Supporting views in network management systems

Carlos Palma and Luis Rodrigues

Rural Informática, SA, Damaia (cpalma@creditoagricola.pt)
and
Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, Lisboa (ler@di.fc.ul.pt)

In network management systems, a view represents a subset of the management domain tailored to a specific activity or management role. The support for views is extremely relevant in large-scale management systems where it becomes infeasible to centralize all the management information. The paper proposes a model that recognizes the importance and provides support for the coexistence of different management views of a large set of entities. A prototype following the proposed model has been implemented as a set of extensions to the open source Scotty platform.

Keywords: Network management, view support, event propagation

1 Introduction

The vast majority of current management systems provide the tools needed to describe the set of network components as well as their attributes. However, in a system with a large number of components it becomes infeasible to manage a flat space of components. Therefore, it is extremely important to aggregate components in *views* that represent different management domains.

This paper proposes a model to support the representation of complex networks in management systems. The model, that distinguishes the representation of individual components from the representation of the relationships among these components, is based on a two level architecture:

- The *component level*: At this level each network component is described individually.
- The *view level*: The view level defines the relationships among components that are specific to a particular management activity. At this level one defines also which attributes of the component are visible in the context of the view.

In order to manage and monitor a system, one or more views need to be activated. The management system must ensure that the information provided to different managers is consistent, even if they interact with different views. This is ensured via the two-level architecture: changes in the managed objects are registered at the component level and then reflected at all relevant views.

The model also proposes a flexible event propagation mechanism. Event propagation is performed not only within a view but also across different views. Event propagation is used both to apply operations to groups of components and to notify the users of relevant occurrences. A prototype of a system implementing the proposed model has been developed as a Tcl extension to the Scotty platform [SL95].

The paper is organized as follows. In Section 2 we motivate the need for the view support in management systems and Section 3 briefly introduces related work. The proposed model is described in Section 4 and its implementation in Tcl introduced in Section 5. An evaluation of the model is given in Section 6 and Section 7 concludes the paper.

O. Festor and A. Pras (Eds.) : 12th International Worshop on Distributed Systems: Operations and Management
DSOM'2001 Nancy France, October 15-17, 2001.

2 Motivation

Any complex system includes a large number of network components such as computers, devices, applications, etc. Each of these components is characterized by a set of attributes that need to be accessed in the course of management activities. However, most of management activities do not need to access all attributes of each component nor the complete set of managed components. The excess of non-relevant information in a given interface may even be detrimental to the efficiency and security of the management activities. Therefore, it is extremely important to define different *views* of the managed system. Each view is created having a specific activity or role in mind, and should include only the objects (and attributes) relevant for that activity.

Consider for instance a system with machines (running a combination of Linux and some proprietary operation system) physically located in different rooms and connected to different LAN segments. For certain activities it is interesting to organize a *view* of the system according to the geographical deployment of the machines, with an indication of identifier and operating system. For other activities, it is more useful to present the view according to the LAN segments and showing only the machines that run a specific operating system. Therefore, the management system should allow to express these different relations among the management components.

The notion of *view* addresses this need. The view is an abstraction that allows to aggregate and make visible the representation of the managed components that belong to a particular context. Additionally, the view filters the component attributes and behavior, such that only the properties that are relevant in the context of the view are shown to the user.

Using a view, the user interface is simplified, since all the information that is non-relevant for a specific management activity may be hidden from the operator. Furthermore, the view can also be used a mechanism to control the access to sensible information, improving the security of the management system.

3 Related work

The need for structuring and grouping the managed objects has been present from the beginning in network management platforms. Therefore, most network management systems include some form of representation mechanism. One can distinguish two main classes of network management systems: systems whose elementary unit is the component attribute and systems whose elementary unit is the network component.

The first class of approaches considers that each individual attribute is a basic unit of management. Network elements are defined as groups of attributes. It is also possible to create components that mix attributes from several distinct physical devices and that have no direct mapping to concrete network components. We can cite two main examples of this approach: the SNMPQL [Yeo90] that represents the network and a set of relational tables that can be accessed using SQL and the approach described in [KSSZ97] where the network is represented as a spreadsheet. Using these approaches it is difficult to support effectively event propagation and handling mechanisms. A similar approach has been described in [Gol96], where extensions to the SNMP-SMI support the creation of MIB views in an agent. The creation of views is supported trough relational operations (joins, filtering, etc) on SNMP table using a SQL like language. However, these views are restricted to a single MIB.

