer receiving data and processing it, does not require to handle the previous concepts. In a more complex setup, it is possible to create a model based on the previous time definitions in order to handle such delays. A simple way to deal with such issues, based on the framework for multi-modal VR applications presented in [13], requires that every filter waits for a period of time before collecting information from input ports. Such a period of time assures that all simultaneous events are received before its processing, so a filter can execute in the normal way without extra considerations.

3 InTml Examples

InTml is useful to represent devices, interaction techniques, content behavior, and applications that combine all these elements. Let us illustrate these concepts with some simple examples. A real design of such elements might be different in order to support a more modular and reusable structure, but the presentation here aims at illustrating instead of optimizing for reuse.

One can start with a common input device, a three–button mouse. A graphical representation of a mouse with three buttons in InTml is shown in Figure 6. We show the information one can get as separate output ports: x position, y position, both coordinates together as a mouse move, and events for the actions of pressing and releasing each button[3]. Other representations and events are possible. For example, buttons can be represented as separate devices inside a mouse, and events for clicking or double clicking a button can be generated, separate from the press and release events. Any element can have redundant output ports. In this way, other elements can choose what is the right type of information they are interested in.

[3]Types for each event are not shown in this diagram, but each port only allows events of a certain type.

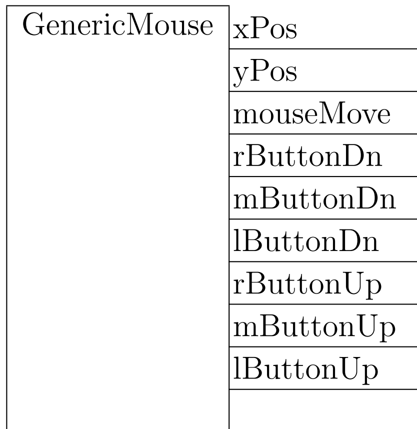

Figure 6: A Simple Representation of a Mouse in InTml.

A more advanced device is the 6DOF wand tracker from InterSense. It not only provides position and orientation in 3D space, but it also has a small joystick and four buttons. Such a wand can be represented in InTml as it is shown in Figure 7. In this case, buttons only generate one type of event, when they are clicked. It is also possible to have events when a button is pressed or released, as in the mouse example.

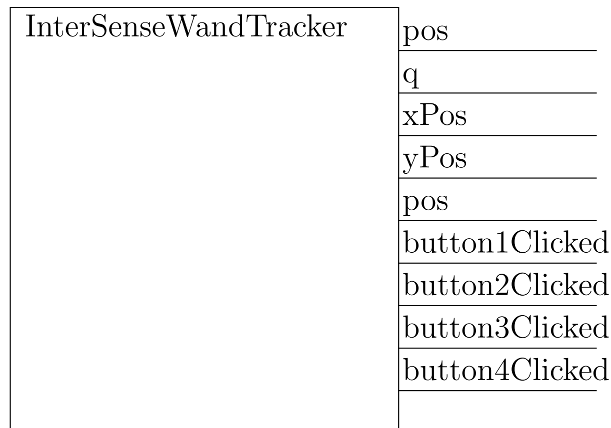

Figure 7: An InTml Representation of the InterSense Wand.

An example of a 3D selection widget is the ring menu [6]. A ring menu shows a set of objects in a ring that can be rotated along its 3D axis.

Extra geometry in the middle of the ring makes the selected object more visible, which is between the user's viewpoint and the ring's axis. An InTml representation of a ring menu is shown in Figure 8.

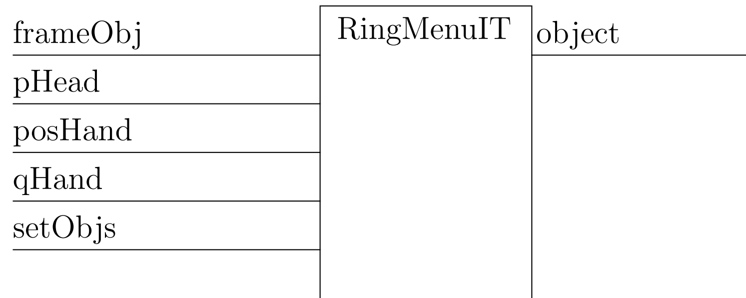

Figure 8: An InTml Representation of the Ring Menu.

The geometry for the frame can be given through the `frameObj` port. The position of the user's head is also required, with the position and orientation of a hand tracker. Its output is the selected object, if it has changed since the last selection.

Barrilleaux [3] describes several interaction techniques for 3D manipulation in standard desktops. For example, object movements can be performed in relation to the displacement of the mouse in the display plane, or the projetion of such a movement over the virtual "floor". The latest option is called world–related–movement, and it can be represented in InTml as it is shown in Figure 9. The manipulation technique receives a position `p` in display coordinates, the plane that represents the floor in the current scene, the object to be moved and the current user's viewpoint. The result is a new 3D position for the object.

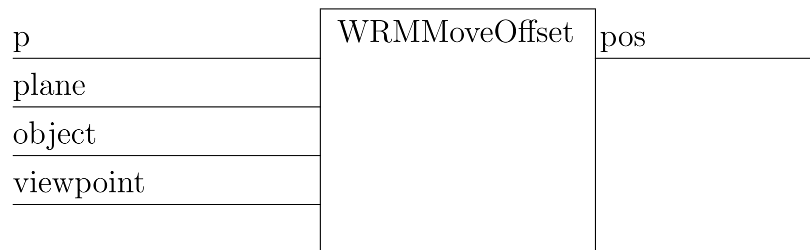

Figure 9: An InTml Representation of World–Related Movement.

The basic behavior for objects in the world can also be represented in InTml. Figure 10 shows `VRObject`, which encapsulates basic behavior of an interactive object. Position, orientation, and scale can be changed, in a separate way or at once by a transformation matrix. Parts can be added and removed, a bounding box can be drawn around the object, and its main color can be changed. An object can also inform any interested filter of any of the previous modifications.

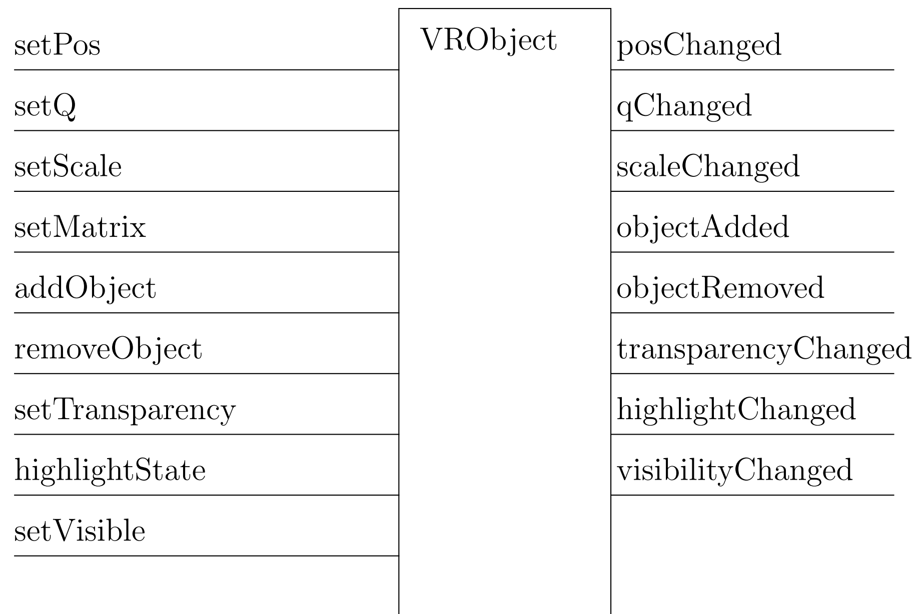

Figure 10: An InTml Representation of an Interactive Object.

4 InTml Ontology and XML representation

We describe in this section the concepts in InTml and their relationships, together to the XML syntax we have created for them. Figure 11 shows two views of the concepts (in rectangles) and relationships (as arrows) in InTml. A dashed box represents an abstract concept with no instances, but useful to describe relationships of several other concepts related to it by an `isA` relationship. The following paragraphs describe these concepts and relationships in more detail. This description is related to the current

XML implementation, according to its DTD definition. The description in Section 5 separates from current implementations and describes a more general approach, with more solid semantics. XML fragments in this section use the following conventions: String values are written as `value`, optional attributes or entities are enclosed in “[ ]”, optional values are written as *opt1/opt2/opt3*, where *opt1* is the value by default.

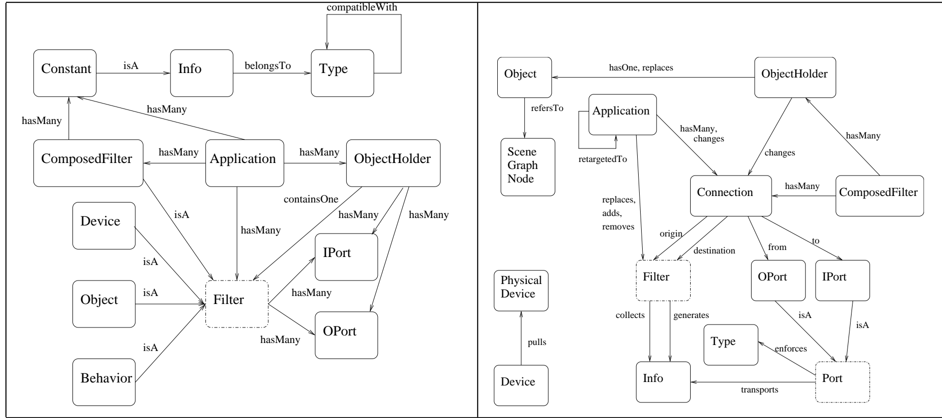

Figure 11: Entities and relationships in InTml.

4.1 Port

A port is a filter’s connection point. It is called an input port (iport) if receives information, and an output port (oport) if propagates information generated from the filter.

