Hasso-Plattner-Institut für Softwaresystemtechnik GmbH
Fachgebiet Systemanalyse und Modellierung
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam

Bachelor's Thesis

Central Event Manager for the Eclipse Modeling Framework

Philipp Berger

June 25, 2010

Supervised by Andreas Seibel

Abstract

Today, in industrial model-driven software development settings, hundreds of developers are working on a huge number of different and related models. Constraint definitions in models are used to preserve consistency within and among them. With increasing complexity of models, consistency checking of models gets more and more resource and time consuming, which hinders its effective application during modeling. An incremental approach for consistency checking considers only updated regions of a model, which enables integration of modeling with instantaneous consistency checking.

This incremental consistency check has been implemented in the *Scalable EMF Impact Analyzer Framework*. This thesis describes preconditions and different concepts towards an event management component for this framework, including filter and event structure, transactional environments, and event mapping mechanism. Also, two different approaches are described and the resulting implementations get compared.

Contents

1 Introduction

The ever-increasing complexity of software and software development processes is undeniable. The primary goal of Model Driven Software Development (MDSD) is to raise the level of abstraction at which a developer operates and simultaneously reduce both the amount of developer effort and the complexity of the software. Abstraction has a long history in the domain of software development. Starting with the introduction of assembly language as abstraction over machine code, followed by third-generation languages and objected-oriented languages. Each abstraction layer has two effects: reaching higher quality and productivity, and the creation of a common language among users, which is closer to the actual problem domain. MDSD similarly abstracts from different software artifacts like the programming language from the underlying machine code. To guarantee the ability of realization and other characteristics, the set of models can be controlled by a set of consistency specifications, so called *constraints*, which are defined at a meta level. Like consistency specifications in traditional databases, it is desirable to ensure the consistency among models by some kind of checking mechanisms at every observable point.

In today's industrial environments there are large numbers of models with various consistency constraints defined on it. Therefore, consistency analyses get very resource and time consuming. Additionally, there is a high number of developers editing the models. Due to the high frequency of changes and the fact that each change potentially invalidates a model, an efficient checking approach is required to avoid the time consuming analysis for actual unaffected constraints. This implies an incremental approach to checking constraints on models.

A component, which realizes incremental constraint checking, is provided by the proprietary Model Infrastructure (MOIN, [1]). MOIN is an established platform for a number of modeling tools developed by SAP AG[1]. Although, there are attempts to transfer diverse MOIN specific tools to the Eclipse Modeling Framework (EMF[2]). EMF is, like MOIN, a modeling framework and code generation facility for building tools and ap-

[1]http://www.sap.com/
[2]http://www.eclipse.org/modeling/emf/

plications based on models. EMF gets support by the Eclipse community and offers a diversity of extensions. This is the motivation for the transfer from MOIN to EMF, which enables SAP to contribute tools to a wide range of developers to test, improve, and integrate features from the open source community.

The goal of the Scalable EMF bachelor's project is to transfer the impact analyzer component of MOIN to EMF. This component offers an incremental approach to the evaluation of consistency constraints specified in the Object Constraint Language (OCL[3]). It is used to reduce the set of OCL expressions an application has to reevaluate to ensure the correctness of a model.

The operating principle is divided into two phases. During the first phase, called class scope analyzes, the impact analyzer maps an OCL expression to a set of internal events [1]. This mapping is used to identify the different change categories, which affect the analyzed statement. The second phase, called instance scope analysis, is about the navigating over OCL expressions to identify affected model elements. Hereby, the impact analyzer needs to know the location of the change in the model. Both phases have been implemented in two components named Filter Synthesis (bachelor's thesis of Tobias Hoppe [3]) and Instance Scope Analyses (bachelor's thesis of Martin Hanysz [2]).

These two components have structural assumptions on the enclosing environment. Firstly, a data structure is needed to specify the change categories and exchange them with the environment. Secondly, the environment has to offer proper change notifications that point to the change location. Thirdly, the environment should provide a fast and reliable OCL evaluation infrastructure. During our bachelor's project we investigated the mapping from OCL expressions to MQL expressions. Although EMF provides an reliable OCL infrastructure, the evaluation component for MQL named Query2[4] offers faster results, especially for huge instances, due to the use of a model index (bachelor's thesis of Thea Schroeter [4]). This bachelor's thesis is about the implementation of a central event management component for EMF that fulfills the assumptions of the impact analyzer concerning change notifications and categories. These four components provide the *Scalable EMF Impact Analyzer Framework* that has been designed and implemented during our bachelor's project.

[3]http://www.omg.org/technology/documents/formal/ocl.htm
[4]http://www.eclipse.org/modeling/emf/?project=query2

2 Background

2.1 Definition of Event Management

In general the main task of event management in the context of software applications is informing an event listener if any relevant change occurs. An event listener, or event handler, is a method or subroutine, which defines how the program should react on arising of relevant events. The event denotes a change on an event source. Event management is used in many different domains beside MDE, for instance synchronize-objects in operational systems. The designs of event-driven systems vary widely because of the different requirements they have to face. In general one can identify two approaches towards event managing, which are explained in the following.

2.1.1 Decentralized Approach

An event listener gets directly registered to an event source. Therefore, before the registration of the listener all relevant sources should to be known. Alternatively, listeners can be later attached to new sources during the execution of the program. In general, all event sources have to be identified before events occur. Likewise applications have to unregister the listeners manually. In case that an event occurs, the listeners get directly notified by the event source and analyze the event and react if any action is required. For example, in Java one can register an listener thread with the method `wait(100)` on a random object. When another thread makes the object notified, the listener thread will wake up and continue executing. Otherwise, the listener thread will wait 100ms and continue the execution.

2.1.2 Centralized Approach

As contrast, an application does not know an event source before an event occurred. The application directly registers on an event managing component of the environment henceforth referred to as event manager. The event manager handles all event sources as a black box. In case of an event the event manager will delegate it to all registered listeners. One has to add that an event manager is usually defined for a *scope* of event sources defined in different ways, e.g., all events that occurred on a html div-element and all containing elements. An event manager itself registers on all event sources in scope and unregisters from an event source, once it gets removed from the scope. In addition, some implementations of event managers offer the opportunity to limit the set of events sent to the listener on registrations. This is done by passing so called event filters. Event filters vary widely over the different implementations, starting with passing just an event type name through to passing logical linked complex event filters. Overall, event filters define event categories with specific characteristics.

2.2 The SAP Modeling Infrastructure (MOIN)

The goal of SAP's MOIN project is to implement the consolidated platform for SAP's next generation modeling tools. For example, it is basis for FURCAS, which defines textual view with respect to underlying domain models.

2.2.1 Overview

From a bird's eye perspective, MOIN is composed of six components. These components are the repository, the query mechanism, the commands, the eventing framework, the model transformation infrastructure and the MOIN Core. Overal,l MOIN is a comprehensive repository infrastructure to manage MOF-based models.

2.2.2 MOIN Eventing Framework

As mentioned above, MOIN has a centralized event management approach. There is an event manager that will be triggered by the repository for each event raised during the context of one session or connection. MOIN differs between several events, some of these (e.g., `PartitionChangeEvent`) are platform-specific. All platform-independent event types are visualized in figure 2.1. The names of the event types are self-explanatory. For instance, an `ElementCreateEvent` denotes the creation of a model element. In addition, the eventing framework consists of a number of event filters. Each filter defines its own matching mechanism for an event. For example, the class filter holds a reference to a `MOFClass` and a `Boolean`, which signals whether subclasses should be included or not. Therefore it matches, for example, an `ElementDeleteEvent` that denotes the creation of an instance from the wanted class. A special class of filters are the `LogicalOperationFilters`. Each subclass holds a set of filters, which semantically get combined under the operation of a filter (`AndFilter`, `OrFilter`, `NotFilter`).