Most of the existing commercial and open source network management system fit into the second category. Our model and implementation can also be classified in this class. Representative examples are the Tkined/Scotty [SL93], the *Domains* system [SM89, MS93], the *CA Unicenter* [Ass], and the *HP Openview* [Pac]. In is interesting to note that the last two commercial systems, despite their strong penetration in the market, do not satisfactorily address the issues addressed by our proposal.

The *CA Unicenter* distinguishes the object model from the topology model. The topology model is concerned with the relations among objects and, to a certain extent, supports the definition of different perspectives of the managed network. The *HP Openview* also makes a distinction between the representation of the managed objects and its presentation: the same object may be presented using different icons; icons can be grouped in maps that can be linked in a hierarchical manner. However, in both cases, there is no support to refine the presentation of a given object in a view sensitive manner: if an attribute is visible in a view it must be necessarily visible in all views. Both our proposal and the previous examples can be

considered as different application of the Model-View-Controller design pattern [BMHRS96] to network management tools.

4 Model

In this section we present a model for the implementation of the view abstraction. Our model intends to combine flexibility, in the sense that the same set of managed objects can be represented in many different views, with the consistency of information provided across views. For instance, if two operators are monitoring the same component using two distinct views and the component changes its status due to an action executed in the context on a view, this change of status should also be visible in the other views.

4.1 Basic elements

The model proposed here uses a composition of the following basic elements: *core-elements*, *links*, *elements*, *groups*, *maps* (a map represents a management view), and *repositories*.

- A *core-element* represents a concrete managed object. Typically, it represents a network component (physical or logical) such as a node or a process. Each managed object is represented by a single core-element, no matter in how many maps it is included. The core-element component encapsulates the complete set of attributes and behavior of the managed object.
- An *element* represents a core-element in a specific map. Therefore, the element may restrict the number of attributes that are visible and the behavior of the managed object in the context of that map. Note that a single managed object, represented by a core-element, can be associated with multiple elements.
- A *link* represents a physical or logical connection among two managed objects in a given context. Therefore, a link connects elements or groups of the same map.
- A *group* aggregates elements and other groups allowing to hierarchically structure the map. The group can be used to merely aggregate elements that are logically related or to create a composite component from other components.
- A *map*, that represents a management view, is a set of groups, elements and links that together represent a concrete management context. It is possible to associate *global constraints* that restrict the type of information from the core-elements that can be visible in the map. Note that constraints cannot associated with groups.
- A *repository* is a persistent representation of the relevant state of the previous elements. The repository contains the characterization of a set of core-elements and the description of all the maps (with associated elements, links and groups).

4.2 Attributes

Each element of our model is characterized by a set of *attributes* and *methods*, as described below.

- An *attribute* is a variable that represents a property of the managed object. Attributes are simply numbers or strings. The attributes can be used to store management information, parameters that define the way a component is tested and monitored, information required for network access, temporary data resulting from monitoring activities, etc. Some of the attributes can be used to uniquely identify the managed object, such as, for instance, a MAC address or a serial number.
- The *methods* define management activities (in our concrete implementation, a management script). These scripts can be executed periodically or only when explicitly invoked (when the attributes are accessed). The methods can be used to return values that are obtained by applying some function to a set of simpler attributes. The methods can also invoke some management protocol to interact with the concrete devices.