The main purpose of a port is to transport information from one filter (origin) to another (destination), which are connected by an oport and an iport, respectively. The type of information flowing through a port should be compatible with the port type.

We have not designed any XML representation for this abstract entity.

4.2 IPort

An input port is an entry point for information of a certain type. Its declaration inside a filter class has the following syntax:

```
<IPort id="aName" type="aType" [defValue="stringRep"]  [policy="aPolicyName"]
```

```
        [isArray="false/true"]  [typeArray="static/dynamic"]
        [maxArray="aNumber"]  >
  <ShortDesc></ShortDesc>
  [<Description></Description>]
</IPort>
```

An input port is identified by a name. The information received by the port should have a compatible type with the input port's type. An input port might have a default value, which corresponds to the first value received. The attribute `policy` refers to the name of the policy management for multiple events of the same type at the same time. It is optional, with implementation–dependant values. It is also assumed that any implementation has one policy by default [4]. As a way to reduce the syntax of some filter classes, and also as a way to create multiplexors and mergers, we use the following attributes to describe an array of ports with the same basic syntax. The attribute `isArray` is true when the node is an array of input ports, in which case the next two attributes can be defined: `typeArray` that says if there is a fixed (static) or variable number of ports in the array, and `maxArray` that defines the maximum number of ports if the array is static. For example, an IPort declaration such as

```
<IPort id="ip1" type="int" isArray="true" typeArray="dynamic"/>
```

will allow references to ports `ip1[2]` or `ip1[15]`, whereas

```
<IPort id="ip1" type="int" isArray="true" typeArray="static" maxArray="10"/>
```

will allow just the first one.

Any input port might have a short and a long description. It can also be associated to an object holder, in which case it is not explicitly declared (i.e. there is no declaration of a type for an object holder), but it is deduced from the type of the first object connected to the object holder [5].

References to ports are used inside connections. This syntax will be described later in this chapter.

[4]For example, in a Java3D implementation we did we use two names: ALL and ANY to describe a policy that takes into account all events, or one at random, respectively.

[5]More details in Section 4.11.

4.3 OPort

An output port is a channel of information from a filter. A filter can have several output ports, some of them redundant [6], with an entire set of information it can produce. Its XML structure is similar to the one of an input port:

```
<OPort id="aName" type="aType"
       [isArray="false/true"]  [typeArray="static/dynamic"]
       [maxArray="aNumber"]  >
  <ShortDesc></ShortDesc>     [<Description></Description>]
</OPort>
```

The attributes `id`, `type`, `isArray`, `typeArray`, and `maxArray` have the same meaning as the ones in an IPort. Output ports do not have a `defValue`, since the only information that goes out of a filter is the one computed from the information in its input ports, at any period of time. Neither does it have a `policy`, since all information generated is sent to interested filters, and it is up to their input ports to decide a policy for several simultaneous events. In the same way as an IPort, it may have some textual description associated and it may be associated to an ObjectHolder.

4.4 Filter

A Filter is the minimum unit of computation in InTml. It defines a set of input ports that receive events from other filters, and a set of output ports that sends computed values from the received events at any time. It also can have an internal state, hidden from the external definition in terms of ports [7]. A filter can be part of composed filters, applications, or object holders. It is involved in connections by binding an output port of a filter with an input port of another. There are three concrete flavors of filters: devices, objects (or media), and behavior (or interaction techniques). It performs the following three tasks at every execution step: collection of events from its input ports, processing of this information, and generation of events to be send through output ports. It does not directly have instances, but the ones from its subclasses (Device, Object, Behavior).

[6] In some cases it is useful to count with several ports with several representations of the same information, i.e. a matrix and a quaternion representation of a rotation.

[7] Such an internal state is usually documented for completeness reasons

4.5 Device

A device [8] is the simplest filter class. It represents a physical unit that produces certain type of information. It usually gets its information by pulling it from a physical device, but the relationship is not necessarily one to one, i.e. a logical device can pull information from several physical devices, or a physical device can be associated to several logical ones. Devices are instances of device classes, declared as follows:

```
<DeviceClass id="aName" >
  <ShortDesc></ShortDesc>
  [<Description></Description>]
  [<Implements classId="aClassName"/>]
  [<IPort>...</IPort>]
  [<OPort>...</OPort>]
  ...
</DeviceClass>
```

A device class has a name as identifier, and it contains the input and output ports of every instance of such a class. The special tag `Implements` allows an inheritance–like relationship between device classes: The class contains all input and output ports of the implemented one. An instance of a device class can be created inside an application as follows

```
<IDevice id="aName" type="aType"/>
```

or

```
<ODevice id="aName" type="aType"/>
```

The IDevice emphasizes the fact that the device will produce input information for other filters in the application. The purpose of ODevice is to mention output devices in the environment. If a device both produces and consumes information at the same time, it should be considered as an IDevice.

4.6 Physical Device

A physical device is a physical entity that produces information to be read by the computer. It is represented in an InTml application as one or many

[8] Sometimes is called logical device, in contrast to a physical device, defined below.

instances of class `Device`. Special tuning of physical devices (callibration procedures, startup, alignment, particular method for accessing information from it) are considered out of the scope of InTml. In this way, InTml users concentrate on what type of information they can use from devices, instead of both information and setup. There is no visibility of physical devices in InTml, appart from a tacit correspondence with logical devices.

4.7 Object

An object is a filter that affects the media involved in the VR application, i.e. a piece of geometry, a sound effect, a haptic effect, etc. In the case of geometry, media is usually represented with a scene graph. InTml objects are built on top of scene graph nodes, and are used inside an InTml application to make changes in particular nodes in the scene. During an execution step, an object queues all changes that filters request. At the end of the execution step, once all filters involved in its input have been executed, an object executes the requested changes according to the policies of its input ports. If the object report changes through its output ports, they will be noticed in the next execution step. In this way it is assured that all filters can read the same state of an object during one execution step, no matter the order of execution. An object type is defined as follows:

```
<ObjectClass id="aName" >
  <ShortDesc></ShortDesc>
  [<Description></Description>]
  [<Implements classId="aClassName"/>]
  [<IPort>...</IPort>]
  [<OPort>...</OPort>]
  ...
</ObjectClass>
```

This structure is identical to the one for device classes. An instance of an object can be created inside an application or a composed filter as follows:

```
<Object id="aName" type="aType" [fileName="aFile"]
        [primitive="Box/Cone/Cylinder/Ellipse"] />
```

Every object has an unique identifier (aName), declares the name of its object class (aType), can load a piece of geometry or other loadable type of content from a file (aFile), or it can correspond to one of 4 simple shapes (*Box/Cone/Cylinder/Ellipse*).

Objects can also be sent to other filters as information, and in particular to object holders. The following syntax is used for this task:

```
<Binding iE="_self" iP="objectName"
         oE="aFilter" oP="anIPort" />
```

The special identifier _self refers to the current composed filter or application that contains the object called objectName. The filter aFilter will receive objectName through the port anIPort. Objects can be plugged to object holders through a special port called object.

4.8 Scene Graph Node

A Scene graph is the main data structure for geometry processing in modern graphic APIs, such as Performer [11] or Java3D [7]. An object in InTml can refer to an element in such a hierarchy, in such a way that it hides the complexity of geometry rendering inherent to the scene graph structure but at the same time allows interaction with such elements. Figure 12 shows the relationship between nodes in the scene graph and InTml objects. There could be an implicit dependance between InTml objects, since they can refer to scene graphs nodes that depend on each other, i.e. objects A and B in the figure. There are ways to make this relationship explicit, for example, by allowing InTml objects to replicate such a hierarchy. However, in the general case, we consider this relationship out of the scope of the InTml description.

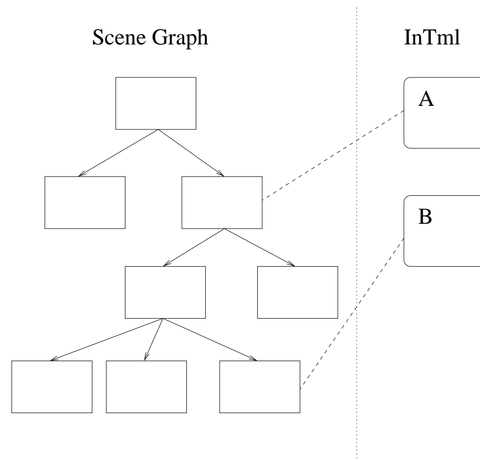

Figure 12: Relationship between scene graph nodes and InTml objects.

4.9 Behavior

Behavior is represented by filters in InTml. A filter can represent a piece of computation in an InTml application. It represents interaction techniques, special object behavior, animations, or application specific behavior as filters in the dataflow. In its simplest form, the type of a filter in InTml is declared as follows:

```
<FilterClass id="aName" >
  <ShortDesc></ShortDesc>
  [<Description></Description>]
  [<Implements classId="aClassName"/>]
  [<IPort>...</IPort>]
  [<OPort>...</OPort>]
  ...
</FilterClass>
```

This has the same structure of a DeviceClass or an ObjectClass. Filter instances can be created in applications or composed filters with the following statement:

```
<Filter id="aName" type="aType"/>
```

As in devices and objects, a filter is identified by a name and a type.

4.10 Connection

A connection defines a relationship between devices, behavior, and media content. It is the only way to pass information from one software component to another, and the only visible relationship possible at runtime [9]. The XML syntax of a connection is as follows:

```
<Binding iE="aFilter" iP="anOPort" [iI="anIndex"]
         oE="aFilter" oP="anIPort" [oI="anIndex"] />
```

A connection is uniquely identified by an origin element (`iE`), an output port in such an element (`iP`), an optional index in the output port declaration (`iI`), a destination element (`oE`), an input port in such an element (`oP`), and an optional index for the input port (`oI`). Connections appear inside applications and filter classes in order to describe the dataflow between elements.

[9] Devices are related to physical devices and objects to scene graph nodes, but such relationships are considered out of the scope of InTml.

