A Framework for Component–Oriented Tool Integration

Kai–Uwe Sattler
Department of Computer Science, University of Magdeburg
D–39016 Magdeburg, Germany

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. This paper presents a framework for control integration in open tool environments. The approach is based on a component model which supports the description of tool interconnections in an abstract implementation–independent manner.

1 Introduction

The development of complex technical products requires the use of computer based tools. These tools provide services for users to accomplish their tasks. They support the modelling of the shape of a product, the adjustment of parameters or the simulation of load. 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 [1].

Data integration deals with the management of design data and documents in a consistent way. Moreover, this aspect includes services for all the tools to store and retrieve their data. *Control integration* enables the direct interaction between tools to share their services and to coordinate activities without the intervention of the user. The subject of *presentation integration* is to support a homogeneous user interface to all services of the system. Other classifications identify additional aspects: *platform integration*, which represents services for network and operating system transparency and *process integration*, which refers to the support of the development process [2].

In the last few years, several tool environments and reference models have been developed. Mostly these approaches base on a central repository for data management with a standardized access interface for the tools. A global manager allows the user to control the activities of tools, e.g. to start and stop a tool in order to process data.

Recently it has become clear that openness and flexibility are important characteristics of a tool environment. Openness indicates the ability to integrate newly developed or third–party tools; flexibility means the adaptability of the overall system

structure to changing circumstances. An open and flexible tool environment should provide facilities 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.

This paper presents an approach for control integration in tool environments based on the description and runtime mapping of tool interconnections. The remainder of the paper is structured as follows. In section 2 we propose a model of basic abstractions for tool components and their connections. In section 3 we present a framework as software infrastructure of a component based tool environment. Related work is discussed in section 4. Finally, section 5 concludes the paper and points to further work.

2 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 accomplishes the representation of the environment configuration in the form of interacting tool components. We separate two aspects of a configuration:

- The *computation part* for describing the tool services and their requirements to the environment.
- The *coordination part* for organizing the behaviour of a group of components by managing interdependencies between their activities.

This distinction is supported by the basic abstractions of the model: components and interactors.

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. A component can be implemented in several ways: by a single object, a group of collaborating objects, a function library or an encapsulated execution flow. 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 extend 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 an identifier plus the provided event interface or the required operational interface. They enable the explicit binding between components.

To illustrate these concepts, a simple scenario shall be considered (Figure 1). A modelling tool for the design of assemblies requires services in order to select parts which should be assembled. These services could be provided by a simple file selection tool or by a more advanced tool based on an engineering database. The interface to the selection tool is represented by a slot. Furthermore, the assembly tool should notify other tools about changes in the assembly structure. For this purpose the assembly tool offers an event channel. In this way a product structure browser is able to update its view. In addition, the assembly tool provides services for other tools, like creating new assemblies.

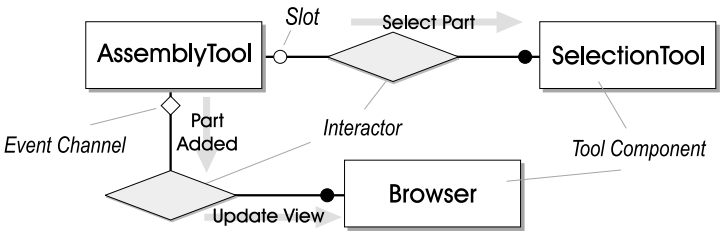

Figure 1: Example Scenario

Tool components can be connected in different ways for various tasks. These interconnections between the components form the coordination part of a configuration. They are represented by the concept of interactors. An *interactor* is an object 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 and specifies the requirements to this represented component. *Links* define the behavioural dependencies between roles by describing the message flow. There are the following kinds of links:

- *Simple links* connect the slot of a component with the service provider. All operation requests are directly delivered to the provider.
- *Event links* are based on the mediator approach [3]. There can be 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.

The benefits from using interactors are the decoupling of interacting components, the binding of components with incompatible interfaces (concerning interface types and interaction styles) and the representation of n–ary relations. In our scenario interactors could be responsible for binding the assembly tool with the selection tool (via simple or operational link) and for the propagation of updates to the structure browser (via event link).