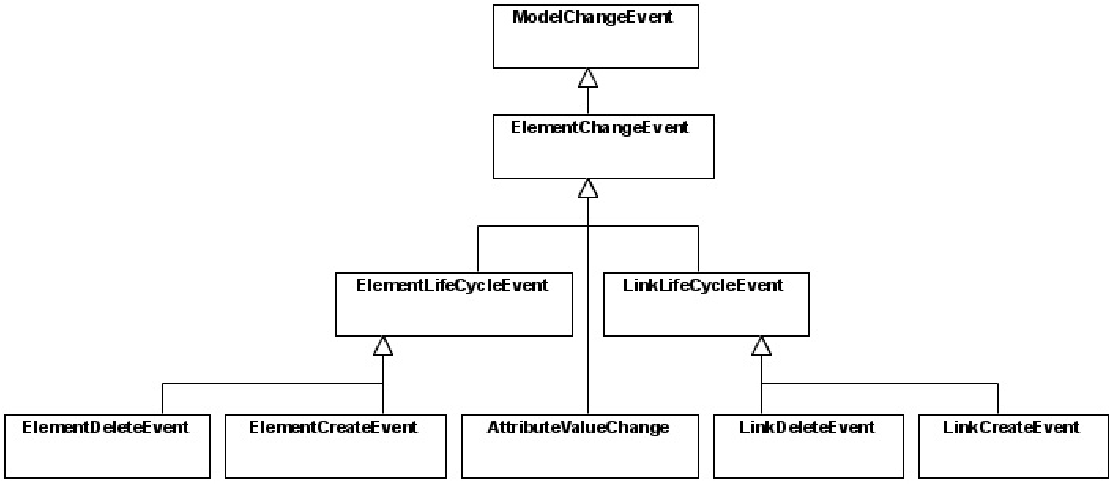

Figure 2.1: Class diagram of MOIN ModelChangeEvents

For example, the `AndFilter` only matches if all contained filter matches the current event. During the course of MOIN, two event filtering mechanism were developed. Firstly, a simple matching algorithm got introduced. This algorithm is based on a traversal matching over the event filter tree. Hereby, each filter implements a `matchesFor(Event)` method. Secondly, a complex filtering mechanism got introduced, which is based on the usage of hash tables, where each filter has to define a filter criterion as key of the map. These algorithms get adapted to EMF and are discussed in detail in section 3.3. Furthermore, MOIN enables applications to register on two different scopes. On the one hand, an event listener can register on the connection's registry, which means that the filter is unregistered once the connection is closed. On the other hand, the application is able to register on the session's registry, which enables listener to be active over several connections.

2.3 Eclipse and EMF

Eclipse is a modular and easy extensible Java-based development platform. It is used and extended in various ways. Especially, it supports model-driven development implemented by the Eclipse modeling framework (EMF[1]). The framework delivers similar to MOIN diverse features and is the basis for many other components.

[1]http://www.eclipse.org/modeling/emf/

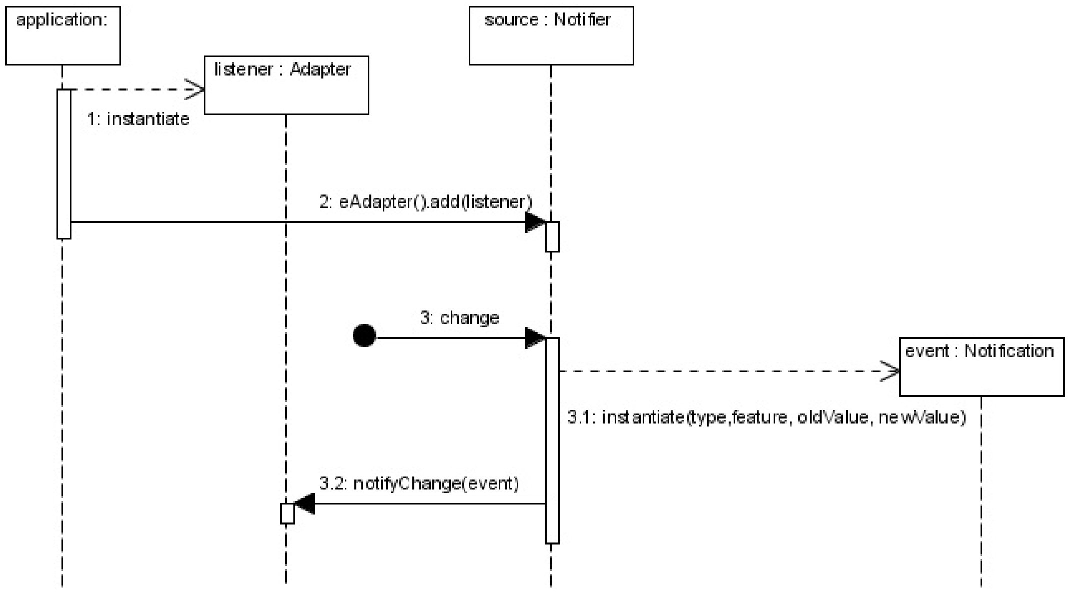

Figure 2.2: Sequence Diagram for the common use of Adapters in EMF

2.3.1 EMF Notifications and Adapters

As contrast to MOIN, EMF has no central event management component. Instead, there is a decentralized approach to event management. In EMF, events are called notifications and event listeners are called adapters. All implementations of the Notifier interface in EMF can serve as event sources. The important notifiers are `EObject`, `Resource` and `ResourceSet`, because these guarantee that all model elements can be event sources equally to MOIN. An `EObject` is in EMF a model element that is contained in one `Resource`. A `Resource` is a persistent document, which is loaded in the context of a `ResourceSet`. A `ResourceSet` is in EMF used to define scopes for all application like the generated editor. As shown in figure 2.2, the common usage of the EMF adapter concept is to add an `Adapter` to the `eAdapters` list of a notifier. In case a change on the notifier occurs, the notifier instantiates a `NotificationImpl` and sets attributes to the matching values for the current change. Afterwards, the notifier iterates over all owned adapters calling the `notificationChanged`-method and passes the generated notification. The most interesting methods of the `Notification` API are shown in figure 2.3.

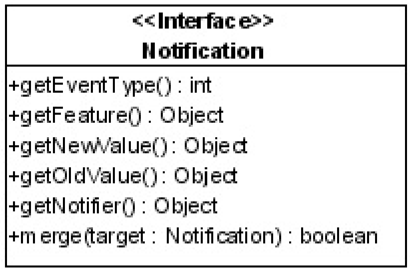

Figure 2.3: Class diagram showing selected methods of the notification interface

`Object getNotifier()`	Return the event source of the notification
`int getEventType()`	Returns the event type
`Object getFeature()`	Returns the type of the event type
`Object getOldValue()`	Returns the old value (`DataType`, `EObject`, list)
`Object getNewValue()`	Returns the new value (`DataType`, `EObject`, list)
`boolean merge(Notification)`	Merges two notification if possible, returns true if successfully merged

In comparison to MOIN, the EMF event types are very similar. The EMF event types are defined as constant `Integers` in the Notification interface (`ADD`, `ADD_MANY`, `SET`, `UNSET`, `REMOVE`, `REMOVE_MANY`) As difference to MOIN, EMF supports no `ElementLifeCycleEvents` at all. Although there is an event type `CREATE`, which is unfortunately deprecated and no longer in use. Furthermore, platform specific events of MOIN, like `PartitionChangeEvents`, have no counterpart in EMF.

2.3.2 The EContentAdapter

The `EContentAdapter` is a special adapter implementation of EMF, which adapts itself to the notifier and all contained objects. In addition it registers to new children and unregisters from objects that get removed from the containment hierarchy. Therefore, applications are able to register on a complete composition tree spanned by `Resources`, `ResourceSets` or `EObjects`. One has to mention that the `EContentAdapter` only properly works, if one adds the adapter to the list of all `eAdapters` from the target. Using the `setTarget` method will not activate the traversing of the composition tree.
Moreover, it is not trivial to unregister an `EContentAdapter` from all registered `Notifiers`, because the target of the adapter changes once the composition tree changes. In addition, on manually unregistering the `EContentAdapter` from a random notifier, the adapter gets

only deleted from the containment tree spanned by the specific element and not from all known notifiers. For that reason, every application has to hold the root notifier. Thus it is able to remove the `EContentAdapter` from the `eAdapters` of the root and unregister it successfully.

2.3.3 EMF Transactional Environments

A model transaction comprises a unit of work performed within a modeling infrastructure against a model. Transactions provide an *all-or-nothing* proposition, stating that work unit performed at a model must either complete in it entirely or have no effect at all. As a consequence, all changes contained in one transaction are combined in one atomic unit of work. This enables the modeling environment to ensure consistency of models and isolation between multiple users. There are two different approaches, to work with transactions in EMF.

EMF Transactions. EMF, like MOIN, natively implements a transaction API named EMF Transactions[2]. All changes made to a model are executed as so called `Commands`. Each `Command` contains a set of changes to the model and offers a revert method that undoes each effect of the command. An application sends `Commands` to a so called `EditingDomain`, which contains a set of `Resources`. Each registered `Resource` is managed by the `EditingDomain` that provides isolation and atomicity for the transactions. One has to mention that the EMF Transactions also offer an event management component. This component provides a filter concept, which could be used to filter simple properties of notifications. However, the set of filter is less expressive than expected by the impact analyzer. The event filtering is executed in a naive way, that is to iterate over each registered filter and match it to the current notification. In case a match occurs, the affected listeners get looked up by the matched filter. It supports two types of listeners, which can be registered. Firstly, a `PreCommitListeners` interface is offered, which get notified for all changes during a transaction just before the model is affected durably. Secondly, it supports `CommitListeners`, which get notified right after the changes of a transaction take effect. Unfortunately, it does not support so called `ChangeListeners` like implemented by MOIN. `ChangeListeners` get notified for every change in a transaction in the moment the change is executed. Therefore, applications can monitor the development of changes during one transaction. Similarly to MOIN, EMF Transactions support a veto mechanism, which gets triggered on raising a RollbackException dur-

[2] http://www.eclipse.org/modeling/emf/?project=transaction

ing the execution of a transaction. Therefore all changes made by the transaction get reverted.

