Comparing Inter-Tool Communication in Control-Centred Tool Integration Frameworks

Jennifer G. Harvey Chris D. Marlin

Department of Computer Science,
Flinders University of South Australia,
Adelaide, South Australia

email: {jenny, marlin} @cs.flinders.edu.au

Abstract

Tool integration frameworks provide the devices needed to define and refine customised integrated software engineering environments. The customisation that they provide relates both to the specific tools populating the environment and the nature of the interaction between tools (i.e. the style of integration). A number of such tool integration frameworks are available, either as the results of research projects or as commercial products. Unfortunately for potential users or purchasers of these frameworks, it is unclear to what extent the provided integration devices can adequately describe the integration required in a particular situation. This paper presents progress towards an approach to the precise description of tool integration devices; this approach uses an operational model based on information structures to formally describe tool integration devices. The approach is illustrated by describing selected features of the integration devices of two control-centred tool integration frameworks – a research prototype framework, FIELD [27], and a commercial framework, Hewlett-Packard's SoftBench [6]. The paper shows how this approach facilitates the comparison of the features concerned and thus informs a discussion on the styles of integration which can be expressed in the two integration devices.

Keywords: software engineering environments, CASE, tool integration, tool integration frameworks, control integration, inter-tool communication, FIELD, SoftBench.

1. Introduction

Tool integration frameworks offer a reusable facility for the integration of software engineering tools; typically, they provide at least a communication mechanism, a data storage and control facility, and a vehicle for the construction of consistent user interfaces. In order to afford access to these facilities by the tools which populate a tool integration framework, the framework incorporates one or more integration devices, usually in the form of a specially developed programming language or extensions to an existing language; this language is used to describe the desired style of tool integration. The nature of the provided integration devices clearly limits the range of integration styles which can be expressed in a particular framework. Furthermore, although much work has been done defining and characterising both integration and these integration devices (e.g. [2-5,20]), there is little work which seeks to assess the expressiveness of the integration devices provided by tool integration frameworks. This is surprising, as the amount of support that an integrated environment can offer to software developers is determined both by the tool set provided and the manner in which the tools can cooperate to achieve a software development goal (i.e. the extent to which they are integrated).

Our work represents one approach to assessing and comparing the expressiveness of integration devices. The motivation for this work is described in [13], and [12] presents a layered, information structure model; this model is based on the work of Wegner [32] and Plotkin [25], and in the style of Marlin [15-18] and others (e.g. [7,17,18,21-24]). Specifically, the model has been developed to provide a formal approach to describing the semantics of the *inter-tool communication* features (encompassing control integration, together with some aspects of data integration) of integration devices. Our approach yields significantly more precise comparisons of the functionality provided by various frameworks than has been obtained with the less formal comparative techniques employed in the past (e.g., those surveyed in [11]). Furthermore, we illustrate how this approach facilitates comparisons of the features provided by the framework and how such a comparison assists with an assessment of the styles of integration which can be expressed using the respective devices.

This paper is organised as follows. Section 2 discusses the focus of our work, and briefly outlines the information structure model. In Sections 3 and 4, we illustrate the application of the model with descriptions of aspects of inter-tool communication in a research prototype framework, FIELD [28], and a commercial framework, Hewlett-Packard's SoftBench [6]. Section 5 discusses how the descriptions in Sections 3 and 4 elucidate differences which are significant to the construction and use of integrated environments. We characterise the different approaches identified as "tool driven" and "user driven".

0-8186-8019-9/97 $10.00

The paper closes with some concluding remarks and a discussion of ongoing work.

2. Modelling inter-tool communication

2.1. Inter-tool communication

As described above, this paper is concerned with the integration devices provided in tool integration frameworks. Specifically, we focus on *control-centred tool integration frameworks*, in which there is a separation of the issues of data management and communication between tools[1]. Our focus on control-centred integration should not be taken as an indication that control-centred tool integration frameworks are necessarily superior to the data-centred approach, merely that the modelling required for the two approaches is quite distinct and that we have chosen to study the former for the present.

In the case of control-centred tool integration frameworks, flexibility is achieved via variations in the tool set comprising an instantiated framework and through the ability to vary the modes of interaction between these tools. With respect to the latter, there will be some means to express the modes of inter-tool communication; this integration device is usually a language which may be highly specific or may be effectively an extension to some more general-purpose language. In any case, this language of inter-tool communication has a domain of discourse relating to services provided or to be provided by tools, and of data or references to data (since tools will typically manipulate data of some kind as part of carrying out the services they provide). Thus, in relation to the five themes of integration described by Wasserman [31], inter-tool communication includes not only control integration, but also some aspects of data integration, such as recording which data is relevant to the current project.

Thus, the model presented in this section has been developed to describe the inter-tool communication facilities provided by integration devices in a range of control-centred tool integration frameworks. In this particular paper, we focus on several issues of service provision. The descriptions presented in Sections 3 and 4 of the paper concentrate on these concerns as selected features of two frameworks are examined. Readers familiar with the languages involved may thus notice that unnecessary details (such as the full syntax of a language feature) are elided for the purposes of our discussion.

2.2. A motivating example

We introduce an example, which is revisited in Section 5, to illustrate the sort of information which is of concern to environment builders and users, tool integration language designers and integration programmers, and yet which is not freely available in vendor documentation or other formal and informal literature.