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.

3 Integration Framework

TiFrame is an object–oriented framework intended for the control integration layer in tool environments. Based on the model from section 2 the framework supports the composition and coordination of heterogenous tool components. 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. Special characteristics of TiFrame 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.

3.1 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 are suitable mechanisms for this aim. For the use with the TiFrame framework we have developed a language called *Tool Interconnection Language (TIL)*.

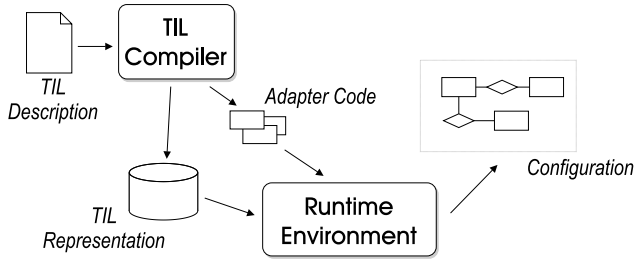

Figure 2: Integration process using TIL

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 2). 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 [4].

```
interface AssemblyServices {
    void edit_assembly (in Assembly a);
    Assembly get_active_assembly ();
};
```

In this way it is possible to import existing IDL specifications in TIL without changes and to treat CORBA objects as components in TiFrame. This opens the framework for future component development, because CORBA has gained widespread acceptance and commercial support. Event interfaces are specified in a modified notation considering the characteristics of event notifications (e.g., no output or return parameters):

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

The definition of a component class at a minimum requires the supported operational interface (`supports`) and the reference to the implementation. The implementation part defines the framework adapter (`bridge`) for accessing the native component implementation and the framework for creating a component instance. The optional event interfaces of the component are specified in the `announces` clause by giving an operation (`using`) for accessing the interaction point (the event channel). Slots – interaction points where services of other components are required – are specified in a similar way in the `requires` clause:

```
component AssemblyTool {
    supports AssemblyServices;
    announces AssemblyEvents notifier
            using update_listener;
    requires PartSelection selector
            using attach_selector;
    primitive implementation {
        bridge = "IIOP";
        class = "tiframe.cad.AssemblyTool";
    };
};
```

Apart from component classes implemented by native code, components can also be defined as a composition of other components. In this case the implementation part of the component class describes the subcomponents, their interactions (represented by interactors) and the associations between the interaction points of the composite component and the subcomponents (the `using` part in the `supports` clause).

```
component ModellingTool {
    supports AssemblyServices using editor;
    composite implementation {
        AssemblyTool editor = new AssemblyTool;
        SelectionTool selector = new SelectionTool;
        new simple_link<editor, selector>;
    };
};
```

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. In addition, a set of attributes can be defined. 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 simple_link<AssemblyTool editor,
           PartSelector pselector> {
    editor.selector → pselector;
};

interactor observation<AssemblyTool 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 represented by a role object. 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 (see interactor `simple_link`). 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 and events of the initiator interface. The actions will be performed with the corresponding invocation request or notification (see interactor `observation`). 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 adapter 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 {
    AssemblyTool editor = new AssemblyTool;
    StructureBrowser browser = new StructureBrowser;
    SelectionTool selector = new SelectionTool;

    new simple_link<editor, selector>;
    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 – either by component instances already defined or as a part of coordination activities specified in the actions of interactors.

3.2 Runtime Environment

The TiFrame runtime environment provides mechanisms for the mapping from a component–based configuration to a particular 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 tool implementations or framework objects, and the access to component services in order to support framework–based 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. All components are treated in a uniform way by introducing *proxy* objects. A proxy is a local representative for a tool component in a different address space [5]. It hides the different mechanisms required for interacting with the real component. 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.