4.11 ObjectHolder

An object holder is a way to define placeholders with a certain structure of connections in a dataflow, places where media content elements can be plugged, executed, and changed. Section 5 will describe a more general concept, a filter holder, allowed to hold any type of element. Current implementations of InTml are limited to hold just objects, but this constraint can be avoided in the future, by following the semantics of filter holders [10].

The XML syntax for an object holder is as follows:

```
<ObjectHolder id="aName" />
```

We can notice that an object holder does not have a type in its declaration. Our current implementation of object holders finds out its ports by “copying” the ports of the first object it gets to hold. Additionally, an object holder will always have two additional ports: an iport called object that allows receiving new objects, and an oport called objectChanged that will inform interested filters about changes in the contained object. If such an object is replaced, subsequent objects will be connected to the ports already defined at the object holder, provided that ports are compatible [11]. An example of this mechanism is shown in Figure 13. When the application starts and an object holder has not received any events, there are only two ports declared: one for receiving object events, one to inform changes in the contained object. Once an object A is connected, the object holder copies its definition and adds to itself the corresponding ports. Later on, when A is replaced by an object B of a different class, just the compatible ports are connected, i.e. ports r and s will not be connected. If A is embedded again in the object holder, it will be conneted as it is was before.

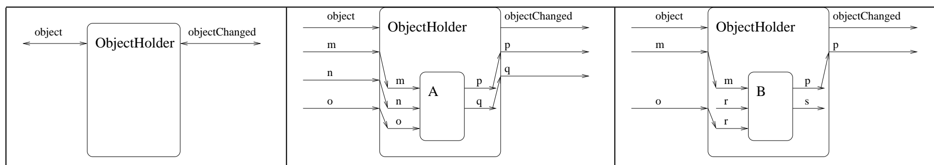

Figure 13: Different states of an Object Holder during execution.

[10] Filter holders are defined later in this chapter.
[11] Currently, compatibility between ports is defined by their names.

4.12 Events in the Dataflow and Types

Every piece of information flowing through the dataflow is an event, or *Info* in Figure 11. Pieces of information have a type associated to it and its value could be atomic or composed. A value is considered atomic if it is implementation–dependant and do not have a complex representation in InTml. Note that a complex structure such a quaternion [10] can be treated as atomic if there is no definition of its structure in InTml. A value is composed if they correspond to objects defined in InTml. Types can have an implicit compatibility relationship, i.e. inheritance in object–oriented languages. However, this relationship is not declared in the current implementation at the level of InTml, but at the implementation level. Type declarations are either implicit, when they appear as types of ports or constants, or explicit, when they are described as an `ObjectClass`.

All events are time–stamped, and all events received at the same frame from devices and propagated through the dataflow are considered simultaneous. Events are immutable, so their values can not be changed by filters. An event can be propagated to several filters (fan-out), so several filters can refer to the same Info.

4.13 Constant

A constant is a value of a certain type. Constants are used to give a default initial value to a filter, through a particular input port. A constant is declared as follows

```
<Constant id="aName" type="aType" value="aValue"/>
```

A constant has an identifier and a type. Its value is a string–based representation of its value inside the InTml execution. Constants can be pushed through input ports, so its value will be the first value received by a port. Constants can be declared inside applications and composed filters, and sent to filters through a particular input port. The following is an example, in which a filter `f` receives a constant `c1` through its input port `ip`. We use the special identifier `_self` in order to refer the constant `c1` defined in the current context.

```
<Binding iE="_self" iP="c1" oE="f" oP="ip" />
```

4.14 ComposedFilter

A composed filter is an InTml construct that allows designers to hide complexity by encapsulating a piece of an InTml program as a simple filter in the environment. A composed filter describes a subset of filters, objects, object holders, constants, and connections that execute certain tasks. We use the same XML element to declare simple and composed filters, but a composed filter can include other elements, such as:

```
<FilterClass id="aName" >
  <ShortDesc></ShortDesc>
  [<Description></Description>]
  [<Implements classId="aClassName"/>]
  [<IPort>...</IPort>]
  [<OPort>...</OPort>]
  ...
  [<Filter>...</>]
  [<Object>...</>]
  [<ObjectHolder>...</>]
  [<Constant>...</>]
  [<Binding>...</>]
  ...
</FilterClass>
```

A composed fiter describes a dataflow with a certain interface given by its own input and output ports. IPort and OPort elements should be connected to ports of internal entities through `Binding` statements. For example:

```
<FilterClass id="ComposedFilter1" >
  <ShortDesc>An example of composed filter</ShortDesc>
  <IPort id="ip1" type="Type1"/>
  <OPort id="op1" type="Type1"/>
  ...
  <Filter id="f1" type="Filter1"/>
  <Object id="obj1" type="Object1" fileName="f1"/>
  <Binding iE="_self" iP="ip1" oE="f1" oP="ip"/>
  <Binding iE="obj1" iP="op" oE="_self" oP="op1"/>
  ...
</FilterClass>
```

Assuming that filter `f1` has an input port `ip`, and that object `obj1` has an output port `op`, the previous example connects the input and output of

the composed filter to its internal structure. Note the use of _self to refer to the composed filter, when necessary.

An instance of a composed filter can be created inside an application or inside other composed filters with the same syntax used for normal behavior.

4.15 Application

An application element in InTml describes a dataflow of filters that accomplish certain tasks. The XML syntax for an application is the following:

```
<App id="aName" >
  <ShortDesc></ShortDesc>
  [<Description></Description>]
  ...
  [<IDevice>...</>]
  [<ODevice>...</>]
  [<Filter>...</>]
  [<Object>...</>]
  [<ObjectHolder>...</>]
  [<Constant>...</>]
  [<Binding>...</>]
  ...
</App>
```

An application is identified by a name. It contains all the required elements for a dataflow in InTml: input devices, output devices, filters, objects, object holders, constants, and connections.

Two important development tasks are involved with applications: initial definition, and retargeting to other hardware platforms. The initial definition creates the required filter classes and connections to satisfy certain set of user requirements in a particular platform. A retargeting process consists of deriving new applications from a previously defined one, by replacing elements for the more suitable ones in a new hardware platform. Such a change starts in devices, and changes can be propagated to filters, constants, objects, object holders, and connections.

4.16 Other Language Features

The XML representation of InTml has some extra constructs, which are useful at design time or for documentation purposes. These concepts are the following: `Package`, `Import`, `Overrides`, `Platform`, `Index`, and `PaperRef`.

Package allows the creation of name spaces in InTml. The name of a filter class can be used without qualifiers by classes or applications of the same package. Otherwise it should be fully qualified, or its package should be imported. The sentence Import allows any package to refer to filter classes in a non fully–qualified form. The syntax for these two elements is as follows:

```
<Package id="aName">
  [<Import id="aPackageName">]
  ...
  [<DeviceClass>...</DeviceClass>]
  [<FilterClass>...</FilterClass>]
  [<ObjectClass>...</ObjectClass>]
</Package>
```

A package gives a prefix to all classes declared inside of it. Classes can be used without a qualifier inside classes of the same package. Outside the package, classes can be named either by its fully qualified name, or by importing its package. When an package is imported, all its classes can be used with their simple names. Conflict names in the current implementation are resolved by taking the first match to a filter class in the list of imported packages.

An application is not defined inside a package declaration, but instead it can be created with its fully qualified name. For example:

```
<App id="a.b.c" >
  ...
</App>
```

defines application c, in the package a.b.

Overrides is a special relationship between applications. If application B overrides A, the resulting application is the set operation $(A \setminus B) \cup B$. In other words, B has all objects, behaviors, devices, and connections in A that are not defined in B, plus all elements of B. In this way, B replaces elements in A with new definitions. This mechanism has been used in order to create default values for an application: A corresponds to the basic implementation of an application, common to other ones, while B redefines only some elements and adds many more.

Platform is a special construct that groups together sets of devices. It is planned to be used for automatic retargeting of applications, but their semantics is not fully defined.

`Index` in an extra tag that applications and class declarations have for documentation purposes. Index classifies an application or a class under an index name. Several indexes are allowed, so an element can be classified under several criteria. For example, the following declaration:

```
<FilterClass id="SelectByTouching">
  <Indexes>
    <Index id="first" value="intml.selection.details"/>
    <Index id="papers" value="_hidden"/>
  </Indexes>
  ...
</FilterClass>
```

classifies the class `SelectByTouching` under two criteria: first, with value `intml.selection.details`, and `papers` with the special value `_hidden`, which is used to officially hide an element from a particular classification. Documentation tools can use this information to create browsers of elements.

5 A Formal Description of InTml in the Z Language

A Virtual Reality application is described here as a flow of messages in which input messages or events are read from devices and propagated to all the functionality of an application, represented in terms of filters. This dataflow might be executed in parallel or pipelined, complex functions can be encapsulated in order to reduce complexity, information from several simultaneous devices can be received, and all simultaneous filters are guaranteed to see the same world state. A formal description of this architecture based on the Z language [12] is given here, and it is shown how these special characteristics are accomplished, independently from a particular implementation. Such a formal description serves as a blueprint for new implementations of this architecture, a reference that explains the semantics of InTml, independently from any particular implementation. Finally, we analyze the differences between our presentation and theirs, and some of the lessons learned from this specification exercise.

5.1 Basic Concepts

A *filter* is a processing element in the dataflow. Names for filters are unique in the system, and belong to the set L. Filters are connected by channels, whose names are also unique and belong to the set C. A *channel* uniquely describes a particular output of a filter [12]. *Messages* are indivisible chunks of information that go through channels and belong to the set M.

A *stream* is a function that relates a channel name with a particular sequence of messages. Messages in a stream are ordered in bags, and each bag represents a set of messages received in a particular interval. In this way, more than one message can be received in a particular interval, messages of different intervals can be identified, and the order of messages in the same interval is not relevant, a very useful property for non-synchronous, parallel execution of components. We use the property described by Philipps and Rumpe [8] that it is impossible to distinguish between a function that computes its output given the history of its input so far, and one that selects the current output from a sequence of precomputed results. This gives uniformity between our presentation and theirs in order to reuse some of their specification styles.