CDO Transactions Furthermore, add-ons for EMF offer different transaction concepts which do not build on the native EMF transaction API. For instance the Connected Data Objects (CDO[3]) Model Repository comes with an own `CDOTransaction` class and an own event management component. In comparison to MOIN, EMF extended with CDO works quite similar, because there is also an `CDOSession` that behave equally to MOIN like a scope for EMF applications.

[3] http://www.eclipse.org/modeling/emft/?project=cdo

3 Central Event Management Component for EMF

The event manager is essential to the *Scalable EMF Impact Analyzer Framework* for interacting with the enclosing modeling infrastructure EMF. As shown in figure 3.1, interacting with the event manager happens once the application receives the event filter from the impact analyzer (IA). The application directly registers with this event filter on the eventing framework that internally handles the subscription. Just as an event occurs on the underlying EMF infrastructure the event manager delegates it, in case the registered filter matches, to the application. Afterwards, the application will immediately delegates this event to the instance scope analysis of the IA. Therefore, one can identify two essential requirements in the context of the IA. Firstly, the event filters have to enable the IA to express all information extracted from OCL expression in the filter tree. Therefore, one have to offer similar filters to MOIN. Secondly, the eventing framework has to ensure that all events delivered to the application, including the IA, are equally conclusive.

In the following chapter, one gets an insight to concepts and implementation details of the two fully implemented event management components. Especially, one emphasizes their differences concerning event filtering. Furthermore, an outline of the concepts towards a transactional support for the event manager is given. In general, one can identify the following core functionalities for an EMF event management component. Firstly, the Event Mapping component is needed. This component mainly concerns compatibilities to MOIN and improvements concerning the information depth of EMF notifications. Possibilities are presented, how to map sets of incoming EMF notifications to a set of the same or different events, which could minimize the subsequently calculation effort. Secondly, the registration functionality, which is used by applications to register a pair of an event filter and an event listener on the event manager. This also includes the mechanics that registers the event manager to all model elements in scope. Thirdly, the different approaches to event filtering are discussed. In particular, the dependencies between the event filtering and the two other components are shown.

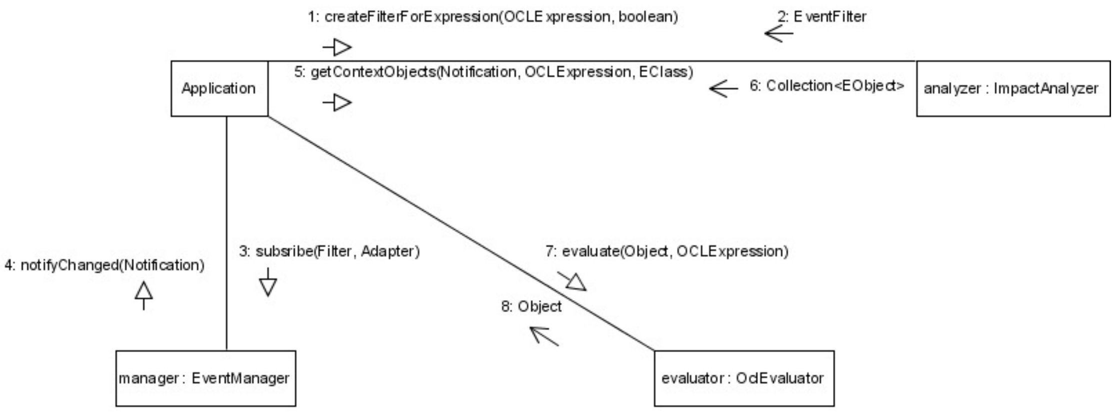

Figure 3.1: Communication Diagram for the common interaction between Impact Analyzer, Application and Event Management

3.1 Event Mapping

In this section, the possibilities for the mapping of events are discussed. As mentioned above the Event Mapping has to generate for the impact analyzer a equally conclusive events to MOIN. Therefore, the mapping of single EMF notifications is necessary. But with the addition that the mapping of element creation or deletion events to EMF is not trivial. This mapping profits if a single notification is brought in context with all events occurring during one change at a model. This and other enhancements concerning the consolidating of event sets get discussed in section 3.1.2.

3.1.1 Mapping of a single EMF Notification to MOIN like events

As shown in figure 2.3, an EMF notification consists of an event source object (usually an `EObject`), a feature, an old and a new value, and an event type. On interpreting these properties, applications have to do intensive calculations. For instance, applications traverse from the event source to affected `EClass` to identify the type of an element. In addition, more complex operations are imaginable like the investigation of all types in the list of all new values. The goal of the mapping is to pre-calculate values from a notification to decrease the computation effort of applications. A prerequisite is, however, that all affected applications are interested in similar values and that the computation of values takes a significant effort. Due to the relative rare execution of the event handling routines of application, which is mainly reasoned by the use of event filters, this computation effort can be disregarded. Nevertheless, in the context of the event manager a mapping is definitely necessary to provide similar conclusive event filter.s This mapping is actual located in the internal matching algorithms as presented in section 3.3. In addition, a mapping from the MOIN event types to the EMF event types is needed, because that is the basis for the impact analysis. Therefore, one realize a `NotificationHelper` class which offers methods to map properties of EMF notification to MOIN. These functions are inter alias `isAttributeValueChangeEvent`, `isLinkLifeCycleEvent` and `isAddEvent`. Essentially, the two first methods matches the feature type of the notification and the last one matches the three different event types indicating an element add on a feature (`ADD`, `SET`, `ADD_MANY`).

Defining an Element Creation in EMF. Furthermore, one has to mention that two functions, `isElementCreateEvent` and `isElementDeleteEvent`, cannot be mapped obviously to the properties of an EMF notification. Hence, in EMF no explicit create notification is supported. Although, there is in the interface of the notification a `CREATE` event type defined, but is declared as deprecated, thus, is no longer in use. Consequently, one has to create another possibility to identify an element creation. On the one hand, an customization of the generated EMF factories is a possible solution approach. Hereby, the generator templates of EMF get customized thus that after each instantiation of an element a notification will be raised to each adapter, which is registered on the factory. Factories are singletons and valid for the whole execution environment. Therefore, one is able to clearly identify all element create event in the scope of one complete Eclipse instance. On the other hand, the first approach got presumable refused by the EMF community, because they clearly removed this feature from the notification API. In addition, this approach only covers the element creation event, but not the element deletion event at all. Furthermore, one has to admit that not all applications are interested in element create events of the scope of the whole running EMF instance. For that reason one has to use another composition-based approach. Hereby, the attaching of an element to the composition tree get defined as the element creation. Likewise, elements detached from the composition tree are defined as deleted. In this way, a notification, which affects a containment-reference and is from an `ADD`-event type, signals that all elements, which can be extracted from the new value of the notification, are created. Likewise, a notification concerning containment feature and has a `REMOVE`-event type signals that all elements extractable from the old values of the notification are deleted. Hereby, one has to face another problem concerning re-linking of subtrees in a composition tree. As shown in figure 3.2, a number of elements can change their root containers by moving only one element from one container to another. This is especially problematic concerning the expected eventing from the IA, because the IA has to react on each new

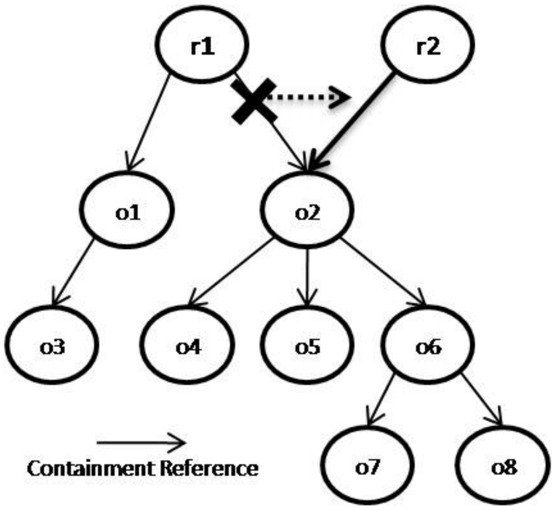

Figure 3.2: Visualization of moving a subtree of elements

element once it comes into the scope. There are two different solution approaches to the problem. Firstly, the logic to face this problem can be located in the filter synthesis of the clients. On creating a filter for an element creating event, the client has to figure out each containment reference the wanted element could be part of. In addition, all transitive containment references have to be identified, too. This also includes further computational effort once the client gets notified, to actually ensure that the wanted element take part in one of the signaled transitive containment references.
The second approach is clearly more intuitive for the client, because all necessary computation is located in the event manager. In the course of this, the client (e.g. IA) only has to use predefined event filters. These filters matches containment events concerning a `EClass`. To create all necessary containment events, a recursive traverser for the composition tree creates for each found element a new notification for the current containment feature. This is visualized in figure 3.3.