In order to support a flexible integration of different component implementation and communication platforms we use a reflective approach [6] for mapping the component model. With this approach the components and interactors form the *base level*. In addition 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*. A metaobject represents the following information about components or interactors:

- structural properties (interfaces, slots, channels, roles or links) and their mapping to the native or composite implementation
- mechanisms required for activating or instantiating the objects
- mechanisms for invoking operations and for registering with event announcements

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) is controlled by the related metaobject. When considering an operation invocation, the metaobject intercepts the request, selects the appropriated communication mechanism, deals with parameter conversation and finally invokes the real operation implementation. The invocation requires an additional request to the meta level, if the component represents a composition of other components.

The interception is carried out by the proxy between the component layer and the bridging layer. For that purpose every proxy contains a reference to the related metaobject and forwards the requests to this. Figure 3 illustrates the message flow resulting from the interaction between base and meta levels.

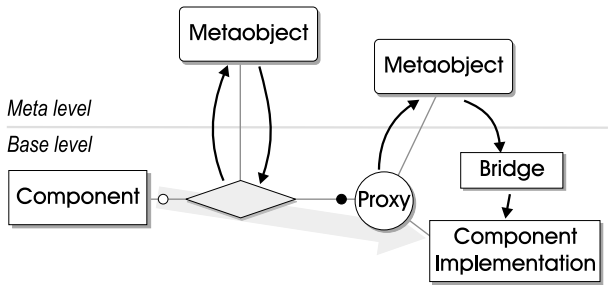

Figure 3: Message flow in an interactor

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 (the case of simple links), they can be directly connected. The communication takes place through component–owned mechanisms without intervention of framework services. The framework is responsible only for instantiating and interconnecting these components.

As a part of a complete tool environment the framework provides the services required for the control integration. The environment has to implement the services for managing the tool data. The activation of the tools and the coordination of their activities is managed by the framework services. Furthermore, the use of the framework enables the implementation of process management and user interface services without consideration of special mechanisms required for tool activation or communication.

4 Related Work

The work presented in this paper is a synthesis of results from various research areas, particularly software composition, coordination and tool integration.

The explicit representation of the interaction and coordination aspect of a software system is the subject of several recent contributions. Approaches like *formal connectors* [7], *synchronizers* [8] or *contracts* [9] introduce special abstractions for

decoupling the interaction part from the component implementation. Our concept of interactors was inspired by these ideas.

Darwin [10] 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. Furthermore, components can be composed of several already defined component instances. The *Darwin* language does not interpret service types, these are only used by the system environment. Bindings are allowed only between compatible service types. The semantics of these bindings are 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 [11] extends the *Darwin* approach with the notion of connectors. Connectors mediate the interaction between components. They define interaction rules in terms of the required interfaces, the protocols used for communication and a set of constraints. *Olan* provides several predefined connector types, but in the current version the configuration language does not support user–defined connectors. The runtime environment of *Olan* consists of a component–based virtual machine on top of an object request broker. Besides supporting the composition of components and connectors *Olan* provides services for application management. There is no information available on the integration of existing components and therefore on the suitability for tool environments.

The problem of interoperability between components with incompatible interfaces is discussed in [12]. This approach is based on an *Interface Adaption Language (IAL)* for describing the mapping between data types and object interfaces. At runtime the object mapping is carried out by an *object mapper* component, which generates so called "inter–objects" for representing remote servers automatically.

Apart from approaches addressing the composition and configuration of software systems, much work has been done in the area of tool integration and integrated environments. A comprehensive classification of existing CASE technology is presented in [13]. This classification considers production–process, metaprocess and enabling technologies and identifies three categories of CASE products: tools, workbenches and environments.

An important contribution to the area of integrated environments was the ECMA reference model [14]. This model defines a framework for constructing software–engineering environments. Control integration in this framework is based on a central message server, which was first introduced with the *Field* environment [15]. In *Field* tools communicate by sending and receiving messages via the message server. A tool sends a message to the server to announce specific events.Other tools can register with the server to be notified of events announced. With the notification these tools can process the corresponding service. However, the tools need to support interfaces and interaction protocols prescribed by the environment to be integrable. The integration of such tools, that do not support these interfaces, requires special adaptations.

