An Integration Framework for Open Tool Environments

G. Paul, K. Sattler, M. Endig
Department of Computer Science, University of Magdeburg
D–39016 Magdeburg, Germany,
EMail: {paul|kus|endig}@iti.cs.uni-magdeburg.de

Abstract

Tool environments supporting the development of complex products need to be open and flexible. These requirements cannot be fulfilled in an adequate way by predefined coordination structures and interfaces. Based on the notion of component–oriented software development this paper presents a framework for open tool environments. Besides the runtime environment this framework provides language support for describing tool interconnections in an implementation–independent manner.

Keywords

Tool integration, software composition, configuration language

1 INTRODUCTION

The development of complex technical products requires the use of computer based tools. These tools provide services for the user to accomplish his tasks. For efficient use these tools must be integrated. Tool environments provide the required mechanisms for the integration. In order to describe the different dimensions of the integration, one usually identifies three aspects: data, control and presentation integration (Schefström 1993). Recently it has become clear that openness and flexibility are important characteristics of a tool environment. An open and flexible environment should provide the possibility to compose tools as building blocks in a new manner needed by special requirements. Prerequisites for this aim are first plugable tools and second suitable composition mechanisms. Furthermore, for a simple adaptation the user should be able to describe any system configuration at an abstract implementation–independent level and instantiate it at runtime. Based on these requirements this paper presents an approach for tool integration based on the description and runtime mapping of tool interconnections.

2 RELATED WORK

The explicit representation of the interaction and coordination aspect of a software system is subject of several recent contributions. Approaches like *formal connectors* (Allen & Garlan 1994) or *synchronizers* (Frolund & Agha 1993) introduce special abstractions for decoupling the interaction part from the component implementation. *Darwin* (Magee et al. 1995) is a language for constructing distributed programs from hierarchically structured specifications of a set of component instances and their interconnections. Components are viewed in terms of both the services they provide to components and the services they require to interact with others. The semantic of bindings between components is predefined as the association of a service required by one component with a service provided by another one in form of a method call. *Olan* (Bellissard et al. 1995) extends the *Darwin* approach with the notion of connectors. Connectors mediate the interaction between components. There are several predefined connector types, but in the current version the configuration language doesn't support user–defined connectors. The *ToolBus* architecture (Bergstra & Klint 1996) is an approach for tool integration based on the separation of coordination and computation. The interactions between tools are described by so called T–scripts containing sequences of communication and control primitives. Existing tools have to be encapsulated by adapters, which are responsible for data transformation and communication protocol adjustment. The problem of interoperability between components with incompatible interfaces is discussed in (Konstantas 1993). This approach is based on an *Interface Adaption Language (IAL)* for describing the mapping between data types and object interfaces.

3 COMPONENT MODEL

Within the scope of a tool environment the tools are the building blocks or components. The granularity of these tools ranges from simple objects with a few services up to full featured applications. The component model enables the representation of the environment configuration in the form of interacting tool components. We distinguish between two aspects of a configuration: the *computation part* for describing the tool services and their requirements to the environment and the *coordination part* for organizing the behaviour of a group of components by managing interdependencies between their activities.

A *component* is a software entity with well–defined interfaces. An interface is an abstraction of the component behaviour and encapsulates the internal representation of state and implementation. The interfaces and the implementation of a component are defined by its *component class*. In principle there are two kinds of component interfaces:

- *Operational interfaces* define a set of operations which can be performed by a component or which a component can invoke at another one.
- *Event interfaces* define a set of events which can be announced by a component.

An interface can be derived from other interfaces and extends these with new operations or events. However, the derivation is only allowed within the scope of the same kind of interfaces.

A component has to implement at least an operational interface. In addition, it can define a set of event channels and slots. These elements are abstract interaction points: an *event channel* represents the point where the component announces events; a *slot* is the point where the component invokes services from another component. Channels and slots are defined by the provided event interface resp. the required operational interface. They enable the explicit binding between components.