3.1.2 Consolidating of Event Sets in a Transactional Environment

As discussed in the previous section, we define an `ElementCreateEvent`, corresponding to MOIN, as an `ADD`-notification on a containment reference. This results in an signaled element creation for each add of an object. Even if this object already exists in the composition tree, because an detaching from one composition branch and attaching to another one does not result in an `MOVE` notification. Instead it results in a `REMOVE`-notification and an `ADD`-notifications, for the containment features. The normal

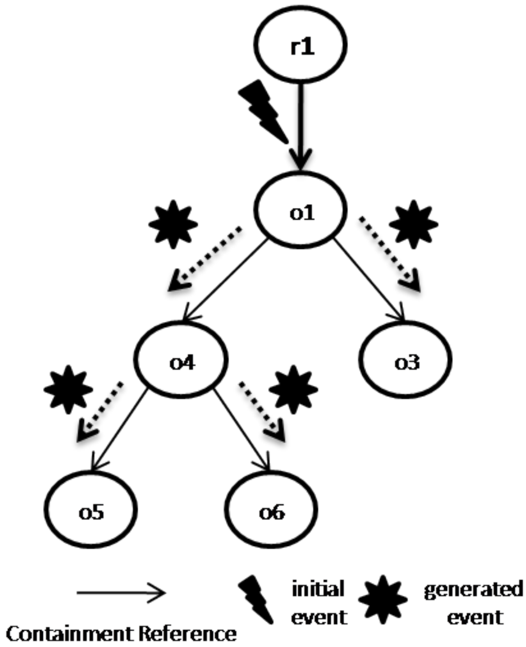

Figure 3.3: Visualization of traversion of the composition hierarchy

implementation of the event manager will handle both events separately. Therefore, all registered applications will receive a deletion and a creation for the same object. This can be avoided by consolidating both notifications to one link move notification. Hereby, one has to identify combinable notifications. This is commonly done on the application-side. For this purpose, applications use transactional environments, because transactions define a set of operations as an atomic work unit. Hence, all events raised during one transition belonging together. This knowledge enables the event manager also to consolidate other sets of notifications. In general, one can map each set from the same type concerning the same instance as notifier to one sole notification. This notification has the old values from the very first notification of the transaction and the new values of the very last notification. Furthermore, it is possible to combine multiple add notifications for one feature of a notifier to one `ADD_MANY` notification that holds all new values. Equally, `REMOVE` notifications can be combined into one `REMOVE_MANY` notification.

Micro Transactions. As discusses in section 2.3.3, there different concepts of transactional environments in EMF. As difference to MOIN, applications are not forced to execute changes in transactions. Consequently, applications can directly change the model without any use of a transactional environment. Therefore, we developed a concept called micro transactions. In detail, this concept is based on the setting delegate mechanism of EMF[1], which enables the application to override each setter and getter methods in a `EPackage`. Hence, the micro transactional component overrides each `eSet`-method in the package. This allow us to raise a micro transaction start-event before each setting and a micro transaction end-event after each setting. Therefore, the mirco transaction environment is able to clearly identify notifications resulting from one method invocation.

Listing 3.1: Inserting mirco transaction start and end to EMF setter methods

```
protected void set(InternalEObject owner, Object newValue) {
  MicroTransactionalEnvironment.startTransaction(owner);
  owner.eSet(eStructuralFeature, newValue);
  MicroTransactionalEnvironment.endTransaction(owner);
}
```

The micro mechanism is mainly developed to convert batched `ADD` and `REMOVE` notifications to sole `MOVE` notification. Due to the moving of a `EObject` is usually the change of a reference using a setter. The setter internally manipulates the `EOpposite` of the old and new value of the `EReference` . For this reason, an `ADD` and a `REMOVE` get raised. This offers in particularly for the IA, an enormous saving of computation time. Since the IA otherwise has to check all constraints for the whole moved subtree. Consequently, the change visualized in figure 3.2 results only in one `MOVE` event.

3.2 Event Registration

Since the API of all event manager implementation is identically, application can subscribe to each implementation in the same way. The event manager interface is shown in

Figure 3.4: Class Diagram of the event manager interface

figure 3.4. To use the event manager, applications first have to initialize a instance of the

[1]http://wiki.eclipse.org/EMF/New_and_Noteworthy/Helios#Support_for_Feature_Setting_Delegates

event manager with a given scope. Scopes in EMF are usually `ResoureSets` like used in the EMF editors. Consequently, applications define the scope by passing `ResourceSet` to the event manager. Afterwards, the `subscribe(EventFilter, Adapter)` is used to subscribe for non-transaction, so called live, events. Hereby, applications pass a filter, which matches on all events that should be delegated to the passed Adapter. Although, the event manager matches the filter against incoming notifications, only the actual matching notification is passed to the adapter. Therefore, applications should hold the mapping from filter to adapter separately if needed. Since the adapter is successfully registered, the adapter `notifyChanged(Notification)` method is called for each raising event.

The event manager does not guarantee that the adapter is prevented from garbage collection. This means that the application has to hold a reference to the adapter, to ensure this. Holding a reference to the adapter in the application code must be done anyway, because it is strongly encouraged to unregister the adapters as soon as they are no longer relevant.

3.2.1 Intern Management of Subscribed Adapters

The two implementations differ in the internal adapter management. The naive implementation, which uses the traversing of filter trees, saves the adapter and the filter in a dictionary which maps all passed filters to the adapters that has to be notified. As contrast, the second implementation (multi filter-table) extracts from the given filter tree all atomic non-logical filters. For this reason, the event manager first transforms the given filter tree into a disjunctive normal form. Afterwards, all filters will be sorted into their filter tables. Next, an internal registration object is created and actually a mapping from each atomic filter to this registration is created. This mapping enables the event manager to finally combine the atomic filters during the event filtering mechanism to decide whether an adapter should be notified.

3.2.2 Subscription of the Event Manager to Model Elements

For the subscription of the event manager one need to define the scope of the event manager. Theoretically, by using the `EContentAdapter` one can imagine each node of a composition tree including the `Resource` and the `ResourceSet` as scope. Although, one has to take care of the common use in EMF, which defines a `ResourceSet` as scope. Therefore, the API only offers a constructor, which gets a `ResourceSet` to ensure a clear scope definition.

Live. In order to use the `EContentAdapter`, one can also imaging a unregistration mechanism for the event manager, not only from a `ResourceSet`, but also from some subtrees in the composition tree. This could enable the application to focus only on model parts. In addition, the registration of a `EContentAdapter` to any number of notifier is very easy to implement and semantically reasonable but the unregistration comes into some semantically obscurities. For instance, the set of registered node, if one subscribes for a root node and a contained sub-node, and afterwards unregisters from the sub-node, is either the complete tree spanned by the root node or the node from the sub-node get excluded. For this case, the expected behavior is inconclusive.

To sum up, for the non-transactional environment the event manager registers an `EContentAdapter` at the passed `ResourceSet` and the application has to create a new event manager for changing the scope. This offers the opportunity to build an event manager registry, that allows mapping event managers to `ResourceSets`. This results in exactly one event manager for one `ResoureSet`, which is adequately because the event manager can handle any number of adapters and more than one event manager per resource will result in redundant calculations and more memory usage.

Transactional. To support the transactional subscription, the event manager needs to identify the appropriate transactional domain for the given `ResourceSet`. The EMF native and the CDO environment offer helper methods, which use registries to map from a `ResourceSet` to the `EditingDomain` or `CDOSession`. For instance, EMF Transactions offers the `TransactionUtil.getEditingDomain(ResourceSet)` helper method. In case the `ResourceSet` is registered neither on EMF nor CDO, the call of the `subscribeTransactional` method result in an `UnsupportedOperationException`. Except that the meta-model is annotated with a setting-delegate annotation, which enables the use of micro transactions. This annotation signals the EMF CodeGen environment to generate the delegation methods to the mirco transaction component.

3.3 Event Filtering

The third responsibility is to deliver only notifications to the registered listeners that actual are of interest to the listener. Therefore, three different approaches to an effective event filtering were developed during the bachelor's project. These approaches are the traversing of event filter trees, the single filter-table approach and the multi filter-table approach. All implementations have to handle the same vocabulary of event filters.