Stream

$c : C$
$m : \mathrm{seq}(\mathrm{bag}\, M)$

We use the operator $\downarrow$ [13] to refer to the messages in a stream up to a certain moment, and it is defined in several contexts. In its simple form, $st \downarrow i$ is the operation of extracting a sequence from a stream st with only the first i elements.

$[X]$

$_ \downarrow _ : \mathrm{seq}\, X \times \mathbb{N} \longrightarrow \mathrm{seq}\, X$

$\forall\, st : \mathrm{seq}\, X;\ i : \mathbb{N} \bullet$
$st \downarrow i = (1 \mathinner{..} i) \lhd st$

We also use this operator at the level of streams

[12] Channels are a concept defined by [8] which facilitates connections between filters. It replaces at the specification level the concept of ports, which are the entry and exit points of channels.

[13] The symbol $\downarrow$ is read “downarrow”.

$$_ \downarrow _ : Stream \times \mathbb{N} \longrightarrow Stream$$

$$\forall\, st, stO : Stream;\ i : \mathbb{N} \bullet$$
$$(st \downarrow i = stO \Leftrightarrow stO.c = st.c \wedge stO.m = st.m \downarrow i)$$

And at the level of sets of streams

$$_ \downarrow _ : \mathbb{P}\, Stream \times \mathbb{N} \longrightarrow \mathbb{P}\, Stream$$

$$\forall\, stSet : \mathbb{P}\, Stream;\ i : \mathbb{N} \bullet$$
$$stSet \downarrow i = \{stI : stSet \bullet stI \downarrow i\}$$

The function *stream* describes the association between a channel name and its corresponding stream. The function *streams* applies to a set of channel names and gives us a set of streams.

$$stream : C \longrightarrow Stream$$
$$streams : \mathbb{P}\, C \longrightarrow \mathbb{P}\, Stream$$

$$\forall\, c : C \bullet (\exists\, st : Stream \bullet stream(c) = st \wedge st.c = c)$$

$$\forall\, cSet : \mathbb{P}\, C \bullet streams(cSet) = \{c : cSet \bullet stream(c)\}$$

We use the function *restr* later, in the definition of a filter function. It is the selection of a subset of streams with specific names.

$$restr : (\mathbb{P}\, Stream \times \mathbb{P}\, C) \twoheadrightarrow \mathbb{P}\, Stream$$

$$\forall\, iSet : \mathbb{P}\, Stream;\ cSet : \mathbb{P}\, C \bullet$$
$$restr(iSet, cSet) = \{s : Stream \mid s \in iSet \wedge s.c \in cSet\}$$

5.2 Filters and Delays

A *primitive filter* is a computation unit that receives some information in its input streams and computes information each interval in its output streams. By definition, input and output streams do not form cycles. We define the function inside a filter in the schema *Behavior*, with the property that the output of two streams that are equal up to a interval i is the same up to such an interval. In this way, the computation of the behavior up to this interval

does not depend on future states, just previous states. Such a function is capable of computing the information in O, the output channels of interest.

$$
\begin{array}{l}
\textbf{Behavior} \\
\hline
fun : \mathbb{P}\ Stream \rightarrowtail \mathbb{P}\ Stream \\
\hline
\forall SI, SO : \mathbb{P}\ Stream \bullet fun(SI) = SO \Rightarrow SI \cap SO = \varnothing \\
\forall S1, S2 : \mathbb{P}\ Stream; i : \mathbb{N} \bullet \\
S1 \downarrow i = S2 \downarrow i \Rightarrow fun(S1) \downarrow i = fun(S2) \downarrow i
\end{array}
$$

$$
\begin{array}{l}
\textbf{PrimitiveFilter} \\
\hline
name : L \\
I : \mathbb{P}\ C \\
O : \mathbb{P}\ C \\
Behavior \\
\hline
restr(fun(streams(I)), O) = streams(O) \\
I \cap O = \varnothing
\end{array}
$$

A *delay* of one unit is a special type of filter, with one input and one output stream, in which the output at interval i+1 is equal to the input at interval i. We define it as:

$$
\begin{array}{l}
\textbf{Delay} \\
\hline
PrimitiveFilter \\
ID : C \\
OD : C \\
\hline
\{ID\} = I \wedge \{OD\} = O \\
fun(streams\{ID\}) = streams(\{OD\}) \wedge \\
tail((stream(OD)).m) = (stream(ID)).m
\end{array}
$$

We can simulate memory inside a filter by having a delay between a pair of input - output streams, in which the information through the delay can be as complex as necessary:

$$
\begin{array}{l}
\textbf{FilterWithMem} \\
\hline
PrimitiveFilter \\
delay : Delay \\
\hline
delay.ID \in O \wedge delay.OD \in I
\end{array}
$$

We can also define parameters, as those input channels that just receive information at the beginning of the execution.

$$\begin{array}{|l}
\textit{FilterWithParams} \\
\hline
\textit{PrimitiveFilter} \\
\textit{params} : \mathbb{P}\, C \\
\hline
\textit{params} \subseteq I \\
\forall\, p : \textit{params} \bullet (\forall\, i : \mathbb{N} \bullet i > 1 \Rightarrow ((\textit{stream}(p)).m)(i) = \varnothing) \\
\hline
\end{array}$$

We define a *filter* as a primitive filter that might have memory and parameters,

$$\textit{Filter} \mathrel{\widehat=} \textit{PrimitiveFilter} \land \textit{FilterWithMem} \land \textit{FilterWithParams}$$

5.3 An Example: A One-Bit Adder

As an example of how a PrimitiveFilter represents an operation, lets describe an adder of the information from two streams. A one-bit adder has three input channels, two for actual values and one for a carry value, and sends its output through two channels, one for the actual addition and one for the possible carry value. All messages are assumed to be either 1s or 0s. Since an input channel can receive several messages at once, we define the functionality in terms of an appearance of at least a 1 value.

$$\begin{array}{|l}
\hline
\textit{asBin} : M \longrightarrow \{0,1\} \\
\textit{asBinBag} : \text{bag}\, M \longrightarrow \{0,1\} \\
\hline
\forall\, b : \text{bag}\, M \bullet (\textit{asBinBag}(b) = 1 \Leftrightarrow (\exists\, m : M \bullet m \sqsubseteq b \land \\
\textit{asBin}(m) = 1)) \\
\hline
\end{array}$$

We use the following definition of a xor function between two bits,

$$\begin{array}{|l}
\hline
\textit{xor} : (\{0,1\} \times \{0,1\}) \longrightarrow \{0,1\} \\
\hline
\forall\, a, b : \{0,1\} \bullet (\textit{xor}(a,b) = 0 \Leftrightarrow (a = 0 \land b = 0) \lor (a = 1 \land b = 1)) \\
\hline
\end{array}$$

A bit adder is then defined as follows:

$BitAdder$

$PrimitiveFilter$
$i1 : C$
$i2 : C$
$cI : C$
$o : C$
$cO : C$

$I = \{i1, i2, cI\} \wedge O = \{o, cO\}$

$\forall\, i : \mathbb{N} \bullet$
$\#(((stream(o)).m)(i)) = 1 \wedge \#(((stream(cO)).m)(i)) = 1$

$\forall\, i : \mathbb{N} \bullet$
$asBinBag(((stream(o)).m)(i)) =$
$xor(xor(asBinBag(((stream(i1)).m)(i)), asBinBag(((stream(i2)).m)(i))),$
$asBinBag(((stream(cI)).m)(i)))$

$\forall\, i : \mathbb{N} \bullet asBinBag(((stream(cO)).m)(i)) = 1 \Leftrightarrow$
$[\![asBinBag(((stream(i1)).m)(i)), asBinBag(((stream(i2)).m)(i)),$
$asBinBag(((stream(cI)).m)(i))]\!] \sharp 1 > 1$

Such an adder can be used as a parallel or a sequential adder. Several bit adders can be connected in order to add several bits at once, or just one adder can sequentially receive several bits to be added, providing some delayed feedback of the carry information.

5.4 Applications

We compose groups of filters to form applications. We define first two auxiliary functions: *successors*, and *paths*. The *successors* of a filter f given a set of filters $iSet$ are all those filters in $iSet$ that have an input channel connected to an output channel of f,

$succ1 : (\mathbb{P}\, Filter \times Filter \times C) \longrightarrow \mathbb{P}\, Filter$
$successors : (\mathbb{P}\, Filter \times Filter) \longrightarrow \mathbb{P}\, Filter$

$\forall\, iSet : \mathbb{P}\, Filter; f : Filter \bullet (\forall\, c : f.O \bullet$
$succ1(iSet, f, c) = \{f2 : iSet \mid c \in f2.I\})$

$\forall\, iSet : \mathbb{P}\, Filter; f : Filter \bullet$
$successors(iSet, f) = \{f2 : iSet \mid (\exists\, c : C \bullet c \in f.O \wedge$
$c \in f2.I)\}$

The *paths* from a filter f in a set of filters is the set of all sequences of consecutive successors, starting at f. This set might be infinite, if there are cycles.

$$paths : (\mathbb{P}\ Filter \times Filter) \longrightarrow \mathbb{P}(\mathrm{seq}\ Filter)$$

$$\forall\ iSet : \mathbb{P}\ Filter;\ f : Filter \bullet$$
$$paths(iSet, f) = \{s : \mathrm{seq}\ Filter \mid s(0) = f \wedge (\forall\ i : \mathbb{N} \bullet$$
$$s(i+1) \in successors(iSet, s(i)))\}$$

Given a set of filters F, we say that there is a cycle from one of its filters if it exists a path that name such a filter more than once. We define the function *cyclic* over a set F that gives us a subset of filters with this property.

$$cyclic : \mathbb{P}\ Filter \longrightarrow \mathbb{P}\ Filter$$

$$\forall\ iSet : \mathbb{P}\ Filter \bullet$$
$$cyclic(iSet) = \{f2 : iSet \mid (\exists\ p : paths(iSet, f2) \bullet$$
$$(items\ p \sharp f2) > 1)\}$$

The concept of cycles is used now for the definition of a composition of filters. A set of interconnected filters create a *composed filter* [14]. A composed filter is uniquely identified by a name, its input and output channels are disjoint, its filters do not share output channels, and filters do not have loops unless they are mediated by delays. Delays can connect two distinct filters or delays, in any combination, so strings of delays are allowed. Objects represent references to content in the application [15]. We avoid two filters from reading different object states in a particular execution frame by forcing objects to be followed by delays. The behavior of a composed filter is defined as the composition of all behaviors in its filters and delays [16].