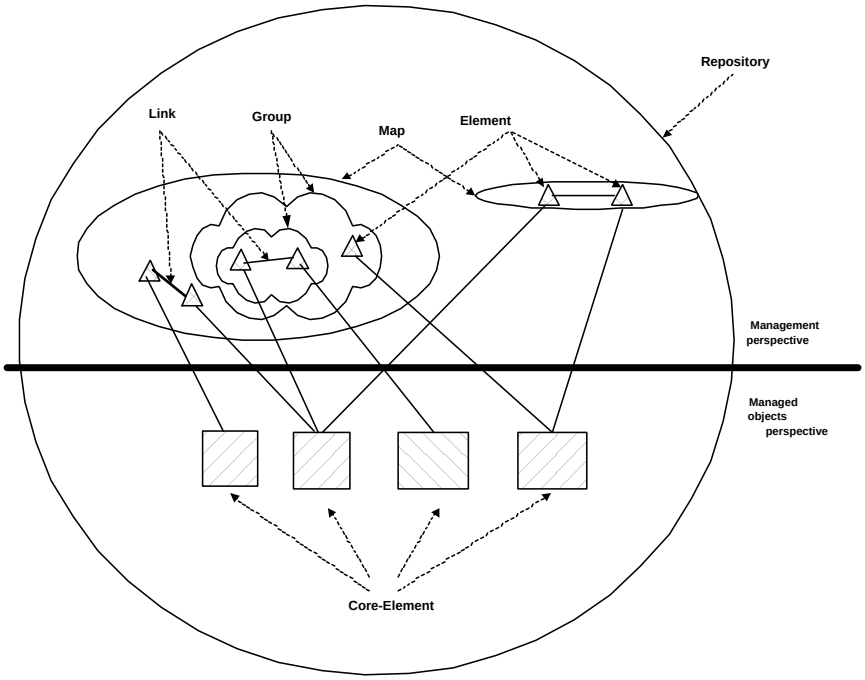

Fig. 1: Proposed object model

4.3 Relationships

The basic entities of our model may have the following relationships, as illustrated in Figure 1.

- A core-element can have one or more attributes and methods. A core-element also maintains a reference to all of its elements. Therefore, the core-element is able to propagate an event created in the context of a given map to all maps to which it belongs.
- An element maintains a reference to the core-element it represents, a reference to the group in which it is contained and references to all its associated links. All the attributes that describe the state of the managed object are not maintained at the element level; they are always maintained at the level of core-elements to ensure the consistency of information to be provided across different views.

  Note that not all core-element attributes are visible in every element, since each view defines constraints on the attributes that are visible through the elements of that view.

  Remember also that each core-element can be identified by the value of some relevant attribute of the managed object that it represents (for instance, a serial number). This unique identifier will be visible in all maps in which the object is represented.
- A link can have one or more attributes and methods, and references to the elements that it connects.
- A group can have one or more attributes and methods. A group has a reference to the upper group (or to the map, if it is the root group) and a set of references to all groups, links and elements that it contains.
- Finally, a map can also have one or more attributes and methods and a reference to a root group.

4.4 Constraints

By default, an element represents all attributes and methods of the associated core-element. However, our model foresees the possibility of defining restrictions to the set of attributes visible in a map by defining

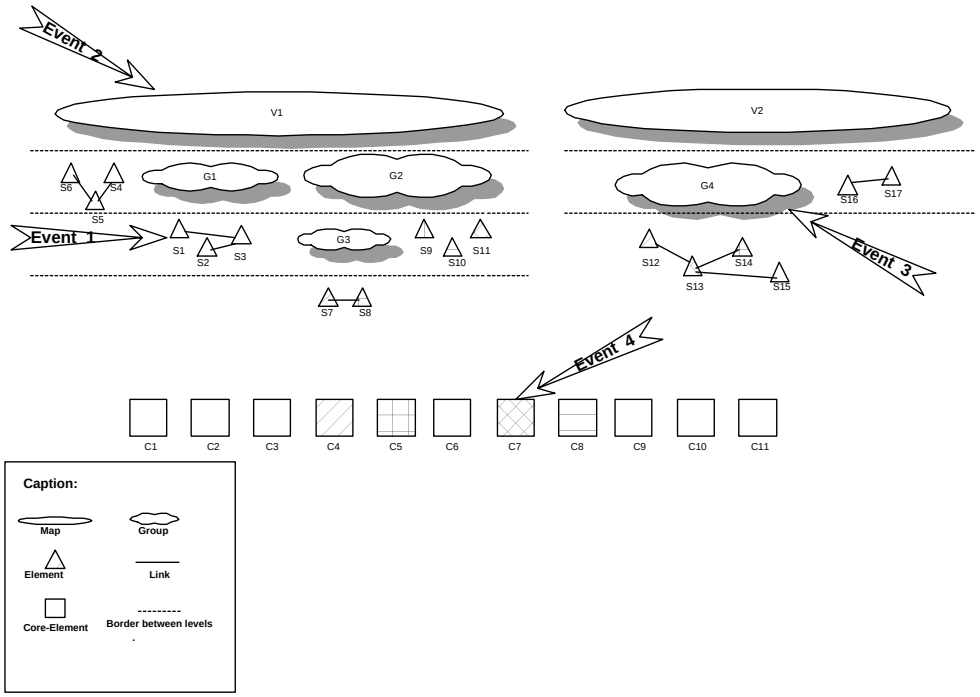

Fig. 2: Event propagation example

constraints at the map level. Constraints can also restrict the propagation of events. As we have noted previously, these constraints may simplify the management activities and improve the security of the management system.