The coordination part of a configuration is represented by the concept of interactors. An *interactor* is an entity encapsulating the collaboration of a group of objects in form of interaction relations. Interactors support the definition of interactions in an implementation–independent way and help to reduce the dependencies between components. In addition, they provide a way for describing synchronization aspects, different interaction styles or interface adaptation. The properties of interactors are defined by *interactor templates* consisting of a set of roles, a set of attributes and a set of links between the interaction points of the components. A *role* is a placeholder for a member of the relationship. *Links* define the behavioural dependencies between roles by describing the message flow. There are the following types of links:

- *Simple links* connect the slot of a component directly with the service provider. All operation requests are directly delivered to the provider.
- *Event links* are based on the mediator approach (Sullivan 1994, Paul & Sattler 1996). There can exist operation invocations or sequences of invocations associated with events of an interactor role. The invocations are performed when the corresponding event ist announced.
- *Operational links* are the most flexible kind. Here the interactor defines how the interface of a requesting role's interaction part is implemented by services of other role objects. An interactor can forward requests, map one request to another one or process parameters via operational links.

With these concepts a configuration of the environment is defined by a set of components interconnected by interactors. The configuration represents the section necessary for processing a task from the set of tools of the environments.

4 INTEGRATION FRAMEWORK

TIFRAME is an object–oriented framework based on the introduced component model and intended for the control integration layer in tool environments. The framework consists of two parts: the configuration language TIL for describing components and their interconnections and the runtime environment for instantiating configurations described in TIL.

4.1 Language Support

In order to adjust the tool environment to specific tasks it should be possible for the user to describe "his" configuration on an abstract, implementation–independent level. Graphical or textual configuration languages like *Darwin* or *Olan* are suitable mechanisms for this aim. For the use with the TIFRAME framework we have developed a similar language called *Tool Interconnection Language (TIL)*.

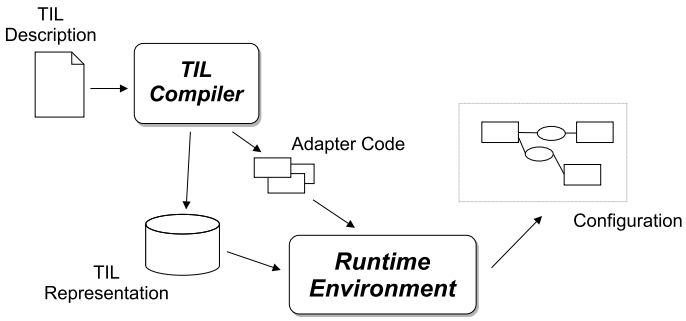

Figure 1 Integration process using TIL

From the TIL specification the compiler generates an internal representation of a configuration readable by the runtime environment and the adapter code required for interconnecting the components (Figure 1). The compiler checks the specified interaction relations for interface compatibility. Based on the configuration description the runtime environment is able to instantiate and connect the components directly or via the generated adapters.

The configuration language provides only elements necessary for specifying component interfaces and interconnections because the components are usually implemented in existing programming languages. According to our component model we distinguish between interfaces as abstract types and component classes as type implementations. In order to simplify the integration of existing tools the description of operational interfaces in TIL is derived from the *Interface Definition Language (IDL)* of the OMG (OMG 1995). In this way it is possible to import existing IDL specifications in TIL without changes and to treat CORBA objects as components in TIFRAME. Event interfaces are specified in a modified IDL notation considering the characteristics of event notifications (e.g., no output or return parameters):

```
events AssemblyEvents {
    part_added (Assembly a);
};
```

The definition of a component class requires at least the supported operational interface (`supports`) and the reference to the implementation. The optional event interfaces of the component are specified in the `announces` clause by giving an iden-

tifier and an operation (`using`) for accessing the interaction point. Slots are specified in a similar way in the `requires` clause:

```
component AssemblyEditor {
    supports AssemblyEditing;
    requires AssemblyView view using attach_view;
    primitive implementation { bridge = "IIOP"; };
};
```

Besides component classes implemented by native code, components can also be defined as composition of other components.