[14] A ComposedFilter is also a Filter, when all its details are hidden,so several layers of composition are possible.

[15] Such objects might be interrelated, but any conflicts between operations in different objects are treated with application logic, which may be generic.

[16] The last condition in this schema is redundant, but we want to emphasize the relationship between the composed filter's function and the functions of each filter inside.

ComposedFilter

$name : L$
$filters : \mathbb{P}_1\ Filter$
$delays : \mathbb{P}\ Delay$
$objects : \mathbb{P}\ Filter$
$IU : \mathbb{P}\ C$
$OU : \mathbb{P}\ C$
$I : \mathbb{P}\ C$
$O : \mathbb{P}\ C$
$Behavior$

$objects \subseteq filters$

$IU = \bigcup\{f : filters \bullet f.I\} \cup \bigcup\{d : delays \bullet d.I\}$

$OU = \bigcup\{f : filters \bullet f.O\} \cup \bigcup\{d : delays \bullet d.O\}$

$I = IU \setminus OU$

$O = OU$

$\forall f1, f2 : filters \bullet f1 \neq f2 \Rightarrow (f1.name \neq f2.name \wedge$
$f1.name \neq name)$

$I \cap O = \varnothing$

$\forall f1, f2 : filters \bullet f1 \neq f2 \Rightarrow f1.O \cap f2.O = \varnothing$

$cyclic(filters) = \varnothing$

$\forall\ o : objects \bullet (\forall\ c : o.O \bullet c \notin \bigcup\{f : filters \bullet f.I\})$

$restr(fun(streams(I)), O) = streams(O)$

$\forall\ c : O \bullet ((\exists f : filters; l : \mathbb{P}\ IU \bullet$
$stream(c) \in f.fun(streams(l))) \vee$
$(\exists\ d : delays; l : \mathbb{P}\ IU \bullet stream(c) \in d.fun(streams(l))))$

Our first attempt to define an *application* is as a composed filter with a special subset of filters, called *devices*. Devices are sources or sinks of information. In general, input channels to sources of information are parameters, and the input of a sink of information is the entire set of objects in the application.

$$\begin{array}{l}
\textbf{Application0} \\
\hline
ComposedFilter \\
devices : \mathbb{P}_1\ Filter \\
\hline
devices \subseteq filters \land devices \cap objects = \varnothing
\end{array}$$

An application can be seen in some cases as a complex filter, by hiding the extra definition. We do that with the function $app2Comp$

$$\begin{array}{l}
app2Comp : Application0 \longrightarrow ComposedFilter \\
\hline
\forall\, a : Application0;\ c : ComposedFilter \bullet app2Comp(a) = c \Leftrightarrow \\
a.name = c.name \land a.filters = c.filters \land a.delays = c.delays \\
\land\ a.objects = c.objects \land a.fun = c.fun
\end{array}$$

5.5 Dataflow Execution

An *ExecutionStep* of a filter is the process of obtaining certain information in the output ports, given the information in the input ports at a particular interval. The operation just shows the information in the output channels at a given interval.

$$\begin{array}{l}
\textbf{ExecutionStep} \\
\hline
\Xi Filter \\
input? : \mathbb{P}(\mathrm{bag}\ M) \\
output! : \mathbb{P}(\mathrm{bag}\ M) \\
i? : \mathbb{N} \\
\hline
\forall\, in : input? \bullet (\exists\, c : I \bullet in = ((stream(c)).m)(i?)) \\
\forall\, out : output! \bullet (\exists\, c : O' \bullet out = ((stream(c)).m)(i?))
\end{array}$$

The initialization state of a filter assigns the input and output channels of such a filter, and executes changes related to its parameters.

$$
\begin{array}{|l}
\hline
\textit{InitFilter} \\
\textit{Filter}' \\
ic? : \mathbb{P}\, C \\
oc? : \mathbb{P}\, C \\
input? : \mathbb{P}(\mathrm{bag}\, M) \\
output! : \mathbb{P}(\mathrm{bag}\, M) \\
\hline
I' = ic? \land O' = oc? \\
\forall\, in : input? \bullet (\exists\, c : I' \bullet in = ((stream(c)).m)(0)) \\
\forall\, out : output! \bullet (\exists\, c : O' \bullet out = ((stream(c)).m)(0)) \\
\hline
\end{array}
$$

An execution step for an application is given by the execution step of all filters that have some information in their input ports in a given frame, plus delays that had information in the previous interval. We define the functions *executingFilters* and *executingDelays* to describe which filters and delays are activated at any given interval.

$$
\begin{array}{|l}
\hline
streamsWithInput : (\mathbb{P}\, Stream \times \mathbb{N}) \longrightarrow \mathbb{P}\, Stream \\
executingFilters : (ComposedFilter \times \mathbb{N}) \longrightarrow \mathbb{P}\, Filter \\
executingDelays : (ComposedFilter \times \mathbb{N}) \longrightarrow \mathbb{P}\, Delay \\
\hline
\forall\, iSet : \mathbb{P}\, Stream; i : \mathbb{N} \bullet streamsWithInput(iSet, i) = \{s : iSet \mid (s.m)(i) \neq \varnothing\} \\
\forall\, f : ComposedFilter; i : \mathbb{N} \bullet \\
executingFilters(f, i) = \{f2 : f.filters \mid streamsWithInput(streams(f2.I), i) \neq \varnothing\} \\
\forall\, f : ComposedFilter; i : \mathbb{N} \bullet \\
executingDelays(f, i) = \{d2 : f.delays \mid streamsWithInput(streams(d2.I), i) \neq \varnothing\} \\
\hline
\end{array}
$$

Is it possible that not all filters and delays are executed in a particular interval, but over time, all filters and delays should be executed. The minimum number of intervals required for the execution of a composed filter is called its minimum execution time.

In order to properly execute an application we define extra requirements. No filter in an application generates output from information in parameters at interval 0, and the only filters with information at interval 1 are devices or connected to devices. These two conditions avoid problems with filters in execution separated by delays.

$$\begin{array}{l}
\textit{Application} \\
\textit{Application}0 \\
\hline
\forall f : \textit{filters} \bullet (\forall c : f.O \bullet ((\textit{stream}(c)).m)(0) = \emptyset) \\
\forall d : \textit{delays} \bullet (\forall c : d.O \bullet ((\textit{stream}(c)).m)(0) = \emptyset) \\
\forall f : \textit{executingFilters}(\textit{app2Comp}(\theta \textit{Application}0), 1) \bullet \\
f \in \textit{devices} \lor (\exists d : \textit{devices} \bullet (\exists p : \textit{paths}(\textit{filters}, d) \bullet \\
p \upharpoonright \{f\} \neq \emptyset))
\end{array}$$

5.6 Changes in the Dataflow

We will consider three types of changes in the dataflow: One that changes channels in a filter, without changing the structure of its output connections, another one that changes a filter in a connection scheme by replacing it for a compatible one and connecting it to the previous scheme, and a last one when filters are removed or added to the dataflow.

5.6.1 Channel Changes in a Filter

We assume that a filter function can compute the new output channels from the new input ones, since we have assumed that in general we see just a subset of possible results [17]. The operation is defined in a way that each old channel keeps the same, or has only one replacement. We also define the function *isCompatible*, which says if two channels can be replaced one by the other.

$$\begin{array}{l}
\textit{isCompatible} : (C \times C) \longrightarrow \{0, 1\} \\
\hline
\forall c1, c2, c3 : C \bullet (\textit{isCompatible}(c1, c1) = 1 \land \\
\textit{isCompatible}(c1, c2) = \textit{isCompatible}(c2, c1) \land \\
(\textit{isCompatible}(c1, c2) = 1 \land \textit{isCompatible}(c2, c3) = 1) \Rightarrow \\
\textit{isCompatible}(c1, c3) = 1)
\end{array}$$

For a filter, a channel change is straightforward: the new channels replace the old ones, providing that they are only pairs between these sets that are compatible.