Suppose a tool, *ToolA*, requires that an editor, *ToolB*, load some data, say *prog.c*, ready for editing. *ToolA* sends such a request to *ToolB*. Can *ToolA* suspend operation until the request is complied with? Will it be notified of the outcome of the request? What assumptions can *ToolA* make if a reply to the request is not expected? Can it continue operation independent of any reply?

Suppose that, some time later, *ToolA* requests the termination of *ToolB*. However, unsaved editing changes have been made to *prog.c*, so *ToolB* asks the user whether to save the file before terminating. Suppose that the user cancels the termination request sent from *ToolA*. Is *ToolA* informed of this event? Can *ToolA* override the user's request? If *ToolA* is not informed of the user's action, what assumptions can it make in subsequent operation?

Suppose that *ToolA* wishes *ToolB* to suspend operation while *ToolA* performs some processing without disturbance from *ToolB*. Can it request such suspension and, later, resumption of processing? Can it request, for example, that *ToolB* disable various user options?

The answers to questions such as these provide valuable insight into the *integration styles* used by various tool integration frameworks. While several current efforts are concentrating on characterising the differences between a number of different tool integration framework architectures such as realised in CORBA, SoftBench and Polylith, as in [1], our work is focused on identifying the differences between implementations of one specific architecture – the control-centred tool integration architecture introduced in Section 2.1.

2.3. The layered operational semantic model

The model used to describe the inter-tool communication facilities is an information structure model [32]; in such a model, a collection of objects, known collectively as *information structures*, are defined to characterise those aspects of interest in the system under examination, and the semantics of the relevant aspects of the system's operation are described as manipulations on the information structures. The manipulations are formulated with primitive operations and other, "higher-order", operations defined by the model.

The various components of our model fuse into a layered model, as illustrated in Figure 1. This figure shows that the model has five layers; from the lowest to

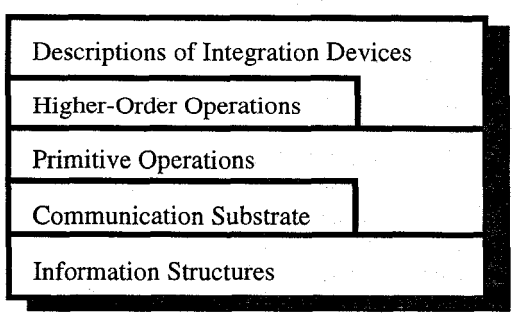

Figure 1. A layered model of inter-tool communication.

[1] Alternatively, in *data-centred integration*, both data management and tool communication are embedded in the environment repository.

the highest, these are:

- the information structures themselves, describing those aspects of the state of tool integration frameworks which relate to inter-tool communication,
- the communication substrate, covering the delivery and receipt of communication messages by tools,
- primitive operations, defining elementary information structure manipulations, such as insertion and deletion operations, and basic communication primitives, such as *send* and *receive,*
- higher order operations, defining more complex operations, such as setting up a communication channel between two tools, and
- the actual descriptions of integration devices.

to the messages which can be sent on that channel. The basic building blocks of the model, messages and tools, are represented by information structures. The `Message` information structure is depicted in Figure 2. It conveys information identifying the message (`messageID`), the sending tool (`senderID`) and the target tool (`recipientID`), as well as the data contents of the message (`messageData`), and the mode of the message (`messageMode`). The last of these, the message mode, indicates the message type, and thus determines how the message is to be interpreted by the recipient; the message modes supported by the model include `"Req"` (for a request message), `"Not"` (for a notification), and `"Repl"` (for a reply).

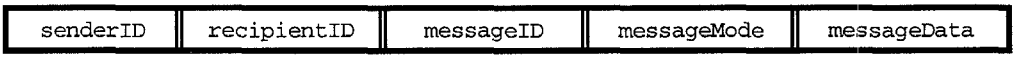

Figure 2. The *Message* information structure.

In Figure 1, the horizontal lines indicate that each layer is defined in terms of the layer below. Thus, the communication substrate layer is defined only in terms of the information structures defined in the information structures layer, whereas the primitive operations layer is defined in terms of both the communication substrate layer and the information structures layer. Likewise, the descriptions of the integration devices, forming the uppermost layer, utilise the higher-order operations and the primitive operations. While the higher-order operations are not strictly essential, they provide a convenient method for eliding various details of processing which are constant across the tool integration frameworks under consideration, thus facilitating the comparison of descriptions of integration features.

By developing a model that consists of several layers, it is possible to have a single description that caters for the different information requirements of various groups, providing clarity while presenting the detail when required; this notion of a layered model has also been explored by Oudshoorn and Marlin [21-23] in the context of the description of programming languages. For example, tool integration framework designers and tool integration language designers can obtain the precise definitions that they require, whilst integration programmers and other interested groups can read to the level most convenient to them.