3.3.1 Event Filters

The event filters offered by the event manager are nearly equal to the event filters by EMF. There are logical filters which enable applications to logical combine filters with each other. The essential operations `or`, `and` and `not` are supported, which enable applications to define each boolean operation. Event filters define constraints to properties of notifications, which are unique for each filter type. All event filter implementations are shown in figure 3.5. The set of implemented event filters is mainly defined by the requirements of the impact analyzer. For this reason, additional filters are not implemented. For instance, a resource filter could be implemented, which calculates the resource of the notification notifier. Nevertheless, the implementation of additional filters is very simple. The only requirement to a filter is the subclassing of the abstract Event Filter class, which defines mainly the needed operation for both underlying implementations. On the one hand, the `matchesFor(Notification)` method, which returns true if the filter matches, and on the other hand, the `getFilterCriterion` method, which returns the characteristic element for the filter. On using the filter one usually uses a factory to instantiate the specific filter and afterwards one set the needed attribute for instance `wantedClass (ClassFilter)`. Although, it is possible as a short-hand to directly call the constructor, which expects all needed attributes. Except one using the `ContainmentFilter`, which does not define any attribute, because it per definition only matches if the incoming notification affects an containment feature including the containment relation between `Resources` and `EObjects`. The concrete filter structure is shown in figure 3.5

Class	**Description**
`EventFilter`	Abstract super class for all event filter. It defines the two abstract methods `matchesFor` and `getFilterCriterion`.
`StructuralFeature-Filter`	Superclass for `AttributeFilter` and `AssociationFilter`. It implement the matching methods using the `getFeature` of a notification to match it with the owned feature
`AttributeFilter`	Specializes the `StructuralFeatureFilter` that only allows `EAttributes` as owned attribute.
`AssociationFIlter`	Specializes the `StructuralFeatureFilter` that only allows `EReferences` as owned attribute.
`PackageFIlter`	Holds an `EPackage` which defines the `EPackage` of the notifier of an notification.
`EventTypeFilter`	Hold an `Integer` value representing an EMF event type defined in the notification interface.
`ContainmentFilter`	As contrast this filter is an singelton instance,it matches if the feature of an notification is an `EReference` and `isContainment` returns true.
`ClassFilter`	Specifies the wanted `EClass` for the notifier of a notification. `includeSubClasses` specifies whether also instances of subclasses should match.
`OldValueClassFilter` and `NewValueClassFilter`	Specifies the wanted class which shall be contained in the old/new value set of a notification. These filter matches if the old/new value contains one or more instance of the wanted `EClass`.
`LogicalOperation-Filter`	Superclass for all logical filters. It has a reference to a set of `EventFilter` that are the operands for the underlying logical operations.
`OrFilter`	Matches if at least one operands matches the current notification.
`AndFilter`	Matches if each operand matches the notification.
`NotFilter`	Negates the contained operand. Therefore matches if the operand not matches.

The two methods used for the event filtering mechanism are described in detail in the two following sections.

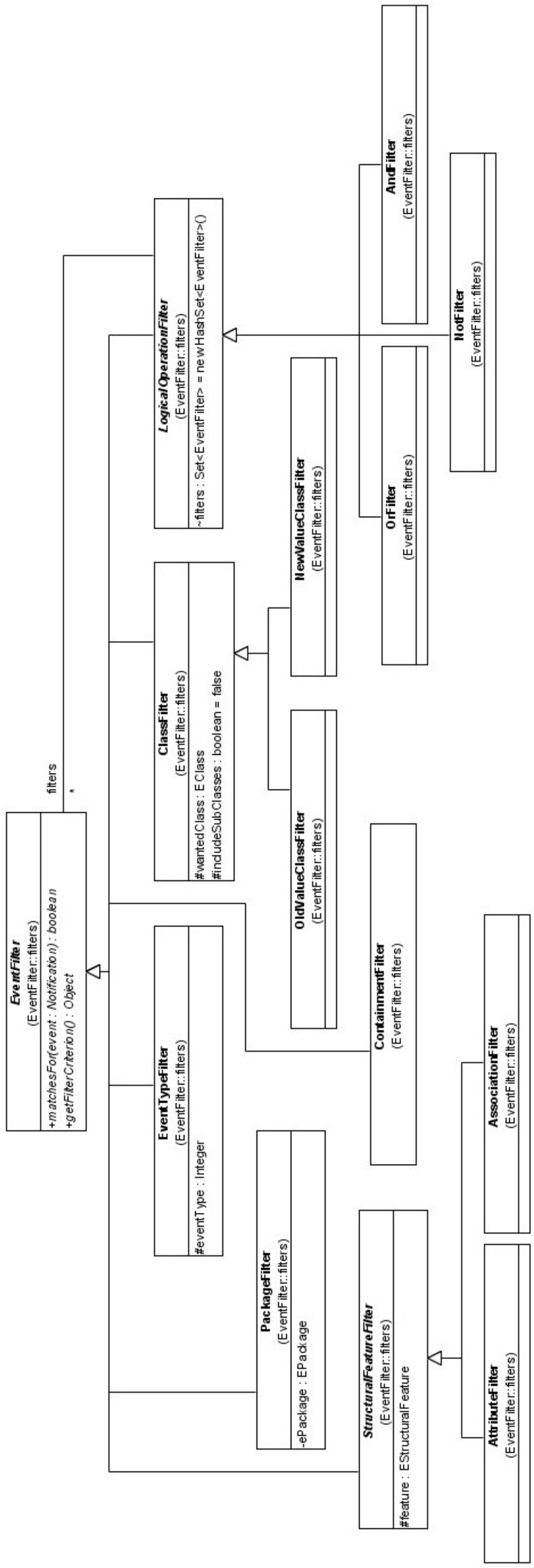

Figure 3.5: Class diagram of the event filter structure

3.3.2 Traversing Event Filter Trees

The traversing of the event filter trees is the most naive solution to the problem. Like it was implemented first in MOIN and is implemented in the EMF transaction component, every listener registers a composite event filter at the event management component. In case a notification is raised, the event manager iterates overall registered filter trees and calls the `matchesFor` method henceforth referred as match-method. Typically the root filter is a `LogicalOperationFilter`, which contains several other filters. For example, the root is an `AndFilter` which contains two `OrFilters`. Therefore, the match-method of the `AndFilter` calls on each contained element the match-method and calculates a logical `and` over the return values. Equally, OrFilters calculates and logical "or" over the return value of their contained elements. The leaves of the filter tree are all non-logical filters and use properties of the notification to match. The subclasses of the `StructuralFeatureFilter` hold a reference to an `EAttribute` or an `EReference` and their match-method return whether the feature of the notification matches the hold feature. As contrast, the `ClassFilter` holds `EClass` and a `Boolean`, which indicates whether subclasses should be considered. Therefore, the match method has to test whether the `Notifier` of the notification is an `EObject` and whether the `EClass` of the `EObject` or a super class matches the wanted class. Likewise, the `OldValueClassFilter` and the `NewValueClassFilter` matches the content of the oldValue or newValue of the notifications. As contrast, the `EventTypeFilter` simply holds an `Integer` identifying the event type of the notification to match. As soon as all leaves are visited, the root filter returns whether the whole filter tree matches. Afterwards the event manager make a look-up in the intern map from filters to adapters and passes the matching notification to the `notifyChanged(Notification)` method of the `Adapter`. To match each registered adapter the event manager iterates overall filters in the filter table and traverse each of this filters completely. Therefore to unsubscribe an `Adapter` the event manager just has to remove it from the values of the intern map. The main performance and memory issue of this implementation is that no consolidation of filters is implemented. The filter code execution is nearly as frequently as without the event filtering component with filter code located in the listening applications. Except that two `Adapters` register the same event filter, this filter code is executed only once.

Listing 3.2: Pseudocode for the event filtering by traversing filter trees

```
for each EventFilter filter in registeredFilters{
  if(filter.matchesFor(notification)){
    interestedListeners = getAllListenerRegisterdForTheFilter
```

```
    notifyEachListener(interestedListeners, notification)
  }
}
```

3.3.3 Single Filter-Table

The single filter-table approach is based on the idea that one can build from each filter tree an so called `NotificationIdentifier`. This identifier object holds all attributes of an notification and, additionally, several pre-calculated values. An event listener registers an event filter tree. Afterwards, the event manager builds an identifier, which identifies each matching notification. On handling a notification, the component also builds an identifier and does a fast lookup in an internal hash table, which holds the mapping from identifier to listener. Due to the ambiguous event filters, it is not possible to define an unique key at subscription time. For example, the `ContainmentFilter`[2] has to know at subscription time all possible containment references in which is able to contain an element. The actual class of the element is defined by a class filter. Therefore, one can use additional values in the `NotificationIdentifier`. Nevertheless, there filter types that cannot clearly map to pre-calculated values. Therefore, the incoming notification has to be mutated in such a way that all possibilities of value combinations get generated. This mutation will result in an immense calculation effort. This is the reason for investigating in a multi filter-table approach.

3.3.4 Multi Filter-Table

As contrast to the previous approach, this approach is mainly focused on the combination of filters and the effective matching. In case that applications register an `Adapter` and an `EventFilter`, the event manager registers the adapter as following. Firstly, it calculates the disjunctive normal form for the event filter. Afterwards, the root filter is an `OrFitler` and consists of any number of `AndFilters`, which hold normal and negated leave filters. The calculation effort for the disjunctive normal form highly raises with raising event filter tree depth. This is the reason for the relative high subscription times to the event manager. Nevertheless, after calculating the normal form the leave filters get sorted into the type-specific filter-tables hold by the event manager. Each filter table holds exactly one type of filters. During the sorting for each `AndFilter`, a Registration is constructed. A `Registration` points with a `WeakReference` to an `Adapter` and is unique for an `AndFilter`. Once a filter is added to a event filter table, there is an filter

[2] The EContainment filter matches each notification concerning a `EReference` which is containment.

entry created which has as key the result of the `getFilterCriterion`-method of the filter and as value, one or more registrations departed in "Yes" and "No" (negated registrations) as shown in figure 3.6. Afterwards, the registration process is complete.