4.5 Filters

Filters are intended as a mechanism to automate and simplify the creation of maps. A *filter* is a constraint on the characteristics of the core-elements that may belong to a given map. After specifying a filter, one can request the creation of an *implicit map*, a map that contains all managed objects with the characteristics defined in the filter of the map. The map designer can later can aggregate the resulting elements in groups and link them in different ways. However, she is not allowed to create elements additional in an implicit map.

Therefore, the model support two different alternatives to create views: implicit maps, whose content is specified by the filter, and explicit maps, whose elements are added explicitly, one by one, by the view designer. Explicit maps give more configuration possibilities to the map designer but also require a greater deal of effort to define.

4.6 Event model

In order to model the behavior of the components, our architecture defines an *event* processing and propagation model. Events are typed and can be propagated among the objects of active maps according to different policies. Event handlers can be defined at the level of individual components (core-elements, groups, elements, etc). Therefore, it is possible to have fine-grain control on how the events are processed at each level of the hierarchy.

There are two main classes of events: *internal* events and *external* events. Internal events are raised automatically when there is a change in any component of a repository, such as changes in attributes or addition/removal of elements. Internal events are not propagated and must be processed by the handler associated with the affected group. One of the main goals of using internal events is to trigger the execution of scripts that check the consistency of the components that belong to the repository. For instance, the event

that is triggered whenever a new element is inserted in a view, can be used to if it has the attributes required for that view.

External events reflect events associated with one or more managed objects. External events can be propagated within a map and across all maps of a repository. There are three sub-classes of external events that reflect different policies for event propagation, namely: *ascendant* events, *descendant* events, and *lateral* events.

Ascendant events are propagated upwards in the group hierarchy defined by the map in which they occur. At each hop, the reception of the event may trigger an appropriate handler. The handler may consume the event (which is not propagated further) or forward the event to the next level after processing. In any case, the propagation of an ascendant event terminates at the map element. Ascendent events can be used to specialize or generalize the processing of specific occurrences: the event can be processed by some specialized handler associated with the element or be propagated upwards and be processed by a global handler that handles all events generated by the lower layers.

Descendant events are similar to ascendant events except that they are propagated in the opposite direction, from the group in which the event was raised downwards until a core-element is reached. In the core-element, the descendant event can raise a set of ascendant events in all elements representing that core-element in different maps. This mechanism eases the task of applying some script to all elements included in a given hierarchy. Additionally, this mechanism supports the propagation of events across maps.

Finally, lateral events are propagated across the links that connect the elements of a map. Since these links may form cycles, the event propagation is limited to a configured maximum number of hops. This mechanism can be used to correlate events. For instance, one can correlate the failure of a components to the failure of other objects connected to that component through the propagation of failure notifications through the established connections.

Figure 2 illustrates the event propagation policies defined in our event model.

The occurrence of Event 1 at the element *S*1 can result in the following sequence of events, depending on the event sub-class:

- If the event is *ascendant*, it will be handled at the level of the element S1, propagated to the group G1 and, possibly, further propagated and handled in the map V1. Note that for many events, a generic handler can be defined at the top level, in the map component. In such case the event is simply propagated to the root and processed there. However, the architecture supports the definition of more specific context-dependent handlers in the lower levels of the hierarchy.

- If the event is *descendant* it can be handled at the element S1 and later handled by the core-element C3. In the core-Element the event could be propagated to the remaining elements that represents that core-element.

- If the event is *lateral* with a maximum number of hops set to 1, it will be only propagated to the element *S*3.

Consider now the occurrence of a descendant event, Event 2, at the level of the map V1: the event will be propagated among elements belonging to map V1 including the core-elements that have elements in V1. Furthermore, the core-elements may raise an ascendant event that will be propagated to the elements in the map V2.

The proposed model for event propagation allows:

- to represent in a configurable and context-sensitive manner changes in the status of managed objects, in all active maps that include a element of that core-element (ascendant events).