The concept of the model is one of tools which communicate by sending messages on communication channels. Each communication channel is typed according

A tool publishes its integration signature, that is, a list of the messages which it is willing to receive (represented as pattern strings) and a list of messages that it can emit (represented as strings). Each tool is represented by a `ToolCommunications` information structure, depicted in Figure 3; each such structure consists of the tool's unique identification (`toolID`), a list of the patterns published by the tool (`inputMsgs`) and the messages emitted by the tool (`outputMsgs`). Each entry in the `inputMsgs` list consists of:

- a `pattern`, which is a string representing a message of interest to this tool,
- a `patternMode`, which will be either `"Req"`, `"Not"`, or `"Repl"`, and which indicates the required delivery mode of input message, and
- a list called `patternBindings`, which records the identity of various tools from which the messages represented by the `pattern` and `patternMode` of this particular inputMsgs entry can be received.

Similarly, each entry in the outputMsgs list consists of:

- `msg`, the content of the message to be sent,
- a `msgMode`, which is the mode of the message to be sent, and
- a list called `msgBindings`, which records the identity of various tools to which the messages represented by this particular outputMsgs entry can be sent.

It is apparent from the contents of the `ToolCommunications` structure that valid communication

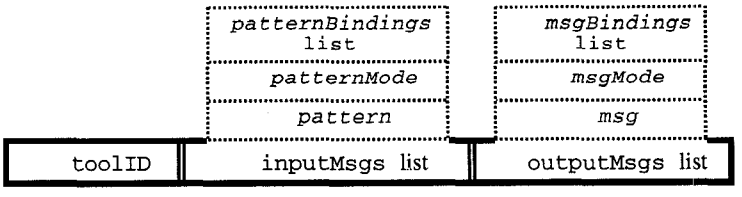

Figure 3. A *ToolCommunications* information structure.

connections between tools can be established by binding entries in the patternBindings list of the inputMsgs list in one tool to entries in the outputMsgs lists of other tools; note that the patternBindings lists and the msgBindings lists are reciprocal, in the sense that where a pattern in *ToolA* is bound to a message in *ToolB*, the same message in *ToolB* will be bound to the pattern in *ToolA*.

Indeed, the contents of these lists can be determined statically, and this suggests that communication connections between tools could be determined statically. However, while the set of output messages remains constant for the execution lifetime of the tool, the range of valid input messages for a tool may vary. For example, a tool may wish to accept a certain message for a short period of its operation only; this would be modelled by dynamically altering its inputMsgs list to insert an entry corresponding to the message or messages which are to be received for this time period and then removing these entries later.

A number of primitive and higher-order operations (the second and third layers in Figure 1) are required to describe realistic integration devices; because of limited space, only a brief indication of their usefulness can be presented here. Primitive operations provide insertion, deletion, update and search operations, an iterator, a selector and a matching operation, and basic communication facilities. Three primitives deserve to be mentioned explicitly:

- `noAction`
  indicates that a tool remains idle in terms of inter-tool communication,
- `suspend operation except for {bindings}`
  causes all communication bindings to become inactive except for those listed in bindings (thereby causing the tool to cease receiving messages from the inactivate bindings), and
- `resume operation`
  effects a resumption of message processing, which ceased due to the use of a `suspend` operation.

An example of a typical statement involving a primitive operation might be:

```
A ← find ToolCommunications where
        {toolID == thisTool.toolID};
```

The syntax of this statement reflects the syntax style used for most primitives and higher-order operations in the model, indicating the operation being invoked (`find` in this case), the information structure to which the operation is applied (`ToolCommunications`), and the parameters being transmitted (in this case, the part `where {toolID == thisTool.toolID}`). This particular statement locates the `ToolCommunications` information structure that has its `toolID` attribute equal to the value of the variable `thisTool.toolID`, and returns the result in the variable `A`. A backwards arrow, ←, indicates an assignment and may be used in the primitives' parameters, as in:

```
B ← create new ToolCommunications where
        {toolID ← "EDIT"};
```

In this case, a new `ToolCommunications` information structure is created, and its `toolID` attribute is given the value `"EDIT"`. A pointer to the new structure is assigned to `B`.

There is one higher-order operation used in the descriptions in this paper,

```
START tool where {operation = operation};
```

which causes a tool, currently inactive in the environment, and which can provide *operation*, to register its initial integration interface (and thereby become a contributing part of the environment); this operation returns `TRUE` if successful and `FALSE` otherwise.

In addition, the model provides two variables, `thisTool` and `lastMsg`. The former returns the `ToolCommunications` information structure for the tool under consideration, and `lastMsg` returns the most recent message that was received by the tool. The variables can be used to select attributes of the information structures, as in `lastMsg.messageID`. The descriptions of integration devices also make use of temporary variables to store intermediate values; these are denoted by capital characters, such as `A` and `D`.

2.4. Describing inter-tool communication language features

As mentioned before, the model presented here is a layered information structure model. In order to complete the uppermost layer of Figure 1 (covering the descriptions of integration devices), it is necessary to provide adequate descriptions of the transformations of the information structures caused by the various relevant language features. This is done by giving an algorithmic description of an *event* corresponding to each inter-tool communication language feature. These events describe the semantics of communication between tools by setting up integration interfaces and communication bindings, rearranging interfaces and bindings, and transferring information to and from the integration interfaces. Each of the algorithms is regarded as a set of actions executed in place of the language feature it describes. The descriptions are Pascal-like, using features of the language in conjunction with the model primitives and higher-order operations. More details may be found in [12].