Essentially for the event filtering is the `getAffectedObject` method from each event

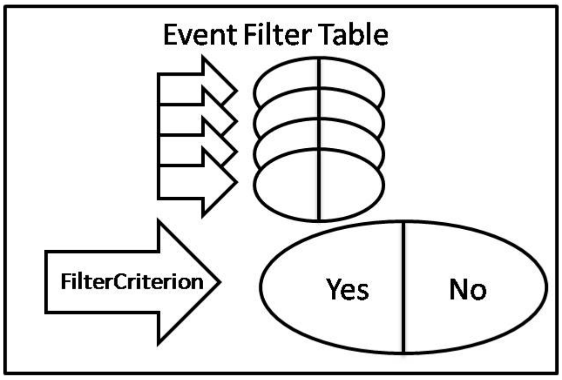

Figure 3.6: Visualization of an Event Filter Table

filter table, these methods extract from a notification the interesting part for the specific filter type. This interesting part serves as key in the event filter table and is returned by the `getFilterCriterion` method of the event filter. Due to the usage of `HashMaps`, the matching is very fast and reach constant calculation effort. The `ClassFilter` for instance returns an `EClass` as filter criterion and the corresponding `TableForClassFilter` returns the `EClass` of the notifier for the notification. Therefore, the event matching mechanism is as following. Firstly, the event manager iterates of all registered event filter tables and collect all "yes"-registration, which matches the event, and all "no"-registration, which explicit not match the event. These are negated registration for negated event filters. Afterwards the logical composition of all registrations is done, which is visualized in figure 3.7.

Listing 3.3: Pseudocode for the event filtering by using multiple filter tables

```
for each EventFilterTable table in allTables{
  yesSet.addall(getYesRegistrationsFor(notification))
  noSet.addAll(getNoRegstrationsFor(notification))
}
matchingSet.addAll(yesSet)
matchingSet.removeAll(noSet)
matchingSet.removeAll(unaffectedRegistrations)
```

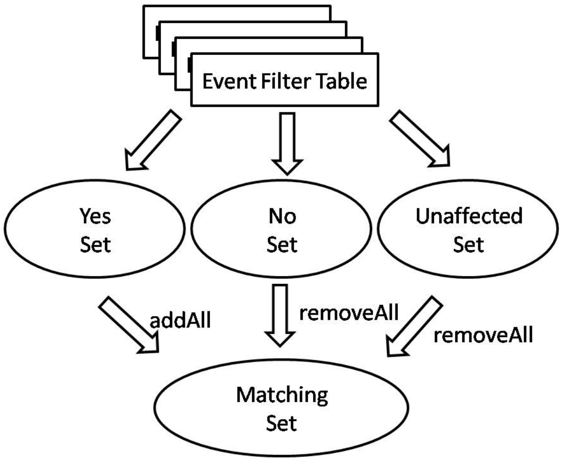

Figure 3.7: Logical combination of Registration sets

```
for each Registration reg in matchingSet{
  interestedListeners.add(reg.listeners);
}
interestedListener.makeUnique()
notifyEachListener(interestedListeners, notification)
```

Hereby, all registrations from the `YesSet` get added to the resulting `MatchingSet`. Thereafter, all *Yes*-registrations from unaffected tables and all explicite *No*-registrations get removed. Due to the fact that identical registrations only appear if matching filters are conjunct, only matching filters are left in the `MatchingSet`. Finally, all adapters linked to the matching registrations get notified in sequence. Due to this, the time to notification of an adapter depends also on the calculation time of all adapters previously matching the event.

The current set of event filter tables is shown in figure 3.8. As visualized for each event filter type, which can be instantiated, an event filter table is created. The functionality of the `getAffectedObject`-methods is described below.

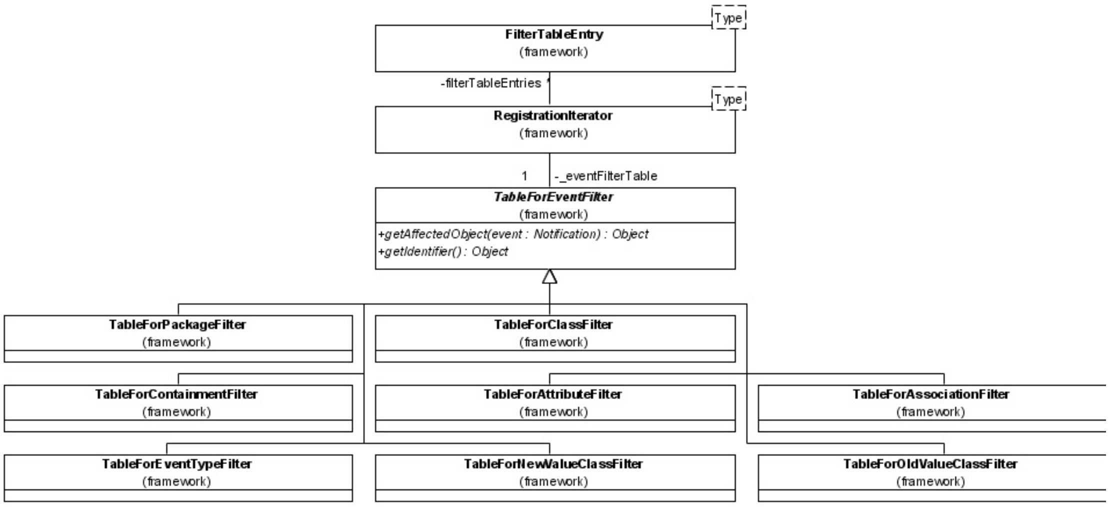

Figure 3.8: Class diagram of the filter table structure for the filter table-based implementation

Class	**getAffectedObject**
`TableForClassFilter`	Gives the `EClass` of the event source
`TableForPackageFilter`	Returns the `EPackage` for the `ECLass` of the event source
`TableForContainmentFilter`	Checks whether the event affects a containment feature of a `Resource` contents relation
`TableForAttrbiuteFilter`	Gives the `EAttribute` if the notification affects one
`TableForAssoziationFilter`	Extracts the `EReference` if the notification affects one
`TableForEventTypeFilter`	Return the event type as Integer
`TableForNewValueFilter`	Returns the collection of all classes for all `EObject` contained in the new value set of the notification
`TableForOldValueFilter`	Returns the collection of all classes for all `EObject` contained in the old value set of the notification

4 Implementation

In the following chapter implementation details for the two different event manager components are discussed.

4.1 Component for Traversing Event Filter Trees

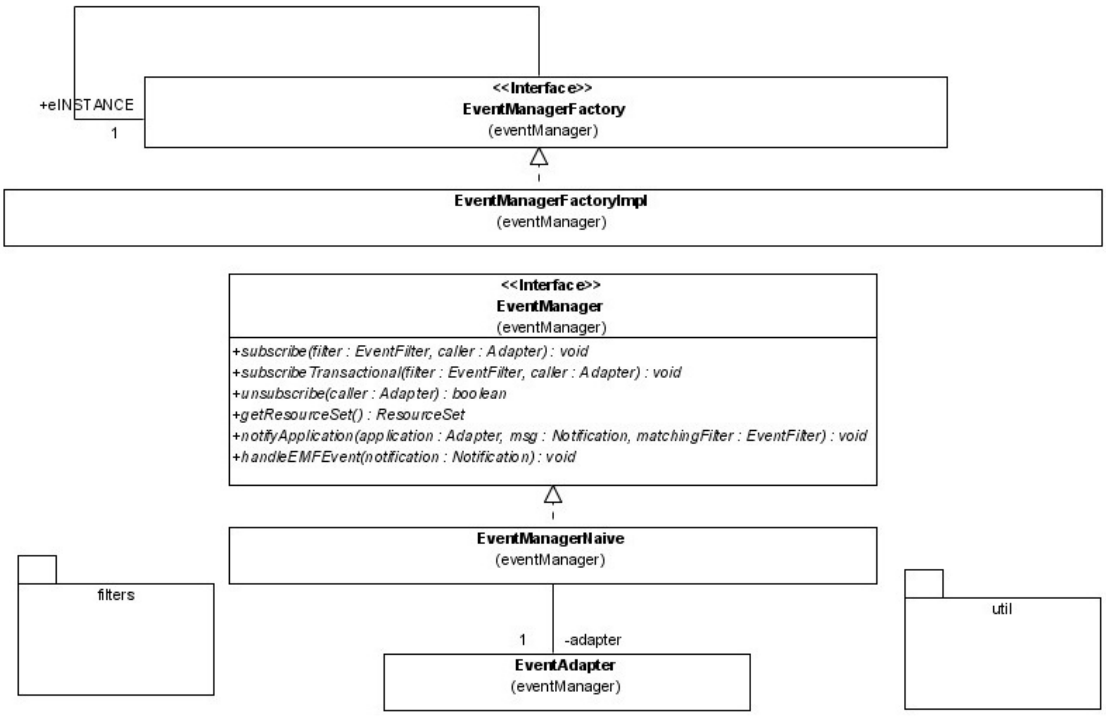

Figure 4.1: Class diagram of the event manager structure for the naive implementation