- to actuate in a large set of items without explicitly naming all the targets (descendant events).

- to propagate the occurrence of events to all elements that are physically or logically connected (lateral events).

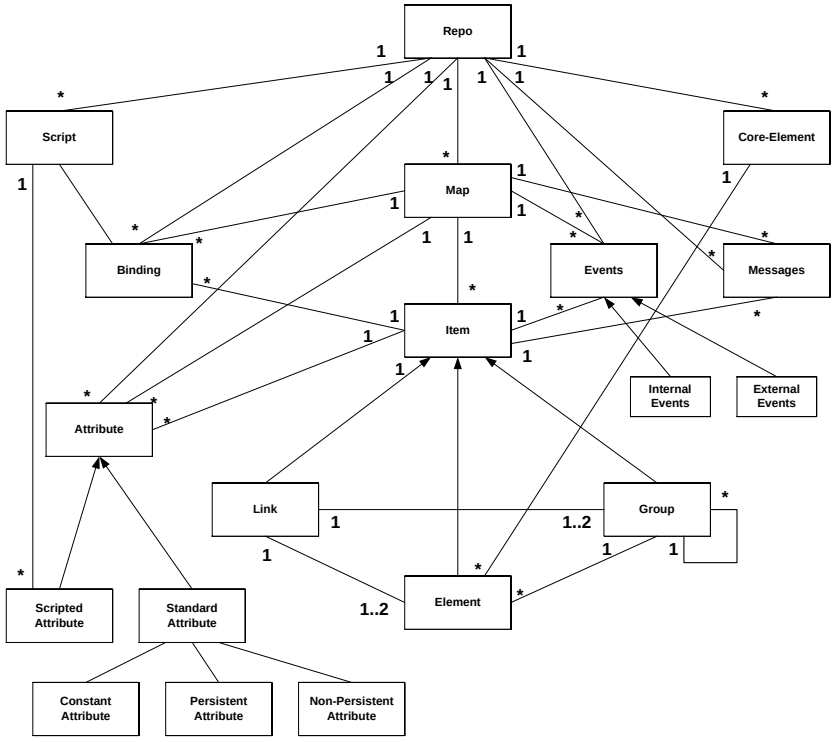

Fig. 3: Data model of `Tnm::repo`

Note that by defining a chain of events and handlers it is possible to model complex propagations of events as suggested in [OMKY97]. However, since our approach does not relies on defining a new language, propagation schemes based on the composition of the three basic modes have to be explicit coded in the handlers while the MODEL [OMKY97] allows these propagation policies to be expressed in a declarative fashion.

5 Implementation

In order to validate experimentally our model, we have implemented a prototype as a Tcl extension to the Tnm/Scotty platform [SL95, SL93]. The Scotty platform is a open source network management system that offers a Tcl interface. This system has a large user base since it provides an open and convenient platform for the development of customized network management solutions. The Scotty platform also includes a Tcl module, the `Tnm::map` [Sch], that intends to support the representation of the managed system. We have opted to implement our model as an extension to the existing `Tnm::map` module.

We have called our new Tcl module `Tnm::repo`. The new module preserves the functionality supported by the original `Tnm::map` module and implements a set of new functionality that support the model proposed in this paper, namely the support for definition of different maps and extended event propagation mechanisms. Figure 3 presents the data model of `Tnm::repo`.

The advantages of implementing our model as an extension to the `Tnm::map` are twofold. To start with, it allows us to reuse much of the management scripts already available for the Tnm/Scotty platform. On the other hand, it promotes the dissemination and usage of our prototype by others. To be compliant with the terminology used in the implementation of `Tnm::map`, we have opted to maintain some of the terms used in the original implementation. Therefore, in the following text a *repository* is simply designated by the keyword `repo`, a *map* by the keyword `map`, a *core-element* by `celem` , a *element* by `elem`, a *group* by `group` and a *link* by `link`.