[17] See the textitBehavior schema.

$$
\begin{array}{|l}
\textit{ChangeChannels} \\
\hline
f : \textit{Filter} \\
f' : \textit{Filter} \\
\textit{newIC}? : \mathbb{P}\ C \\
\textit{newOC}? : \mathbb{P}\ C \\
\textit{oldIC}? : \mathbb{P}\ C \\
\textit{oldOC}? : \mathbb{P}\ C \\
\hline
f'.\textit{name} = f.\textit{name} \\
\textit{oldIC}? \subseteq f.I \wedge \textit{oldOC}? \subseteq f.O \\
f'.I = \textit{newIC}? \wedge \#\textit{newIC}? = \#\textit{oldIC}? \wedge \\
(\forall\, c' : \textit{newIC}? \bullet (\exists_1\, c : \textit{oldIC}? \bullet c' = c \vee \textit{isCompatible}(c, c') = 1)) \\
f'.O = \textit{newOC}? \wedge \#\textit{newOC}? = \#\textit{oldOC}? \wedge \\
(\forall\, c' : \textit{newOC}? \bullet (\exists_1\, c : \textit{oldOC}? \bullet c' = c \vee \textit{isCompatible}(c, c') = 1)) \\
\hline
\end{array}
$$

ChangeChannelsCF defines a change of channels in a composed filter. All references to old channels are replaced by the new ones, and the structure of the composed channel is kept, since there is only one possible replacement for each channel.

$$
\begin{array}{l}
\textit{ChangeChannelsCF} \\
\Delta \textit{ComposedFilter} \\
newIC? : \mathbb{P}\, C \\
newOC? : \mathbb{P}\, C \\
oldIC? : \mathbb{P}\, C \\
oldOC? : \mathbb{P}\, C \\
\hline
name' = name \land \\
\#filters = \#filters' \land \#delays = \#delays' \land \\
\#objects = \#objects' \land \#delays = \#delays' \\[1ex]
\forall f : filters \bullet (\exists_1 f' : filters' \bullet \\
((oldIC? \cap f.I = \varnothing \land oldOC? \cap f.O = \varnothing \land f = f') \\
\lor ((oldIC? \cap f.I \neq \varnothing \lor oldOC? \cap f.O \neq \varnothing) \\
\land (\exists_1 ic, oc, ic', oc' : \mathbb{P}\, C \bullet ic \subseteq oldIC? \land oc \subseteq oldOC? \land \\
ic' \subseteq newIC? \land oc' \subseteq newOC? \land \\
ChangeChannels[ic/oldIC?, oc/oldOC?, ic'/newIC?, oc'/newOC?])))) \\[1ex]
\forall f : filters \bullet (\exists_1 f' : filters' \bullet \\
(\exists_1 c : f.O;\ c' : f'.O \bullet \\
(f.name = f'.name \land \{f2 : succ1(filters, f, c) \bullet f2.name\} = \\
\{f2' : succ1(filters', f', c') \bullet f2'.name\})))
\end{array}
$$

5.6.2 Change a Filter in a Connection Scheme

Let's now define a way to replace a filter by another in the dataflow, while connections are kept as much as possible. A filter holder allows us to replace a filter f by another, while keeping as much as possible previous connections of f. Part of the definition of a filter holder are a merge filter, a duplicator filter, and a compability function between channels similar to *isCompatible*, defined as follows:

$$
\begin{array}{l}
\textit{Merge2} \\
\textit{Filter} \\
i1 : C \\
i2 : C \\
o : C \\
\hline
I = \{i1, i2\} \land O = \{o\} \land i1 \neq i2 \land i1 \neq o \\[1ex]
\forall i : \mathbb{N} \bullet ((stream(o)).m)(i) = ((stream(i1)).m)(i) \uplus ((stream(i2)).m)(i)
\end{array}
$$

$$
\begin{array}{l}
\textit{Duplicate} \\
\hline
\textit{Filter} \\
i : C \\
o1 : C \\
o2 : C \\
\hline
I = \{i\} \wedge O = \{o1, o2\} \\
\forall\, n : \mathbb{N} \bullet ((\textit{stream}(i)).m)(n) = ((\textit{stream}(o1)).m)(n) \wedge \\
((\textit{stream}(i)).m)(n) = ((\textit{stream}(o2)).m)(n)
\end{array}
$$

$$
\begin{array}{l}
\textit{fhCompatible} : (C \times C) \longrightarrow \{0, 1\} \\
\hline
\forall\, c1, c2, c3 : C \bullet (\textit{fhCompatible}(c1, c1) = 1 \wedge \\
\textit{fhCompatible}(c1, c2) = \textit{fhCompatible}(c2, c1) \wedge \\
(\textit{fhCompatible}(c1, c2) = 1 \wedge \textit{fhCompatible}(c2, c3) = 1) \Rightarrow \\
\textit{fhCompatible}(c1, c3) = 1) \\
\forall\, c1, c2 : C \bullet (\textit{fhCompatible}(c1, c2) = 1 \Rightarrow \textit{isCompatible}(c1, c2) = 1)
\end{array}
$$

Mergers and duplicators can be seen as filters with the following functions:

$$
\begin{array}{l}
\textit{m2f} : \textit{Merge2} \longrightarrow \textit{Filter} \\
\textit{d2f} : \textit{Duplicate} \longrightarrow \textit{Filter} \\
\hline
\forall\, m : \textit{Merge2}; f : \textit{Filter} \bullet (\textit{m2f}(m) = f \Rightarrow \\
m.\textit{name} = f.\textit{name} \wedge m.I = f.I \wedge m.O = f.O \wedge m.\textit{fun} = f.\textit{fun} \wedge \\
m.\textit{delay} = f.\textit{delay} \wedge m.\textit{params} = f.\textit{params}) \\
\forall\, d : \textit{Duplicate}; f : \textit{Filter} \bullet (\textit{d2f}(d) = f \Rightarrow \\
d.\textit{name} = f.\textit{name} \wedge d.I = f.I \wedge d.O = f.O \wedge d.\textit{fun} = f.\textit{fun} \wedge \\
d.\textit{delay} = f.\textit{delay} \wedge d.\textit{params} = f.\textit{params})
\end{array}
$$

A *filter holder* surrounds its contained filter by a structure of filters shown in Figure 14. We assume the imposed structure is given by the channels with numbers and the contained filter f originally had the channels with letters.

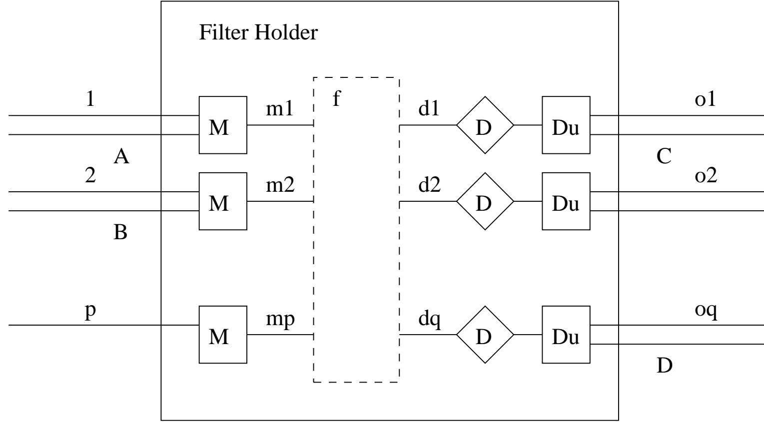

Figure 14: Internal Structure of an Filter Holder

The schema for a filter holder is as follows:

FilterHolder

$nameFH : L$
$IFH : \mathbb{P}\ C$
$OFH : \mathbb{P}\ C$
$predecessors : \mathbb{P}\ Filter$
$successors : \mathbb{P}\ Filter$
$fSet : \mathbb{P}\ Filter$
$mrgs : \mathbb{P}\ Merge2$
$dels : \mathbb{P}\ Delay$
$dups : \mathbb{P}\ Duplicate$

$\#fSet < 2$

$\forall\ p : predecessors \bullet p.O \cap IFH \neq \varnothing$

$\forall\ s : successors \bullet s.I \cap OFH \neq \varnothing$

$\forall\ ifh : IFH \bullet \exists_1\ m : mrgs \bullet ifh = m.i1$

$\forall\ ofh : OFH \bullet \exists_1\ del : dels; dup : dups \bullet$
$(del.OD = dup.i \wedge dup.o1 = ofh)$

$fSet \neq \varnothing \wedge (\exists\ f : fSet \bullet$
$((\forall\ ic : f.I \bullet (\forall\ ifh : IFH \bullet fhCompatible(ic, ifh) = 0) \vee$
$(\exists_1\ ifh : IFH \bullet fhCompatible(ic, ifh) = 1))))$

Once a new filter $f2$ is assigned to an object holder, it disconnects the

previous f contained, restores its original connections, and connects $f2$ inside the structure. Such a structure keeps the original connections of any filter, and avoids cycle problems, due the delays in the outputs.

$$
\begin{array}{l}
\textbf{Schema } NewFilterFH \\
\Delta FilterHolder \\
f? : Filter \\
f! : Filter \\
\hline
nameFH = nameFH' \land IFH = IFH' \land OFH = OFH' \land \\
predecessors = predecessors' \land successors = successors' \\
\\
fSet \neq \varnothing \land (\exists_1 fOld : fSet \bullet \\
fOld.name = f!.name \land \\
f!.I = fOld.I \setminus \{mrg : mrgs \mid mrg.o \in fOld.I \bullet mrg.o\} \cup \\
\{mrg : mrgs \mid mrg.o \in fOld.I \bullet mrg.i2\} \land \\
f!.O = fOld.O \setminus \{del : dels \mid del.ID \in fOld.O \bullet del.ID\} \cup \\
\{dup : dups \mid (\exists oc : fOld.O \bullet fhCompatible(dup.o2, oc) = 1) \bullet dup.o2\}) \\
\\
\exists f?' : fSet'; ic, ic', oc, oc' : \mathbb{P}\, C \bullet \\
(ic = \{c : f?.I \mid (\exists_1 ifh : IFH \bullet fhCompatible(c, ifh) = 1)\} \land \\
oc = \{c : f?.O \mid (\exists_1 ofh : OFH \bullet fhCompatible(c, ofh) = 1)\} \land \\
ic' = \{m : mrgs \mid (\exists_1 ifh : IFH; c : f?.I \bullet \\
(fhCompatible(c, ifh) = 1) \land ifh = m.i1) \bullet m.o\} \land \\
oc' = \{del : dels \mid (\exists_1 ofh : OFH; dup : dups; c : f?.O \bullet \\
(fhCompatible(c, ofh) = 1) \land ofh = dup.o1 \land dup.i = del.OD) \bullet del.ID\} \land \\
ChangeChannels[f?/f, f?'/f', ic/oldIC?, oc/oldOC?, ic'/newIC?, oc'/newOC?])
\end{array}
$$

5.6.3 Add and Remove Filters

The addition and removal of filters — or delays — in a composed filter is straightforward: The new filter is added or removed from the set of filters [18].