Integration devices can be regarded as having three groups of inter-tool communication events. First, there is a group concerned with *integration interface specification*, and includes publication of the *Notification interface* (information messages that will be accepted) and the *Request interface* (services that will be offered to the environment). The second group, *message sending*, incorporates sending of Notification, Request and Reply messages. The final group is concerned with *message reception*, and is comprised of the receipt of Notification, Request and Reply messages. Hence, we determine the following communication events:

(1) Notification_publication,
(2) Request_publication,
(3) Notification_send,
(4) Request_send,
(5) Reply_send,
(6) Notification_receive,
(7) Request_receive, and
(8) Reply_receive.

The next two sections of the paper describe relevant features of integration devices provided by each of SoftBench and Field. Because of limited space, only a selection of these features can be presented. Those chosen for the purposes of illustration are the communication events: (4) Request_send, (5) Reply_send, and (8) Reply_receive.

3. The EDL tool integration language of SoftBench

3.1. The SoftBench tool integration framework

Hewlett-Packard's SoftBench tool integration framework embraced the message-server technology pioneered by Field to fashion the Broadcast Message Server (BMS). SoftBench is furnished with several integration devices with which to describe tool *encapsulations* – a shell script facility (ciclient), header and library files for C and C++, and the Encapsulation Description Language (EDL) [8]. EDL is a C-like specification language, designed specifically for tool integration; it is for this reason that the EDL integration device was selected for examination in this work.

The conceptual model of integration in SoftBench is one of *events* and *actions*. In order to communicate with the rest of the environment, a tool encapsulation

- defines its interface by specifying and publishing the events in which the tool has an interest, and associates one or more actions and tool services with each event – this can be done at any time during the tool's execution lifetime, and a published event can be withdrawn if it is no longer of interest;
- communicates with other tools by generating events, usually after completion of some action by the tool, so that other tools can both monitor the operation of the tool and react accordingly.

Events are published as pattern strings and generated as messages; a tool will receive an event only if the generated message matches one or more of its published pattern strings. SoftBench supports three message types: Request, Notification and Failure messages. Failure messages are generated to indicate the lack of success in fulfilling the requirements of a Request message (successful completion is indicated with a Notification message). Figure 4, adapted from [9], illustrates the interaction between the BMS and an encapsulated tool.

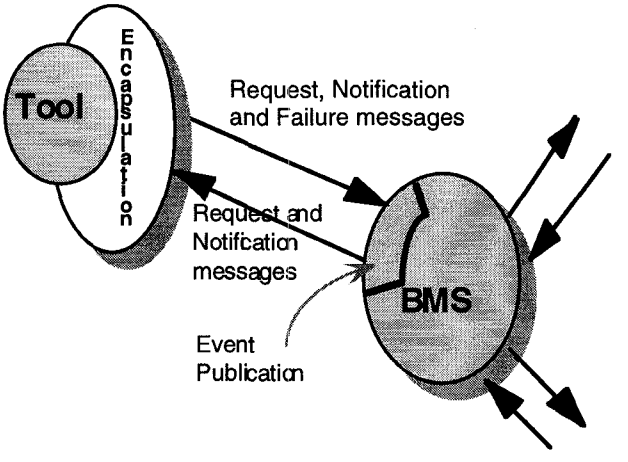

Figure 4. The BMS architecture of Hewlett-Packard's SoftBench.

SoftBench supports a tool class concept, which determines the minimum set of messages that should form a tool's integration interface. Accordingly, the BMS maintains a database of tool classes, a list of known tools in each class, and information about each tool's location and invocation details.

```
1  ID = make_message_id();
2  NP = make_message_pattern(Notify, ID, patternN);
3  NE = make_event(MESSAGE, NP, callbackfunctnN);
4  add_event(NE);
5  FP = make_message_pattern(Failure, ID, PatternF);
6  FE = make_event(MESSAGE, FP, callbackfunctnF);
7  add_event(FE);
8  /* send the request message */
9  send_message(Request, requestedService, ID);
```

Figure 5. Defining and publishing reply events in EDL.

3.2. The description of features of the EDL tool integration language

Section 2.4 described three groups of communication events, and selected (4) Request_send, (5) Reply_send and (8) Reply_receive for elaboration in our descriptions of integration devices. In the language features of EDL, the Request_send and Reply_send communication events relate to the `send_message` statement. The Reply_receive event does not occur as the result of a specific language feature, but as part of the event processing loop of EDL.

Request_send

Consider the statement

`send_message(Request, requestedService, ID);`

where `Request` is the message type, *`requestedService`* and `ID` represent fields or collections of fields to be completed by the user, and where *`ID`* is an optional field. For Request messages, it is assumed that the integration programmer will define and publish events, as described in Section 3.1, which will recognise specific replies from the recipients of the Request message which are of interest to this tool. To denote an event as a reply to a specific request, an identifier is produced for the message and included in each reply event. Figure 5 illustrates this. In this example, italicised items represent fields or collections of fields for which the programmer substitutes values. Line 1 generates a string which is used to identify both the message and events. A Notification event is defined (lines 2 and 3) and published (line 4), and a Failure event is also defined and published (lines 5 – 7). Both of these events will trap replies to the request message, sent in line 9. Control is returned to the encapsulation immediately after the Request message is sent.