The representation of any network is initiated with the creation of a `repo` object. The `repo` encapsulates all the `celem`s, `script`s and `map`s that are stored on that repository. Consider the following examples:

Example 1.

```
# Creating a repository
set repo1 [Tnm::repo create -name repo1]

# Creating Core-Elements
set pc001 [$repo1 create celem -name pc001]
set pr001 [$repo1 create celem -name pr001]

#defining attributes
$pc001 attribute set :MachineType "pc"
$pr001 attribute set :MachineType "printer"

# Creating a map explicitly
set map1 [$repo1 create map -name allpc]
set map2 [$repo1 create map -name allprinters]

# Creating an implicit map
set map3 [$repo1 create map -name allelements -filter [list :MachineType != ""]]

#Creating a script
#Note: the instruction "set function1 true" sets the return value of the script
set script1 [$repo1 create script -name function1 -script {
    if { "%T" == "repo" }{
        set function1 true
    } else {
        set function1 false
    }
}

#Managing attributes
$repo1 attribute set :Global:Test1 "test123"
set result [$repo1 attribute get :Global:Test1]
$repo1 attribute unset :Global:Test1

#defining a method
$repo1 attribute set :Global:Var1 -script $script1

#listing repo info
set myrepomaps [$repo1 info maps]
set myrepocoreelems [$repo1 info celems]

#removing the repository
$repo1 destroy
```

Each network component is represented in the management system by a single *core-element* (`celem`). This entity is responsible for storing all relevant information that describes the network component. On a `celem` object one can perform operations to: read and write its attributes, obtain information about the relevant relationships (which repositories and maps are associated with the `celem`, which are its elements, etc), as illustrated below:

Example 2.

```
#Processing of attributes
$pc001 attribute set :SysType == pc
set result [$pc001 attribute get :SysType]
$pc001 attribute unset :Global:Test1
```

The definition of `scripts` at the level of the repository allows code to be reused in the handling of events and in the implementation of management tasks. The scripts are interpreted at run-time and can include wildcards (a '%' followed by another character) that are bound when the script is invoked. These wildcards permit to access run-time information such as: the token of the entity associated to the script (`%H`), type of entity associated with the script (`%T`), name of the attribute (`%V`), etc. For instance, in the script `function1` of the previous example, the wildcard `%T` would be replaced in run-time by the value `repo` (and the script would return true).

Note that these scripts need to be explicitly invoked through the attribute to which they are bound. Is is also possible to execute scripts by defining event handlers, as it will be described below.

The `map` object supports operations such as the access to its attributes, creation of new elements (`groups`,

`elems` and `links`), to list enclosed elements, etc:

Example 3.

```
# Creating a Element
set elem001 [$map1 create elem  $pc001 -name pc001]
set elem002 [$map2 create elem  $pr001 -name pr001]

# Creating a group
set f1 [$map1 create group -name group1]

# Creating a link
set l1 [$map1 create link -name link1]

# Listing contents
set listelems [$map1 info elems]

# Removing a map
$map1 destroy
```

Note that the explicit creation of new elements is only allowed in explicit maps. In the implicit maps elements are created automatically using the rules defined in the filter associated with the map. The view contents is updated when the view is created or when the `update` parameter is explicitly invoked.

As noted before, the set of attributes that are visible in a map is limited by a constraint that can be defined at the map level:

Example 4.

```
# Deny the access to the attributes of all elements in the map
# whose name starts with ":Global:Private"
$map1 configure -constraints [list deny :Global:Private]
```

There is also a set of operations that can be performed on most items such as: reading and writing attributes, read information about the item relations (to which map they belong, etc). There are also some operations that are specific to each item. For instance, from the elements one can extract the core-element it represents; from a group we can obtain the parent group, etc. Some examples are given below:

Example 5.

```
# Associate an item to a group
$item configure -group $gr1

# Get all elements in a group
set objs [$gr1 info members]

# Connect to items using a link
$lnk1 configure -src $item1 -dst $item2

# Get the endpoints of a link
set source_item [$lnk1 cget -src]
set dest_item [$lnk1 cget -dst]
```

As we have noted before, a fundamental feature of our model is the support provided for event propagation. Events are managed in three distinct phases: definition of scripts, definition of event handlers (binding scripts to events) and raising of events. When an event handler is defined, a script is bound to the event tag; such script will be executed when the event reaches the object where the handler is defined. The occurrence of an event is triggered using the *raise* command. The following example shows how an event handler can be defined and triggered.