As visualized in figure 4.1, the naive event manager component consists eventually one unique class, the `EventManagerNaive` class. The other components are identically for both implementations. The `EventManagerFactory` is used to instantiate an event manager. As described in section 3.2 both components implement the same interface. Internally, the interface has some methods, which are actually not exported to foreign packages. This methods are `handleEMFEvent` (used to fire an event to the intern filtering process) and the `notifyApplication` method, which directly delegates to the `notifyChanged` method of the EMF adapter interface. In addition, there is a specialization of the `EContentAdapter`, which holds a `WeakReference` to the event manager and delegate each received event to the `handleEMFEvent` method. Furthermore, there are two packages `filter` and `util`. The `filter` package contains the complete filter structure as shown in figure 3.5. The `util` package contains of two helper classes mainly used by the event manager. These classes are shown in figure 4.2. The `NotificationHelper` class offers helper methods to recognize properties of notifications in a MOIN like way. Therefore, for instance the `isElementLifeCycleEvent` method returns whether the incoming methods affects a containment feature or a add/remove on the contents of a `Resource`. Additionally, the `EventFilterFactory` offers shortcomings to `createFilterForElementInsertion`, which internally creates an `OrFilter` over a `ContainmentFitler` and the `NewValueClassFilter`. Similarly, the other methods help the impact analyzer to abstract from the underlying platform.

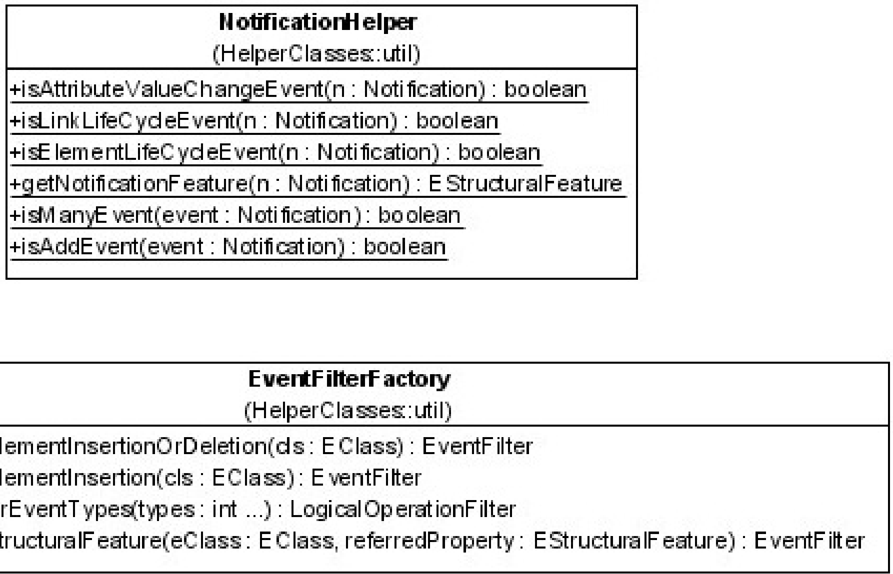

Figure 4.2: Class diagram of the event manager util package

4.2 Component for the Usage of Multiple Filter-Tables

As discussed before, the `filter` and `util` package are used for both components. As shown in figure 4.3. The structure of the table-based event manager is clearly more complex than the structure of the naive approach. In this section a short description to the elementary classes is given.

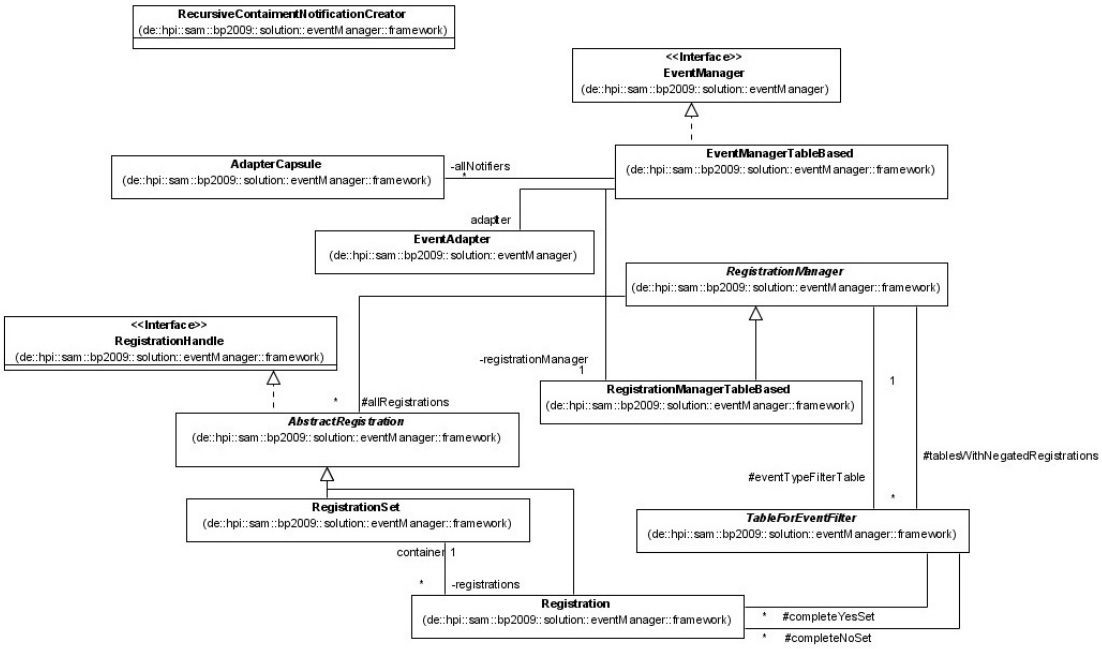

Figure 4.3: Class diagram of the event manager structure for the filter table-based implementation

Class	**Description**
`EventManagerTableBased`	Implements all methods required by the `EventManager` interface. This class conducts all other components to realize all methods.
`Registration`	An registration is the unique pair of an set of `EventFilters` that combined by a logical *and* and a EMF `Adapter`
`TableForEventFilter`	The superclass of all event filter tables shown in figure3.8
`RegistrationManager`	This class holds the filter matching logic and informs the `EventManagerTableBased` for each found match
`RecursiveContainment-NotificationCreator`	Due to the fact that moving a subtree in a containment hierarchy only raises the notification that the root element was moved. This class generates for the complete subtree additional `ADD`/`REMOVE` notifications.

5 Evaluation

In the following chapter both implemented components get evaluated against each other.

5.1 Test Arrangement

All tests were executed on an 1,6 GHz Intel Core Duo machine with 4 GB main memory and in a 32-bit Java VM. The tests run on the ngpm[1] meta model. There are 203 constraints parsed out of all contained packages using the *OCLToAst* component [4]. This component is used to traverse `EPackage` and extract OCL expressions from the OCL annotations [2]. All constraint get transformed to event filter using the impact analyzer component [3]. The constant filter depth of all filter is 4. Thus, none of the filters has more than three level of logical operation filters. The depth of a filter is defined by the maximum number of filters from the root logical operation filter to a non-logical filter type. In average there are 246 leave filter (non-logical filters) per filter tree. The maximum number of leave filters is 602 and the minimum number of filters is 5. Consequently, the traversal of the simple trees is expected to be not very time consuming due to the simple structure of the filters.

Listing 5.1: Calculation for the maximum computation effort for the naive implementation

```
maxNumberOfComparisons = maxNumberOfLeaveFilter * numberOfFilters
maxComputationEffort = maxNumberOfComparisons
```

The table based approach has an equally estimated runtime behavior as the native approach.

Listing 5.2: Calculation for the maximum computation effort for the table-based implementation

```
maxComputationEffort = lookupEffort*numberOfEventFilterTables
  + combinationEffort
```

[1]The ngpm meta model is one of the largest proprietary meta models of the SAP.
[2]http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html

```
lookupEffort = 1
```

The reason for this is the actual combination effort. This include the removing of all non-matching registrations from the yesSet and in addtion all unaffected registrations.

Listing 5.3: Calculation for the combination effort for the different registration sets

```
combinationEffort = min(min(yesSet-Size, noSet-Size)
  unaffectedSize)
```

Due to the fact that all filters are deeper than 3, all filters have to converted into the disjunctive normal form. Therefore the size of the `yesSet` and `noSet` is estimated to be equally. Due to the negation of the event filters during the conversion. The effort of a `removeAll` for a `HashSet` is by ideally 1 access cost, as consequence the overall operation costs are the minimum of the size of set1 and the size of set2.