Figure 6 shows the algorithmic description of the Request_send event, which is the communications event corresponding to this language feature. The description consists of three phases. Phase 1 establishes the set of communication bindings that are associated with the Request message by locating the entry in the tool's `outputMsgs` list that matches *both* the message mode and the contents of the *`requestedService`* field; this occurs in lines 1 to 3. If there are no communication bindings associated with the output message, an attempt is made to locate and invoke a tool which can process the request (line 5). Recall that this is possible because SoftBench maintains a database of tool classes. If an appropriate tool is invoked, and therefore has published its integration signature, the set of communication bindings is again determined (lines 6 through 8). If the set is empty (i.e. no tool willing to accept the request message was found), phase 2 returns "Fail" as a reply to the encapsulation. Phase 3 occurs when a list of communication bindings exists. Here, the message is generated and sent to each tool bound to this output message. Note that the identification used for the message, *ID*, is generated separately by the user (as in Figure 5, line 1).

```
Request_send →
            1   B ← find item in thisTool.outputMsgs where {                    /* phase 1 */
            2       msg == requestedService,
            3       msgMode == "Req" };
            4   if B.msgBindings == NULL then
            5       if START tool where {operation == requestedService} then
            6           B ← find item in thisTool.outputMsgs where {
            7                   msg == requestedService,
            8                   msgMode == "Req" };
            9       end if;
           10   end if;
           11   if B.msgBindings == NULL then                                   /* phase 2 */
           12       theReply ← "Fail";
           13   else                                                            /* phase 3 */
           14       for all A in B.msgBindings do
           15           send message to A.toolID where {
           16                   messageID   ← ID,
           17                   senderID    ← thisTool.toolID,
           18                   messageMode ← "Req" };
           19       end for all;
           20   end if.
```

Figure 6. The Request_send event in EDL.

```
Reply_send event →
1     either
2          [ either  [ A ← "Success";
3                    | A ← "Fail"  ];
4            send message to lastMsg.senderID where {
5                 messageID   ← lastMsg.messageID,
6                 messageMode ← "Repl",
7                 messageData ← A,
8                 senderID    ← thisTool.toolID };
9          | noAction].
```

Figure 7. The Request_receive event in EDL.

Reply_send

The Reply_send event also relates to the `send_message` command. The formats used for Reply messages are:

```
send_message(Notify, "Success", requestID);
send_message(Failure, requestID);
```

The first format indicates a successful completion of the requested service, and the second represents a failure to provide the service. As noted before, there is no obligation for an encapsulation to reply, to limit the number of replies sent, or to define and publish events to recognise possible replies.

The algorithmic description in Figure 7 describes the Reply_send communication event in EDL. A Reply message in the model will indicate either "Success" or "Fail" (lines 2 and 3), and includes the `messageID` field of the associated Request message (line 5).

Reply_receive

The Reply_receive event is defined by the algorithmic description in Figure 8, which describes the effect of the receipt of a Reply message – there is no associated inter-tool communication operation. Note that the events defined and published previously in order to recognise this and other replies to the Request message will be removed from the integration interface of the tool only if the integration programmer specifies such removal.

3.3. Discussion

From the previous two subsections, we have seen that the request-reply sequence in SoftBench's EDL consists of the following steps:

(1) A tool, say *ToolA*, defines and publishes zero or more events which will recognise incoming Reply messages of interest associated with the Request message to be sent (Figure 5).
(2) *ToolA* composes and sends the Request message (Figures 5 and 6).
(3) If no tools exist which can service the request, a "Fail" reply is returned (Figure 6).
(4) If one or more tools can service the request, they might not return a Reply message, or will reply indicating "Success" or "Fail" (Figure 7).
(5) If a reply has been sent, it will be received by *ToolA* if it defined and published an event to recognise this reply (Figure 8). *ToolA* might remove this event and/or other reply events, or it might leave some or all of these events in place.

The last step is demonstrated more clearly in the finite state machine in Figure 9. In this figure, transitions are depicted as arrows, transition events are placed above the transition, and actions taken upon a transition are in bold, and located below the transition. Transition guards are placed in square brackets at a transition source. Variables and semantic structures are used in the diagram, and these are indicated in italics. The diagram extends the description of the Request_send and Reply_receive event descriptions by demonstrating the interleaved operation of the encapsulation. For EDL, it also elucidates the importance of the additional pre- and post-event processing required to make the Request_send event meaningful and to distinguish the effect of a tool sending a Request message from that of sending a Notification message.

The transition from the start state, *Send request*, to the state *Process replies* is caused by a Request_send event – at this time, the identifier of the Request message and the events published by the tool to recognise replies to this message, *rEvents*, are recorded. The first part of the figure illustrates particularly well the difficulties that arise from the flexibility provided by the Request_send and Reply_receive language features of EDL. Here there is no transition from *Process replies* to *Send request* – given the right conditions a tool may remain in this state indefinitely, continuing to receive replies for any previous Request_send communication event. Naturally, this is not the intention behind these language features. EDL requires that they be used in conjunction with the features used normally to establish and remove an integration interface

```
Reply_receive event →
     theReply ← lastMsg.messageData.
```

Figure 8. The Reply_receive event in EDL.