An interactor template is defined by a set of roles representing the participants of the interaction relation and a behavioural part. The roles are characterized by the required type (in terms of the interface or the component class) and an identifier. An interactor template provides a common definition for a set of components related through their interfaces. Similar to parameterized types in languages like C++ or Eiffel a single interactor template might be used to instantiate individual interactors for connecting different components. The roles of the template specify the requirements to the parameters (the components):

```
interactor observation<AssemblyEditor editor, StructureBrowser browser> {
    editor.notifier → {
        part_added (asm) { browser.update_view_for (asm); }
    };
};
```

The behavioural part of an interactor template specifies a set of links between the role objects in the notation `initiator → responder`. The initiator part of a link consists of interaction points initiating the communication. The responder part of a link respresents interaction points which implement the interface required by the initiator. In the most simple case this might be the operational interface of a component represented by a role object. For complex interactions like 1–to–n relations or interface adaptations it is possible to specify the responder part operationally. In this case actions are associated with the operations or events of the initiator interface. An action is a sequence of statements for manipulating arguments, accessing attributes and invoking operations of role objects. Based on these actions the TIL compiler generates the appropriated code for connecting the component instances.

A configuration specified in TIL describes the instantiation of components and their interconnections via interactors. Instances are created by using the *new* operator and can be assigned to variables. The instantiation of an interactor requires component instances or their interaction points as parameters:

```
configuration SimpleConf {
    AssemblyEditor editor = new AssemblyEditor;
    StructureBrowser browser = new StructureBrowser;
    new observation<editor, browser>;
};
```

In this sense a TIL configuration specifies the initial structure of the tool environment. During the computation further components might be instantiated as a part of coordination activities written in the actions of interactors.

4.2 Runtime Environment

The TIFRAME runtime environment provides mechanisms for the mapping from a component–based configuration to a concrete set of interconnected tools. In detail the following functions are implemented: the instantiation of configurations according to a given TIL specification, the mapping from components and their connections to implementations and the access to components in order to support framework applications.

The runtime environment of the framework consists of a collection of classes organized in several layers. The *configuration layer* contains services for instantiating configurations and for accessing the various components. The completion of this layer requires the services from the *component layer*. This layer represents the components and interactors independent from their implementation and their communication interfaces by proxy objects. The *bridging layer* provides services for the communication between proxies and real components as well as the mapping of the component properties to the implementation. This layer defines an abstract interface for the communication bus and enables the use of concrete communication solutions by using platform–specific bridges, e.g. CORBA or COM/DCOM.

In order to support a flexible integration of different component implementation and communication platforms we use a reflective approach (Maes 1988) for mapping the component model. In addition to the *base level* components and interactors there is a *meta level* containing information about the objects from the base level. This information specifies the structural and behavioural aspects required for relating the base level objects to their implementation. It is encapsulated by *metaobjects*. Usually metaobjects are initialized by a model specified in TIL, but it is possible to manipulate them at runtime with framework services. The base level objects are directly connected with their metaobjects. Every action (like instantiation, operation invocation or connection) on a base level object is controlled by the related metaobject. Figure 2 illustrates the message flow resulting from the interaction between base and meta level.

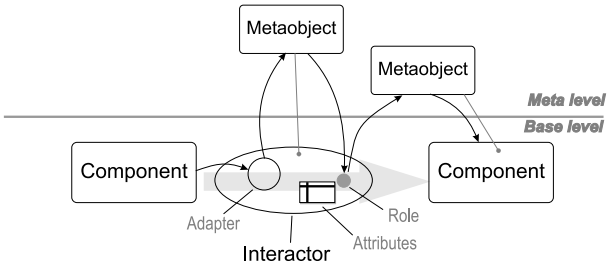

Figure 2 Message flow in an interactor

The use of interactors and metaobjects leads to an increased communication overhead and to performance loss. So if two components do not require any interface

adaptation they can be directly connected without intervention of framework services. The framework is only responsible for instantiating and interconnecting these components.