Example 6.

```
# Creating an script
set scr2 [$repo1 create script -name eventhandler1 -script {
    if {%T == elem } {
      %H attribute set:Global:Error True
    }
}

# Defining the handler for an event
$item1  bind :Events:Error1 -script $scr2 -up

# Raising an event
$item1 raise -up :Events:Error1
```

Note that only external events can be explicitly raised. Internal events, that are bound to well known predefined names, are raised automatically by the platform. For instance, the following command is invalid:

Example 7.

```
# Invalid command, returns an error
$repo1 raise -up :Repo:InternalEvents:CreateCelem
```

In order to validate our model, we have implemented a number of extensions to the `Tnm::map` package, including: the introduction of the view component which offers the possibility of specifying constraints at the view level, the extension of the event propagation model, and the characterization of each entity through attributes and methods.

The resulting `Tnm::repo` extension is independent of the actual method used to interact with the network components being managed. The integration of the `Tnm::repo` with existing network management protocols is made at the level of script definition. At this level, all the available TCL extension for network management, such as those provided by *Scotty* (SNMP, ICMP) or by TCL (HTTP, *sockets*, etc) can be used. This separation of concerns simplifies the tasks of defining the maps in an abstract manner, independently from the concrete protocols used at the device level.

6 Evaluation

In order to evaluate the model proposed in this paper, we have applied the prototype to a concrete network. Additionally, we have compared the resulting management system with the systems obtained from the application of other alternative models, namely the `Tnm::map` and the *HP OpenView*, to the same network. A more detailed report of this experiment can be found in [Pal00].

The target network, depicted in Figure 4, was chosen to allow the definition of three different views of the same managed system: the view used by the managers of the *MS Windows* systems, the view o the *Unix* managers, and the view of the networking components managers. Each of these views includes different information about the target network. The view used by the managers of the *Windows* system must only have access to information about machines running this operating system. However, it should also have access to the location of these machines ins terms of LAN segments, default gateways, servers, etc. The view of the *Unix* managers is similar but covers machines running that specific OS and several particular attributes of this platform. The view of the networking infrastructure managers provides access to information about all inter-networking components (HUB, switches, routers, etc) but no specific information about how the OS of each machine is configured. Note that some components, such as the default gateways need to be included in more than one view, with different levels of detail.

The advantages of our model become more clear when we try to model the target network with the other platforms:

- In the *HP OpenView* the OpenView Topology manager may be used to group the information according to different views. However, there is no way to restrict the way the object attributes are seen in the different views. The OpenView Postmaster service can also be used to implement the event model. However, the only feature that can be specialized for each view is the color of the icons used to represent the objects. On the contrary, our model allows full specialization of event handling according to the view context.

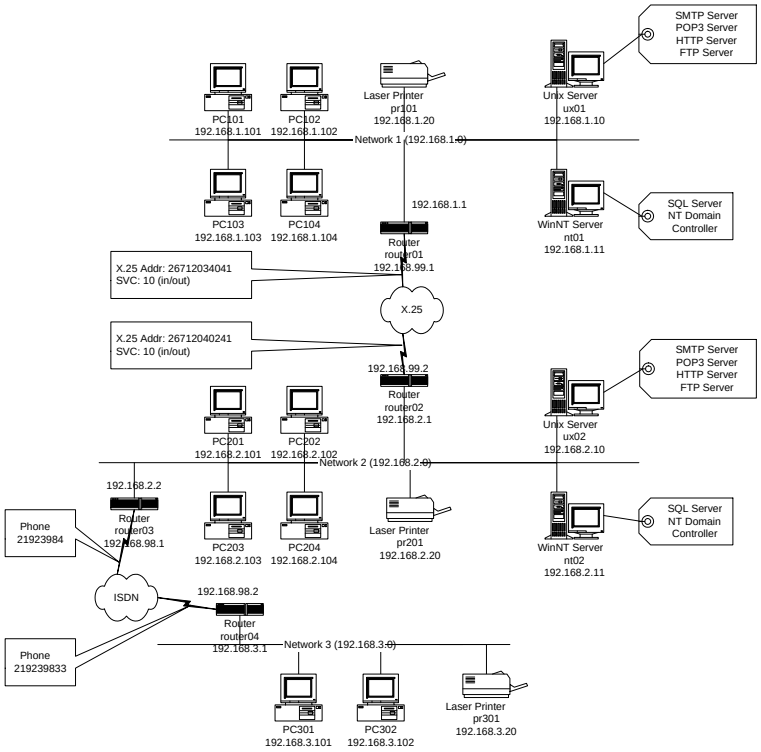

Fig. 4: Evaluation network

- In the `Tnm::map` one can present different attributes of the same object in different maps through the creation of independent `maps`. However, when several maps are created, the consistency of the information provided to the users of different maps is no longer ensured. To enforce the desired level of consistency, the user must create complex scripts that explicitly update all `maps` whenever a change occurs in any of the maps.