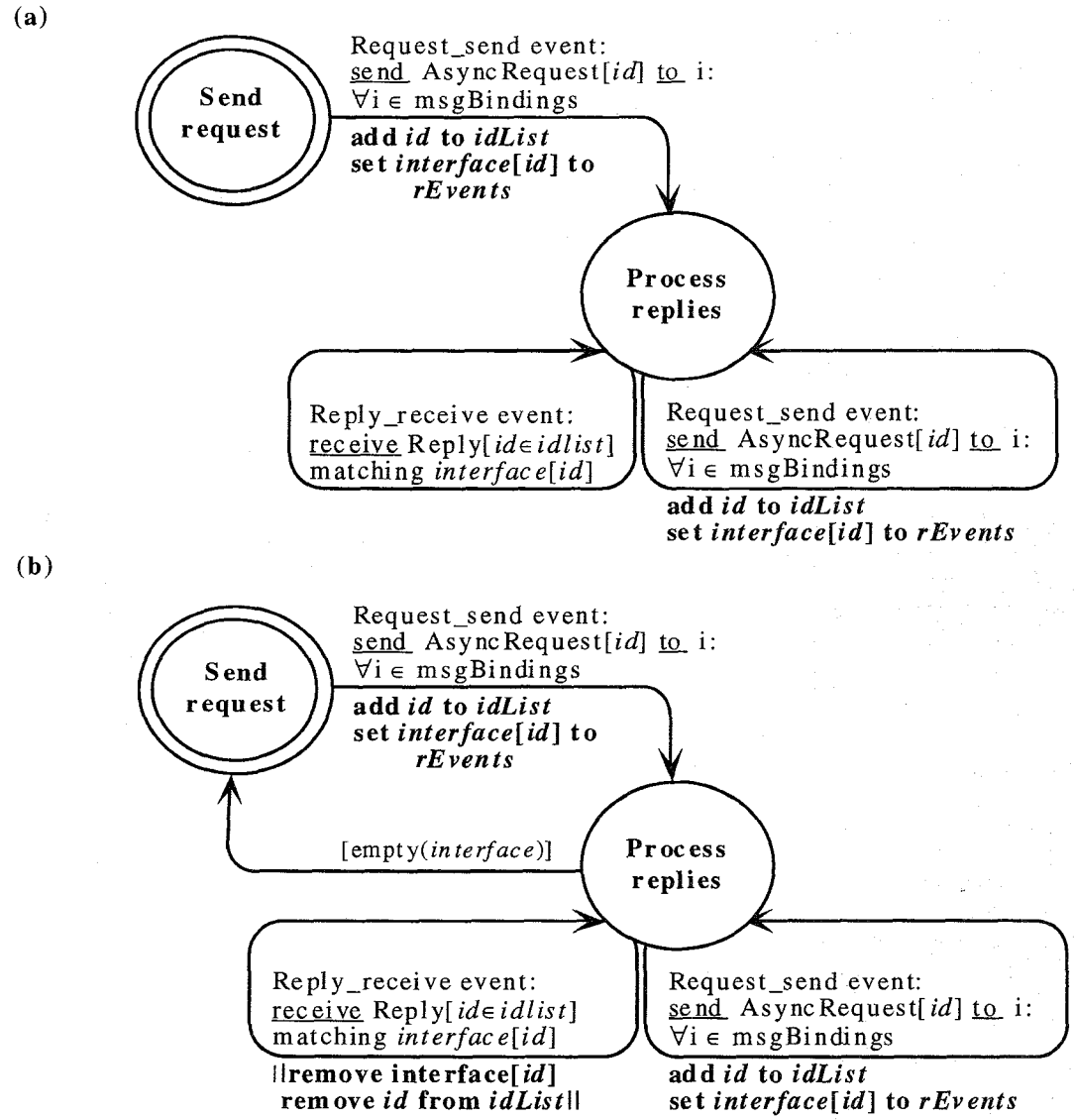

Figure 9. Finite state machine representing the Request_send and Reply_receive events for EDL.

(as in steps (1) and (5) above, and demonstrated in Figure 5).

The second part of Figure 9 includes the usual pre- and post-processing specified by an integration programmer. In this case, as each reply is received, the reply events might be deleted from the *interface* list and the identifier removed from the list of message identifiers – the optional nature of the actions is indicated by enclosing them in vertical bars. Additionally, the start state will again be achieved if there are no Request_send events remaining for which Reply_receive events can occur.

The situation of a Request_send event where no reply events have been defined (i.e., *rEvents* is empty) is illustrated clearly in the second part of the figure. In such a case, the *interface* list will be devoid of content, and the tool will return immediately to the start state. Such a scenario mirrors the Notification_send communication event, which is event (3) in the list given in Section 2.4 and whose description has been omitted here for the sake of brevity.