4.3 Mapping to CORBA

In principle our framework approach is independent from specific communication systems. However, for the implementation of a complete tool environment the integration of a concrete interoperability platform like CORBA is required. In this context, the mapping of the concepts from the component model to the CORBA object model has to be considered.

The mapping from TIFRAME components to CORBA objects is straightforward, if a component only supports an operational interface. In this case the IDL interface is identical to the operational interface. Event interfaces are implemented by using the *Event Service* (OMG 1994). The drawback of this service is that no mechanisms for specifying the structure of event notifications are available. In order to identify the various events, the event type (in fact the type identifier) has to be included in the event message. The slots of a component can be represented in different ways: by operations, attributes or via the *Property Service* (OMG 1994). The concrete implementation is described by the specification of the component class.

The mapping of the interactor concept requires the treatment of interactor instances as CORBA objects. An interactor is implemented by a group of objects: one object supporting the required interface for each initiator role and one object for managing the interactor state, e.g. attributes and references to role objects.

5 CONCLUSION

In this paper we have presented an approach of a framework for component–oriented tool integration. Our approach is based on a model which enables the explicit coordination of tool activities by using interactors as semantic relations. Special characteristics of the framework are the integration of different communication platforms and the absence of predefined interfaces and protocols. Thereby the use of various implementations and the integration of existing tools is simplified.

The *Tool Interconnection Language (TIL)* provides a way for describing the structural and behavioural properties of these relations as well as their use for the configuration of the tool environment. In contrast to similiar approaches TIL supports user–defined interactors for the integration of independently developed tools and the adaptation of incompatible interfaces.

An initial release of the framework is implemented in Java. Currently this framework is used in the PACO environment - a project of the University of Magdeburg for developing an integrated engineering environment. Further development on the framework includes the support of specification activities by interactive tools.

REFERENCES

Allen, R. & Garlan, D. (1994). Formal connectors, *Technical Report CMU–CS–94–115*, Carnegie Mellon University.

Bellissard, L. et al. (1995). Component–based programming and application management with olan, *Proc. of Workshop on Object–Based Parallel and Distributed Computation*, Springer Verlag, Tokyo.

Bergstra, J. & Klint, P. (1996). The toolbus coordination architecture, *in* P. Ciancarini & C. Hankin (eds), *Coordination Languages and Models*, Springer Verlag.

Frolund, S. & Agha, G. (1993). A language framework for multi–object coordination, *in* O. M. Nierstrasz (ed.), *ECOOP'93*, Springer Verlag.

Konstantas, D. (1993). Object oriented interoperability, *in* O. Nierstrasz (ed.), *ECOOP'93*, Springer Verlag.

Maes, P. (1988). *Meta–Level Architectures and Reflection*, Elsevier Science Publishers B.V.

Magee, J., Dulay, N., Eisenbach, S. & Kramer, J. (1995). Specifying distributed software architectures, *Proc. of the 5th European Software Engineering Conference*, Barcelona.

OMG (1994). *Common Object Services Specification, Volume 1*.

OMG (1995). *The Common Object Request Broker: Architecture and Specification, Revision 2.0*.

Paul, G. & Sattler, K. (1996). MediatorService – integrating distributed objects by describing their interactions, *in* O. Spaniol et al. (eds), *Trends in Distributed Systems '96*, Aachen. In german.

Schefström, D. (1993). *Tool Integration – Environments and Frameworks*, John Wiley & Sons.

Sullivan, K. (1994). Mediators: Easing the design and evolution of integrated systems, *Technical Report 94-08-01*, Departement of Computer Science and Engineering, University of Washington.