[18] It is important to notice that these two operations have to fulfill all previous requirements for composed filters; in particular, cycles without delays are not allowed.

$$
\begin{array}{l}
\textbf{AddFilter} \\
\hline
\Delta Application \\
newF? : Filter \\
\hline
name = name' \land delays = delays' \land objects = objects' \land \\
devices = devices' \\
newF?.name \notin \{f : filters \bullet f.name\} \cup \{name\} \land \\
newF?.O \cap \bigcup\{f : filters \bullet f.O\} = \varnothing \land \\
(\forall o : objects \bullet o.O \cap newF?.I = \varnothing) \land \\
filters' = filters \cup \{newF?\} \land cyclic(filters') = \varnothing
\end{array}
$$

$$
\begin{array}{l}
\textbf{RemoveFilter} \\
\hline
\Delta Application \\
oldF? : Filter \\
\hline
name = name' \land delays = delays' \land objects = objects' \land \\
devices = devices' \\
filters' = filters \setminus \{oldF?\}
\end{array}
$$

In the same way, modifications of delays can be defined as follows:

$$
\begin{array}{l}
\textbf{AddDelay} \\
\hline
\Delta Application \\
newD? : Delay \\
\hline
name = name' \land filters = filters' \land objects = objects' \land \\
devices = devices' \\
newD?.name \notin \{d : delays \bullet d.name\} \cup \{name\} \land \\
newD?.O \cap \bigcup\{d : delays \bullet d.O\} = \varnothing \\
delays' = delays \cup \{newD?\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{RemoveDelay} \\
\hline
\Delta Application \\
oldD? : Delay \\
\hline
name = name' \land filters = filters' \land objects = objects' \land \\
devices = devices' \\
delays' = delays \setminus \{oldD?\}
\end{array}
$$

5.7 Operations Over an Application

The operations over an application are based on the previous schemas. An *InTml application* is a well defined application with object holders:

$$
\begin{array}{l}
\textit{InTmlApp} \\
\hline
\textit{Application} \\
\textit{fhs} : \mathbb{P}\ \textit{FilterHolder} \\
\hline
\forall\, \textit{fh} : \textit{fhs} \bullet \textit{fh.IFH} \subseteq \textit{IU} \wedge \textit{fh.OFH} \subseteq \textit{OU} \\
\wedge\ \textit{fh.predecessors} \subset \textit{filters} \\
\wedge\ \textit{fh.successors} \subset \textit{filters} \\
\wedge\ \textit{fh.fSet} \subset \textit{filters} \wedge \textit{fh.dels} \subseteq \textit{delays} \\
\wedge\ \{\textit{mrg} : \textit{fh.mrgs} \bullet \textit{m2f}(\textit{mrg})\} \subset \textit{filters} \\
\wedge\ \{\textit{dup} : \textit{fh.dups} \bullet \textit{d2f}(\textit{dup})\} \subset \textit{filters}
\end{array}
$$

The execution of an InTml application is either the normal flow of messages, or one of the following special changes: channel changes, object holder executions, addition of filters, removal of filters, addition of delays, and removal of delays. The normal dataflow execution is similar to the execution of any filter:

$$
\begin{array}{l}
\textit{InTmlExecuteDataflow} \\
\hline
\Xi \textit{InTmlApp} \\
\textit{inputD}? : \mathbb{P}(\mathrm{bag}\ M) \\
\textit{output}! : \mathbb{P}(\mathrm{bag}\ M) \\
i? : \mathbb{N} \\
\hline
\forall\, \textit{in} : \textit{inputD}? \bullet (\exists\, c : I \bullet \textit{in} = ((\textit{stream}(c)).m)(i?)) \\
\forall\, \textit{out} : \textit{output}! \bullet (\exists\, c : O' \bullet \textit{out} = ((\textit{stream}(c)).m)(i?))
\end{array}
$$

A change of channels is defined on top of the changes in a composed filter, with extra conditions for filter holders that assures that if the change involves a filter connected to a filter holder, the change will keep the structure of the dataflow as it was before:

$$
\begin{array}{l}
\textit{InTmlExecuteChangeChannels} \\
\hline
\Delta \textit{InTmlApp} \\
\textit{newIC}? : \mathbb{P}\, C \\
\textit{newOC}? : \mathbb{P}\, C \\
\textit{oldIC}? : \mathbb{P}\, C \\
\textit{oldOC}? : \mathbb{P}\, C \\
\hline
\textit{ChangeChannelsCF} \\
\\
\forall \textit{fh} : \textit{fhs} \bullet (\exists_1 \textit{fh}' : \textit{fhs}' \bullet \\
(\textit{fh}.\textit{IFH} \cap \textit{newIC}? = \varnothing \wedge \textit{fh}.\textit{OFH} \cap \textit{newOC}? = \varnothing \wedge \\
\textit{fh} = \textit{fh}') \vee \\
((\textit{fh}.\textit{IFH} \cap \textit{newIC}? \neq \varnothing \vee \textit{fh}.\textit{OFH} \cap \textit{newOC}? \neq \varnothing) \\
\wedge (\textit{fh}.\textit{nameFH} = \textit{fh}'.\textit{nameFH} \wedge \textit{fh}.\textit{predecessors} = \textit{fh}'.\textit{predecessors} \wedge \\
\textit{fh}.\textit{successors} = \textit{fh}'.\textit{successors} \wedge \textit{fh}.\textit{fSet} = \textit{fh}'.\textit{fSet} \wedge \\
\textit{fh}'.\textit{IFH} = \textit{fh}.\textit{IFH} \setminus \textit{oldIC}? \cup \{m : \textit{fh}'.\textit{mrgs} \bullet m.i1\} \wedge \\
\textit{fh}'.\textit{OFH} = \textit{fh}.\textit{OFH} \setminus \textit{oldOC}? \cup \{d : \textit{fh}'.\textit{dups} \bullet d.o1\})))
\end{array}
$$

The execution of filter holders in an InTml application is based on the previous schema *NewFilterFH*, with an additional condition to assure that nothing else changes:

$$
\begin{array}{l}
\textit{InTmlExecuteFHs} \\
\hline
\Delta \textit{InTmlApp} \\
f? : \textit{Filter} \\
f! : \textit{Filter} \\
\hline
f? \in \textit{filters} \wedge f! \in \textit{filters} \\
\\
\textit{name} = \textit{name}' \wedge \textit{delays} = \textit{delays}' \wedge \textit{objects} = \textit{objects}' \wedge \\
\textit{devices} = \textit{devices}' \\
\\
\exists_1 \textit{fh} : \textit{fhs}; \textit{fh}' : \textit{fhs}'; \textit{NewFilterFH} \bullet \\
(\textit{fh}.\textit{nameFH} = \textit{nameFH} \wedge \textit{fh}.\textit{IFH} = \textit{IFH} \wedge \textit{fh}.\textit{OFH} = \textit{OFH} \wedge \\
\textit{fh}.\textit{predecessors} = \textit{predecessors} \wedge \textit{fh}.\textit{successors} = \textit{successors} \wedge \\
\textit{fh}.\textit{fSet} = \textit{fSet} \wedge \textit{fh}.\textit{mrgs} = \textit{mrgs} \wedge \textit{fh}.\textit{dels} = \textit{dels} \wedge \\
\textit{fh}.\textit{dups} = \textit{dups} \wedge \textit{fh}'.\textit{nameFH} = \textit{nameFH}' \wedge \textit{fh}'.\textit{IFH} = \textit{IFH}' \wedge \\
\textit{fh}'.\textit{OFH} = \textit{OFH}' \wedge \textit{fh}'.\textit{predecessors} = \textit{predecessors}' \wedge \\
\textit{fh}'.\textit{successors} = \textit{successors}' \wedge \\
\textit{fh}'.\textit{fSet} = \textit{fSet}' \wedge \textit{fh}'.\textit{mrgs} = \textit{mrgs}' \wedge \textit{fh}'.\textit{dels} = \textit{dels}' \wedge \\
\textit{fh}'.\textit{dups} = \textit{dups}')
\end{array}
$$

Adding and removing filters are directly defined over the previous schemas *AddFilter* and *RemoveFilter*, respectively:

$$\begin{array}{l}
\textit{InTmlExecuteAddFilter} \\
\hline
\Delta \textit{InTmlApp} \\
\textit{newF}? : \textit{Filter} \\
\hline
\textit{fhs} = \textit{fhs}' \\
\textit{AddFilter}
\end{array}$$

$$\begin{array}{l}
\textit{InTmlExecuteRemoveFilter} \\
\hline
\Delta \textit{InTmlApp} \\
\textit{oldF}? : \textit{Filter} \\
\hline
\textit{fhs} = \textit{fhs}' \\
\textit{RemoveFilter}
\end{array}$$

Similarly changes in delays are defined as follows:

$$\begin{array}{l}
\textit{InTmlExecuteAddDelay} \\
\hline
\Delta \textit{InTmlApp} \\
\textit{newD}? : \textit{Delay} \\
\hline
\textit{fhs} = \textit{fhs}' \\
\textit{AddDelay}
\end{array}$$

$$\begin{array}{l}
\textit{InTmlExecuteRemoveDelay} \\
\hline
\Delta \textit{InTmlApp} \\
\textit{oldD}? : \textit{Delay} \\
\hline
\textit{fhs} = \textit{fhs}' \\
\textit{RemoveDelay}
\end{array}$$

Finally, an *InTmlStep* is the execution of any of the tasks that change the state of the dataflow, followed by the execution of the dataflow.

$$\begin{array}{l}
\textit{InTmlStep} \mathrel{\widehat{=}} (\textit{InTmlExecuteChangeChannels} \lor \textit{InTmlExecuteFHs} \lor \\
\textit{InTmlExecuteAddFilter} \lor \textit{InTmlExecuteRemoveFilter} \lor \\
\textit{InTmlExecuteAddDelay} \lor \textit{InTmlExecuteRemoveDelay}) \land \\
\textit{InTmlExecuteDataflow}
\end{array}$$

6 Properties of this Architecture

This architecture has the following features:

- Filters can run in different processors. There are no restrictions on the simultaneous execution of filters, and how they send information to followers. This allows parallel implementations and also sequential ones, with the same semantics.
- A filter "knows" all events that are originated in the same time frame and received through its input channels. This allows specific implementations to filter unnecessary information.
- The state of the world, reflected in terms of the state of the objects in the application, is consistent for all filters during the execution of a time frame. This avoids side effects, as in other dataflow–based implementations such as VRML, related to the order of execution of filters.
- Composed filters have clear recursive semantics and allow complexity management, by hiding unnecessary details.
- An application can have as many devices as required, all treated in a uniform way.
- The particular implementation of the behavior of a filter is hidden from the dataflow point of view. In this way, we can separate the high level design of the dataflow and the low level design of behaviors and interaction techniques.
- Object holders define a mechanism similar to pointers in common programming languages, and they are very useful for the definition of dynamic changes.
- The semantics described here can be implemented in several platforms from a simple desktop computer to a massively parallel computer. In this way, a VR application is scalable to a wide variety of hardware platforms, while it keeps the same semantics.
- A designer can easily identify which filters in a dataflow are specific to a particular hardware platform. This allows designers to perform a process of reaccomodation of VR applications to different hardware platforms and interaction styles. We call such a process retargeting, and it will be part of the methodology we define later in this thesis.