4. A description of inter-tool communication in Field's MPI

4.1 The Field tool integration framework

Field (the Friendly Integrated Environment for Learning and Development) [26-29] is a research prototype tool integration framework that first demonstrated that practical integrated tool sets are possible, using a message passing framework. The ideas exhibited by Field form the basis for many of the current generation of tool integration

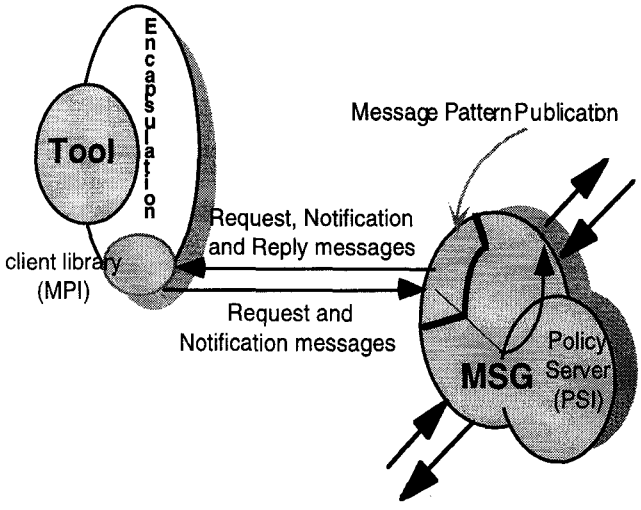

Figure 10. The MSG architecture of Field, showing the MPI and PSI integration devices.

frameworks and software engineering environments, such as DEC's FUSE [19], Sun's SPARCworks, SGI's CodeVision and Hewlett-Packard's SoftBench (described in Section 3).

The conceptual model used by Field is that of control via messages. Tools communicate by sending and receiving messages; the routing of messages is the responsibility of a central message server (MSG) which records details of operational tools and their interfaces as sets of message patterns. Tools register with the MSG upon invocation, and, at the same time, indicate with message patterns the set of messages that they have an interest in receiving. In this way, tools publicise their integration interface. The set of message patterns may change during the execution lifetime of the tool – a tool is free to deregister current message patterns and to register new message patterns. When a tool receives a message which matches one of its currently registered message patterns, an associated function is invoked. Field supports three message types: Notification messages, Request messages and Reply messages.

Rather than providing a specific integration language, Field provides two complementary facilities for describing required integrations – the MSG Program Interface (MPI), and the Policy Server Interface (PSI). The MPI is a message client library, providing entries needed to send and receive messages. The PSI derives from the Forest environment [10,14,30], and supports the MPI by providing the ability to translate messages before they reach the message server and by ensuring priority processing of designated messages. A PSI is described using a Policy Language based on that provided by the Forest system which defines the actions to be taken on *message-tool-user* triplets; this allows for different actions to be defined for various *tool-user* group combinations. The MPI, rather than the PSI, will be the focus of the descriptions in this paper. Figure 10, adapted from [26], illustrates the message server architecture and the MPI and PSI devices.

4.2. The description of aspects of the Field tool integration language

Extensions to the basic model

The description of the MPI language features uses a special list – an additional attribute, `replyData` list, attached to the `ToolCommunications` information structure. This is shown in Figure 11. There will be one entry in this list for each Request message emitted by the tool for which a reply remains outstanding. It will contain two fields. The first, `rID`, records the identifier associated with the Request and Reply messages. The next, `rCount`, keeps a record of the number of outstanding Reply messages, and is initialised to zero.

Request_send

The Request_send event in MPI is divided into two sub-events, which we will call Sync_Request_send and Async_Request_send. The MPI function

```
MSGcall(MSGID, requestedService);
```

relates to the Sync_Request_send event, and suspends operation until a reply is received. The function

```
MSGcallback(MSGID, callbackFunctn,
            requestedService);
```

is the language feature related to the Async_Request_send, and returns control immediately to the tool – thus, it specifies a function to be called upon receipt of a reply by the tool's encapsulation. Figure 12 shows the algorithmic description of the Request_send communications event corresponding to these functions. It comprises three phases. Phase 1 is comparable to the phase 1 of Figure 6 (the Request_send description of EDL); however, it is apparent at lines 2 and 6 that the criteria for locating the corresponding output message is less restrictive than that

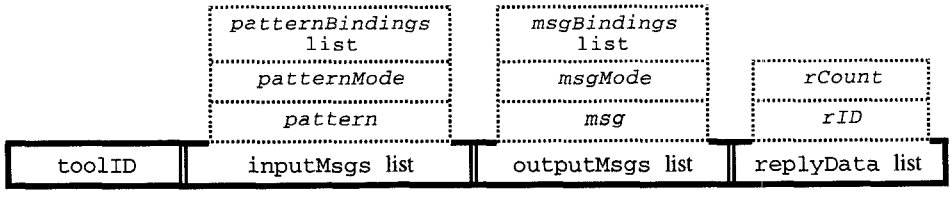

Figure 11. The extended ToolCommunications information structure.

```
Request_send →
1    A ← find item in thisTool.outputMsgs where {                    /* phase 1 */
2        msg = requestedService };
3    if A.msgBindings = NULL then
4        if START tool where {operation = requestedService} then
5            A ← find item in thisTool.outputMsgs where {
6                msg = requestedService };
7        end if;
8    end if;
9    if A.msgBindings ≠ NULL then                                    /* phase 2 */
10       MSGID ← provide_msgID;
11       B ← insert item in thisTool.replyData where {
12               rID ← MSGID };
13       for all C in A.msgBindings do
14           B.rCount ← B.rCount + 1;
15           send message to C.toolID where {
16               messageID   ← MSGID,
17               senderID    ← thisTool.toolID,
18               messageType ← "Req" };
19       end for all;
20       /* for the Sync_Request_send event */                      /* phase 3 */
21       suspend operation except for {A.msgBindings};
22   end if.
```

Figure 12. The Request_send event in MPI.

required by EDL. Note that if the set of communication bindings for the Request message remains empty at the end of phase 1, no further action is taken.