5.2 Execution

Figure 5.1: Visualization of the Filtering effort for all ngpm constraints

The first measurement in figure 5.1 is executed on a corpus of 200 event filters. As one can see the naive component is in average 10 percent slower than the table-based approach. Nevertheless, the small difference was likely unexpected. Due to the fact that the naive component should iterate each time over all registered filters. As contrast, the inner-structure of the naive solution reveals that all filters are hold in `HashSets`. This enables the naive approach to combine duplicated filters. As shown in figure 5.2, the

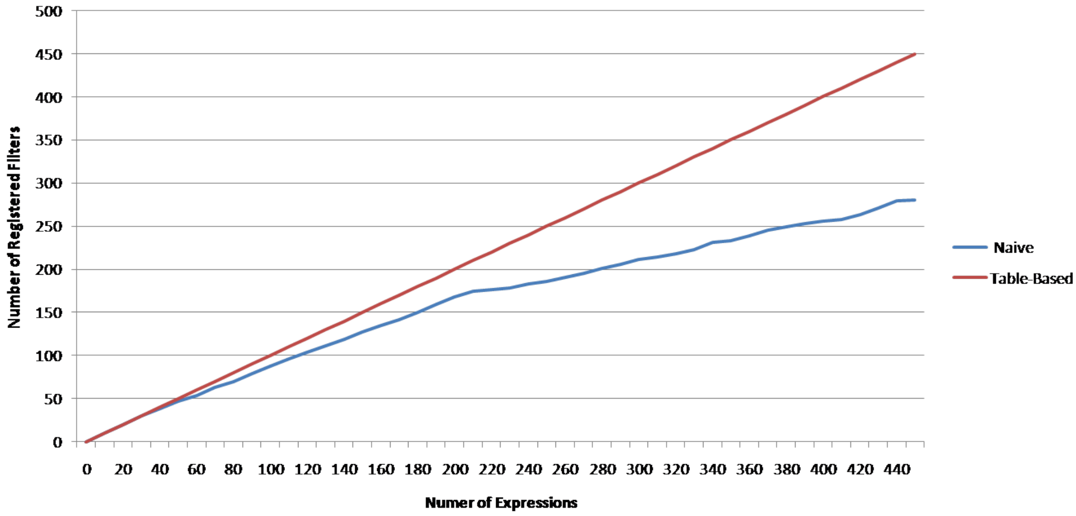

Figure 5.2: Visualization of the filter reduction of the naive implementation

generated filters of the impact analysis are not unique among the diverse expressions of the ngpm model. Therefore, the naive implementation has to match less filters during the event handling. This effect is more visible in figure 5.3. Since the automated generation of additional filters only adds an additional `OrFilter` as root of the filter. Hence, the generation creates an equally set of filters. Thus, the registered filter set of the naive solution is constant. As contrast, the calculation time of the table-based approach increases linear.

To generate different filters, one manipulates the filter structure, so that no `hash` or `equals` methods get specialized by any logical filter. As consequence, the measurement in figure 5.4 shows that on simulating all filters as clearly different, the naive approach has an nearly square increasing rate. As contrast, the table-based approach shows an nearly constant behavior. Due to the diversity of filters the naive implementation is not able to combine filters and the table-based approach profits from the small `YesSets`. This shows especially that the naive component is more appropriate for simple scenario among similar set of filters. As contrast the table-based approach clearly more scales withe an increasing number of various filters.

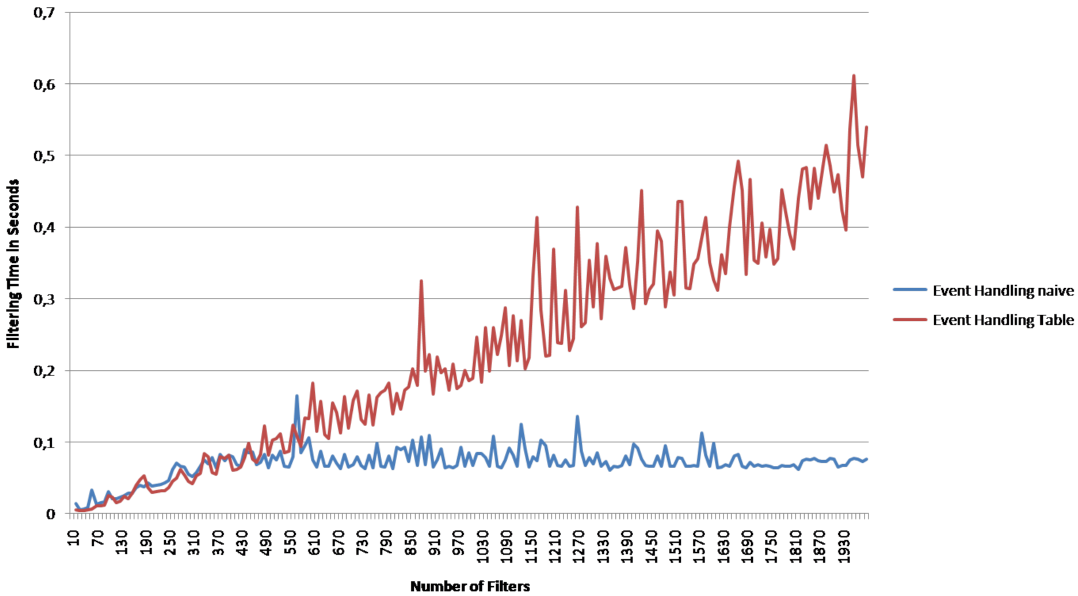

Figure 5.3: Visualization of the Filtering effort for all ngpm constraints and generated copies with methods

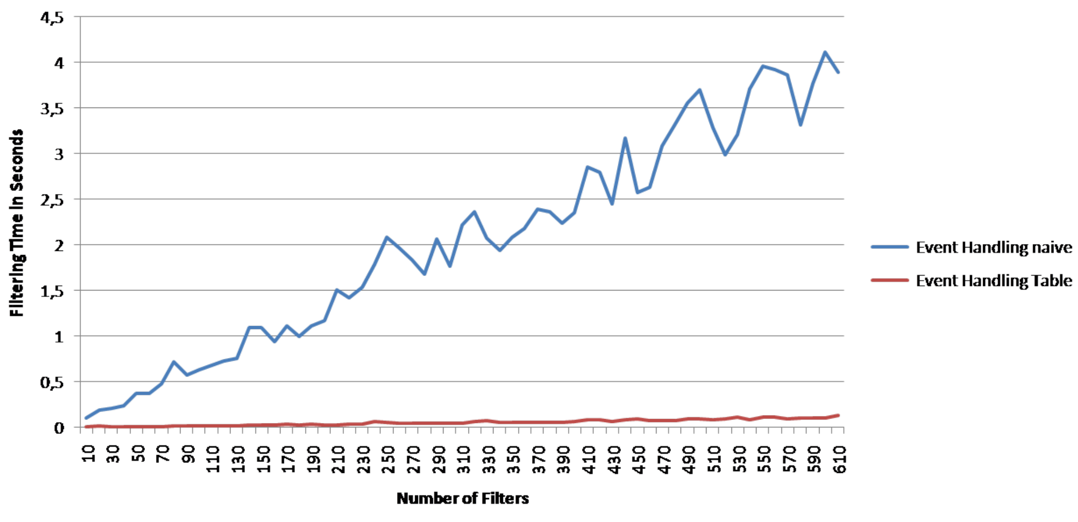

Figure 5.4: Visualization of the Filtering effort for all ngpm constraints and generated copies without hash methods

6 Conclusion

The increasing complexity and importance of software models in the industrial environment is obvious. Due to the high frequency of changes to models and the high number of constraints, an incremental consistency check for models is required.

In this paper, I motivated the implementation of such a component for EMF and defined the preconditions to implement such a component. Consequently, the concepts of event managing for model infrastructures have been described, which are essential to an incremental consistency check. The requirements for an event manager have been identified, which are mainly the expressiveness of the provided event and filter structure. We also identified and solve issues in the context of transferring of tools from MOIN to EMF.

Furthermore, implementations were discussed, which manifest two different approaches concerning the effectiveness of such an event manager component by tweaking the event filtering mechanisms. Nevertheless, both implementation are suitable for the *Scalable EMF Impact Analyzer Framework* and can serve the impact analyzer component as an event manager. In addition, I introduced concepts towards the transactional mechanism for the event manager, which offer a basis for upcoming implementations.

Finally, both implementations have been compared. As discussed in the evaluation section, the naive approach is satisfying simple scenarios with a small number of different filters. In contrast, the table-based approach is more suitable for a large number of filters, which are very different. To sum up, one can identify the multi filter-table event manager as one vital solution for the usage in industrial environments.

List of Figures

Bibliography

[1] Altenhofen, Hettel, and Kusterer. OCL support in an industrial environment, 2007.

[2] Martin Hanysz. Instance-Based Context Calculation of OCL Expressions, 2010.

[3] Tobias Hoppe. Synthesis of Event Filter determining the reevaluation of affected OCL expressions, 2010.

[4] Thea Schröter. Effiziente Navigation bei der OCL-Auswertung (Efficient Navigation for OCL Evaluation), 2010.

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und dafär keine anderen als die genannten Quellen und Hilfsmittel verwendet habe.

Potsdam, June 25, 2010 ....