There are also some issues that require further research, such as:

- Actual comparative results between a parallel and a sequential implementation of InTml have to be performed. Ideally, we should be able to create some analytical methods for measuring performance advantages in a parallel implementation of a specific application, but we have to explore this possibility in more in detail.
- InTml is a new component technology that takes into account the specific quality attributes required in VR applications. However, we require more applications implemented in this technology to validate more thoroughly its advantages and features. Such experimental tests will require a community of users and an active developer community to support them.
- Exact time management and synchronization are issues not directly visible in InTml. This allows designers to worry about requirements without taking care of synchronization issues. However, such issues should be addressed in a succesful VR application. Our current approach forces designers to hide synchronization issues under filters that transparently handle several streams into a synchronized one. This approach has to be tested in more detail to understand its implications.
- More development tools are necessary in order to provide a true professional environment for VR designers.
- As InTml libraries grow, tools for finding and organizing them will be required in order to make them usable and avoid work repetition.

6.1 An Example of InTml Interpretation

This specification allows us to understand the semantics behind an InTml application representation. For example, if we analyze again Figure 2 one can infer the following characteristics:

- All filters in the application can be executed in one step, since all filters are reachable from the device `handTracker` without delays in the middle [19]. The `handTracker` sends changes to both the object and the selection technique. The selection technique takes the new coordinates and the current object state and decides if the object will collide any

[19] See definition of *executingFilters* in Section 5.5

other object in the scene, once the new coordinates are set. If this is the case, the collided object is sent to the feedback technique. All these operations are executed at the same interval. Once the dataflow execution has finalized, changes in objects are executed (In this case, new position and orientation for `handRepr` from the tracker).

- Since *InTmlExecuteFHs* is executed before *InTmlExecuteDataflow*, the object `handRepr` is assigned to the object holder `handRepresentation` before events from the tracker arrive (i.e., `position` and `orientation`).

- Another representation of the same application is shown in Figure 15, with a slightly different meaning but with the same results. In this case, changes in position and orientation are propagated after they are executed in the object. This representation executes the entire dataflow in two steps: the first one with {`handTracker, handRepresentation`} and the second with {`SelectByTouching, Feedback`}. This approach might be preferred if the object can decide if it can do the requested operation. In this case, the processing is executed first, and if the object decides that the change can be done, it will propagate the changes to the selection technique. Also note that when the selection technique executes, `handRepr` actually has the same position and orientation as the ones received in the input ports `position` and `orientation`. We allow then `SelectByTouching` to operate in two slightly different modes, since it does not take into account the actual position and orientation of the object.

- The actual description of the function executed by a filter is hidden from the diagram, and in the current InTml implementation, it is defined in a textual description attached to each filter class. For example, the function `SelectByTouching` is described as follows: It is a selection technique that takes an object and checks if it will collide with another object in a scene after moving to a new position and orientation, received as parameters. Addition and removals of objects in the scene are allowed, but those changes have to be explicitly informed. It is possible to design this selection technique in a different way, and Figure 16 shows a different one, with the following interpretation: it is a selection technique that takes the current position of an object and checks if it collides with another object in the scene. Changes in the scene or in the object are implicitly taken into account. This definition might be preferred, since it allows less modes of operation and avoids possible misinterpretations of its execution. It might also be defined

in terms of the previous one. However, it takes two intervals to be totally executed, due the inner object holders.

- If the first representation of `SelectByTouching` is used, `handRepr` will receive two copies of the events from `handTracker`: one from the object holder inside the selection technique, one from `handRepresentation`. While this might be redundant it does not affect the semantics of the application in any way, and compiler techniques can be used to avoid this redundancy.

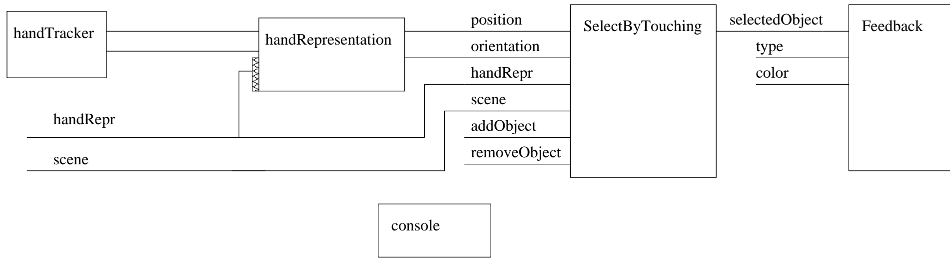

Figure 15: A Modified Version of a Simple Application.

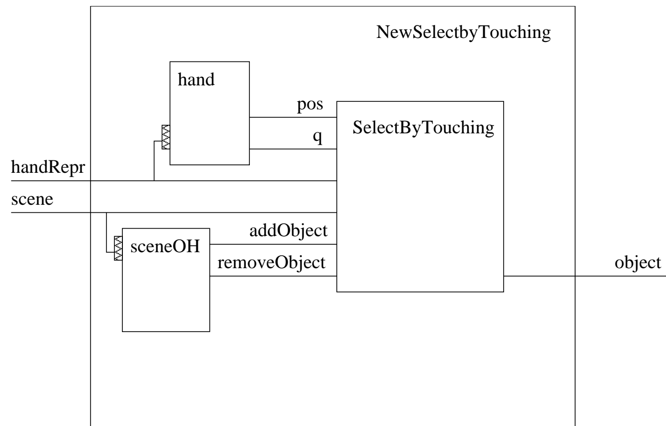

Figure 16: A Modified Version of SelectByTouching.

6.2 Lessons Learned from the Z Language Description

Our attempt to formally describe the semantics of InTml gave us a better understanding of what our model is, and what are its capabilities. We

started our first implementation after prototyping three environments for VR development, which gave us a good understanding of the requirements of such an endeavour. However, the description in the Z language that we started after the first implementation gave us a more generic, clean, and scalable description than the one that we implemented. We extended the concept of object holders to allow any type of filter, which makes InTml a second–order language (filters can have filters as arguments).

Formalization allowed us to better understand the meaning of an InTml application than from just a drawing, since the inner semantics of each element and the overall execution model is clearer. The formalism allowed us to cleanly separate the meaning of a filter from the concept of a filter holder, and what could be the type of a holder. It also allowed us to make clearer the differences and similarities between applications and composed filters.

The concept of a delay emerged from the work Lee and Parks [5] as a very useful addition to the set of concepts in InTml. Despite the fact that it does not yet appear in the XML syntax, a delay is a very useful abstract concept that allows us to understand the execution model, how cycles are executed, and how object holders work.

An important concept in the Z description is the one of dynamic changes in the structure of the dataflow, that will allow us in the future to offer a richer language, i.e. with support for adding or deleting filters, changes in connections, delays at the XML level, and filter holders.

Finally, the description in the Z language also allowed us to compare the semantics of our dataflow proposal with other dataflow proposals, and reuse part of their notation and their semantics. The differences of our proposal with previous ones in the area of dataflow based computation are clearer, and new questions have emerged for future work.

References

[1] Alias Wavefront. 3D Max. http://www.discreet.com/products/3dsmax/, 2003.

[2] Alias Wavefront. Maya. http://www.aliaswavefront.com/en/products/maya/index.shtml, 2003.

[3] Jon Barrileaux. *3D User Interfaces With Java 3D*. Manning Publications, August 2000.

[4] Blender.org. Blender. http://www.blender.org/, 2003.

[5] Edward A. Lee and Thomas M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, May 1995.

[6] Jiandong Liang and Mark Green. Geometric modeling using six degrees of freedom input devices. In *3rd International Conference on CAD and Computer Graphics*, pages 217–222, 1993.

[7] Sun Microsystems. Java 3D Home Page. http://java.sun.com/products/ java-media/3D/index.html, 1997.

[8] Jan Philipps and Bernhard Rumpe. Refinement of pipe-and-filter architectures. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing System. LNCS 1708, pages 96-115.* Springer, 1999.

[9] Ivan Poupyrev, Mark Billinghurst, Suzanne Weghorst, and Tadao Ichikawa. The go-go interaction technique: non-linear mapping for direct manipulation in vr. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 79–80. ACM Press, 1996.

[10] The matrix and quaternion faq. http://skal.planet-d.net/demo/matrixfaq.htm.

[11] SGI. Iris performer home page. http://www.sgi.com/software/performer, 2003.

[12] Mike Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition edition, 1992.

[13] D. Touraine, P. Bourdot, Y. Bellik, and L. Bolot. A framework to manage multimodal fusion of events for advanced interactions within virtual environments. In S. Müller W. St?rzlinger, editor, *Virtual Environments '02*, Eurographics, pages 159–168. Springer-Verlag Wien New York, 2002. Proc's Eurographics Workshop, Barcelona, Spain, 2002.