The *ToolBus* architecture [16] 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. Tools can communicate solely through the ToolBus. Inside the bus two communica-

tion mechanisms are available: first the synchronous message passing between two tools and second the asynchronous sending of notifications (called "notes") to a group of interested tools. Existing tools have to be encapsulated by adapters, which are responsible for data transformation and communication protocol adjustment.

5 Conclusion

Integrated applications demand flexibility and openness of software environments. Standard applications with a fixed set of tools cannot support different development processes, enterprise–specific requirements, or the fast introduction of newest technologies in an adequate and cost–effective way. The request for the development of customized applications by interconnecting prefabricated elements seems to be evident.

In this paper we have presented an approach of a framework for tool integration. Our approach is based on a model which enables the explicit coordination of tool activities by using interactors as semantic relations. 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. A compiler generates the "glue" code required for the interface adaptation and the interaction control from a TIL specification. The runtime environment of the framework supports the instantiation and interconnection of the tool components. The mapping from the abstractions of the component model to the tool implementations and the integration of different communication platforms is supported by a reflective approach in conjunction with a bridging layer.

An initial release of the framework is implemented in Java [17]. This implementation is part of a project for developing configurable engineering environments. Tools like CAD modellers, word processors or calculation and management tools are treated as TIFRAME components. These components are implemented as CORBA objects (by "wrapping" existing applications) or Java applications. The control integration relationships of the components can be described completely in TIL.

Further work on the framework includes the development of mechanisms for process integration based on the notion of tasks associated with configurations as well as the support of specification activities by interactive tools.

References

[1] D. Schefström and G. van den Broek. *Tool Integration – Environments and Frameworks*. John Wiley & Sons, 1993.

[2] A.I. Wasserman. Tool Integration in Software Engineering Environments. In F. Long, editor, *Software Engineering Environments: Proc. Int. Workshop on Environments*, LNCS 467, pages 137–149. Springer Verlag, 1990.

[3] K.J. Sullivan. Mediators: Easing the design and evolution of integrated systems. Technical Report 94-08-01, Departement of Computer Science and Engineering, University of Washington, 1994.

[4] Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.0. OMG Document 97–02–25, July 1996.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, Reading, Mass., 1995.

[6] P. Maes. *Meta–Level Architectures and Reflection*. Elsevier Science Publishers B.V., 1988.

[7] R. Allen and D. Garlan. Formal connectors. Technical Report CMU–CS–94–115, Carnegie Mellon University, Pittsburgh, 1994.

[8] S. Frølund and G. Agha. A Language Framework for Multi–Object Coordination. In O.M. Nierstrasz, editor, *ECOOP'93 – Object–Oriented Programming: 7th European Conference*, LNCS 707, pages 346–360. Springer Verlag, July 1993.

[9] I.M. Holland. Specifying reusable components using Contracts. In O. Lehrmann Madsen, editor, *Proc. of the European Conference of Object Oriented Programming (ECOOP'92)*, LNCS 615, pages 287–308. Springer Verlag, 1992.

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schäfer and P. Botella, editors, *Proc. of the 5th European Software Engineering Conference (ESEC 95)*, LNCS 989, pages 137–153, Sitges, Spain, September 1995.

[11] L. Bellissard, S. Ben Atallah, A. Kerbrat, and M. Riveill. Component–based Programming and Application Management with Olan. In *Proc. of Workshop on Object–Based Parallel and Distributed Computation*, LNCS 1107. Springer Verlag, June 1995.

[12] D. Konstantas. Object Oriented Interoperability. In O.M. Nierstrasz, editor, *ECOOP'93 – Object–Oriented Programming: 7th European Conference*, LNCS 707, pages 80–102. Springer Verlag, July 1993.

[13] A. Fuggetta. A Classification of CASE Technology. *IEEE Computer*, pages 25–38, December 1993.

[14] European Computer Manufacturers Association. Reference Model for Frameworks of Software Engineering Environments. Technical Report ECMA TR/55 (3rd edition), June 1993.

[15] S. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[16] J.A. Bergstra and P. Klint. The ToolBus Coordination Architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, LNCS 1061, pages 75–88. Springer Verlag, 1996.

[17] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Mass., 1996.