- Naturally, our prototype did not suffer from these limitations. Using `tmn:repo` it was possible to model the three management views, with specialized constraints and event handling procedures, without compromising the consistency of the information provided regarding the same atributes in different views.

Since our prototype is based on the original `tmn:map`, we were able to measure the performance degradation resulting from the additional complexity required to support our abstractions [Pal00]. When doing comparative tests to measure the time needed to read or update an attribute of a single element of a map, we have observed that our implementation executes 6.5% less operations per minute than the original `tmn:map`. This overhead is due to the additional tests required to validate if the attribute is visible in that view. We believe that this performance degradation is perfectly acceptable when compared with the functionality gains supported by the model.

7 Conclusion

The evolution of the information systems, both in terms of size and heterogeneity, has created the need for management tools that, at the same time, allow to obtain a global perspective of the complete managed system but, on the other hand, allow to define selective views tailored to specific management tasks.

We have proposed and implement a new model that satisfies these new requirements. Although some of the aspects covered by our model have been addressed in an independent way in previous systems, our

proposal has the merit of integrating old an new features in a coherent whole. The main properties of the proposed model are:

- A clear separation between the devices, that represent managed objects, and the entities that represent the context in which the object can be managed (this feature is supported in some commercial platforms such as HP OpenView);
- An event model that allows to selectively define the behavior of each component with different levels of detail (inspired in the event model offered by the `Tnm::map` module);
- Mechanisms that allow to express access control policies (included in the system Domains);
- The possibility of each view to define constraints on the device attributes that are visible in the view;
- A new and powerful set of event propagation policies (ascendant, descendant and lateral) that can be used to preserve the consistency of the information provided across multiple views.

We have developed a prototype of our model as an extension to the Tnm/Scotty system and we have applied this prototype to some concrete network examples. The experience has shown that complex management structures can be represented and implemented in a much more elegant and efficient manner using a model that, as ours, offers a well defined view abstraction.

References

[Ass] Computer Associates. *Unicenter TNG SDK Programmer's Guide*.

[BMHRS96] Frank Buschmann, Regine Meuniera, Peter Sommerlad Hans Rohnert, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Son Ltd, 1996.

[Gol96] G. Goldszmidt. Network management views using delegated agents. *Proceedings of the 6th IBM/CAS Conference, Toronto, Canada*, 1996.

[KSSZ97] P. Kalyanasundaram, A. Sethi, C. Sherwin, and D. Zhu. A spreadsheet-based scripting environment for SNMP. 1997.

[MS93] J. Moffett and M. Sloman. User and mechanism views of distributed systems management. *IEE/IOP/BCS Distributed Systems Engineering*, 1993.

[OMKY97] D. Ohsie, A. Mayer, S. Kliger, and S. Yemini. Event modeling with the model language. In *Proceedings of the 1997 IEEE Integrated Management*, San Diego, CA, USA, 1997.

[Pac] Hewlett Packard. http://www.openview.com.

[Pal00] C. Palma. Suporte para vistas em ambientes de gestão de redes. Master's thesis, Faculdade de Ciências da Universidade de Lisboa, 2000. (in Portuguese, available at http://www.di.fc.ul.pt/˜ ler/students/carlospalma.html).

[Sch] J. Schönwälder. *Scotty User Manual*. http://wwwsnmp.cs.utwente.nl/˜ schoenw/scotty/.

[SL93] J. Schönwälder and H. Langendörfer. Ined - an application independent network editor. Technical report, Technical University of Braunschweig, Germany, 1993.

[SL95] J. Schönwälder and H. Langendörfer. Tcl extensions for network management applications. In *3rd Tcl/Tk Workshop*, Toronto, July 1995.

[SM89] M. Sloman and J. Moffett. Domain management for distributed systems. Technical report, 1989.

[Yeo90] W. Yeong. Snmp query language. Technical report, Performance Systems International Inc, 1990.