Phase 2 proceeds if communication bindings exist. Firstly, in line 10, a message identifier is produced for use in the Request message and all subsequent replies that are received from the recipients of the request. At lines 11 and 12, a new entry is inserted into the `replyData` list of this tool's `ToolCommunications` information structure. Then, the count of Reply messages expected is incremented for each communication binding (lines 13 and 14), before the Request message is sent (lines 15 through 18). Phase 3, on lines 20 and 21, will only occur for the Sync_Request_send event.

Reply_send

The language feature of MPI which corresponds to the Reply_send communication event is

```
MSGreply(MSGID, replyData);
```

The algorithmic description of this feature is shown in Figure 13. The description is similar to that of EDL, but lacks the option to refrain from sending a reply.

Reply_receive

The receipt of a Reply (or Notification or Request) message occurs as part of the event loop of MPI. Again we define two sub-events – a Sync_Reply_receive event that occurs in response to a Sync_Request_send event, and an Async_Reply_receive event that occurs in response to an Async_Request_send event. The algorithmic description of these in Figure 14 comprises three phases. The first locates the `replyData` entry for the incoming reply. The reply is ignored if no matching entry is found (line 3). Phases 2 and 3 expose the stricter control of the Reply_receive communication event that Field preserves.

```
Reply_send event →
1        [ either  [ A ← "Success";
2                  | A ← "Fail"  ];
3          send message to lastMsg.senderID where {
4              messageID   ← lastMsg.messageID,
5              messageType ← "Repl",
6              messageData ← A,
7              senderID    ← thisTool.toolID }.
```

Figure 13. The Request_receive event in MPI.

```
Reply_receive event →
1    A ← find item in thisTool.replyData where {                    /* phase 1 */
2            rID == lastMsg.messageID };
3    if A not NULL then
4        if lastMsg.messageData == "Fail" then                      /* phase 2 */
5            A.rCount ← A.rCount - 1;
6            if A.rCount == 0 then
7                theReply ← "Fail";
8                remove item from thisTool.replyData where {A};
9                resume operation; /* for a Sync_Reply_receive */
10           end if;
11       else if lastMsg.messageData = "Success" then               /* phase 3 */
12           theReply == "Success";
13           remove item from thisTool.replyData where {A};
14           resume operation; /* for a Sync_Reply_receive */
15       end if;
16   end if.
```

Figure 14. The Reply_receive event in Field's MPI.

The last "Fail" reply (phase 2) or the first "Success" reply (phase 3) is returned to the tool encapsulation as the unique reply to be processed. In phase 2, "Fail" messages cause the count of Reply messages received to be decremented (line 5) and checked to determine whether the current reply is the last expected (line 6). If so, this is set as the unique reply message (line 7), and all other communication channels are reopened for a Sync_Reply_receive event (line 9).

In phase 3, the first "Success" message is designated as the reply, and all other communication channels are reopened for a synchronous Request_send event. Note that in phase 2 and phase 3, the entry in `replyData` for this message is removed (lines 8 and 13); therefore, once a reply has been specified, further Reply messages bearing the same identifier will be ignored.

4.2.3. Discussion

Field supports both synchronous and asynchronous Request messaging (Figure 12), and guarantees that there will be only one reply processed for each request (Figure 14). The Sync_Request_send event causes an encapsulation to

- cease the generation of further Request_send events, *and*
- delay Reply_receive events which match any other previous Async_Request_send events

until the appropriate Reply_receive event for this message has occurred. An encapsulation continues to operate in a normal manner, receiving and sending messages, after an Async_Request_send event. Figure 15, representing the Request_send and Reply_receive events using finite state machines, clearly demonstrates these differences.

The left-hand side of the diagram illustrates the Synchronous Request–Reply sequence. From the start state, *Send request*, a Sync_Request_send event causes a transition to the *Gather replies (Sync)* state. From here, the only option is to return to the start state, via the *Process reply* state, after one of the designated replies is received.

The right-hand side depicts the asynchronous Request-Reply sequence. Here, the *Gather replies (Async)* state demonstrates clearly that a tool can generate further Request_send events of either type (by remaining in this state or transitioning to the *Gather replies (Sync)* state). Furthermore, after a Reply_receive event from either of the *Gather replies* states has caused a transition to *Process reply*, the system returns to *Gather replies (Async)* if Async_Request_send events remain for which replies are outstanding (indicated by the contents of the variable *idList*).

5. Comparison of the language features

The layered model for inter-tool communication has been used to present clear and precise descriptions of inter-tool communication events in tool integration frame-works. The precision engendered by the model exposes the differences in the semantics of the relevant aspects of the tool integration frameworks under scrutiny. Most often, these differences are not discernible from the framework documentation provided by the vendor, nor from other literature. In other cases, descriptions of the language features in the documentation are ambiguous, or descriptions of the frameworks and the integration languages are contradictory. In both cases, the model serves to clarify the semantics. Finally, where the semantics of a language feature or group of language features are known, the impact in terms of the integration style that can be achieved because of those semantics is frequently not apparent. Again, the model indicates the style or styles of integration supported by a framework.

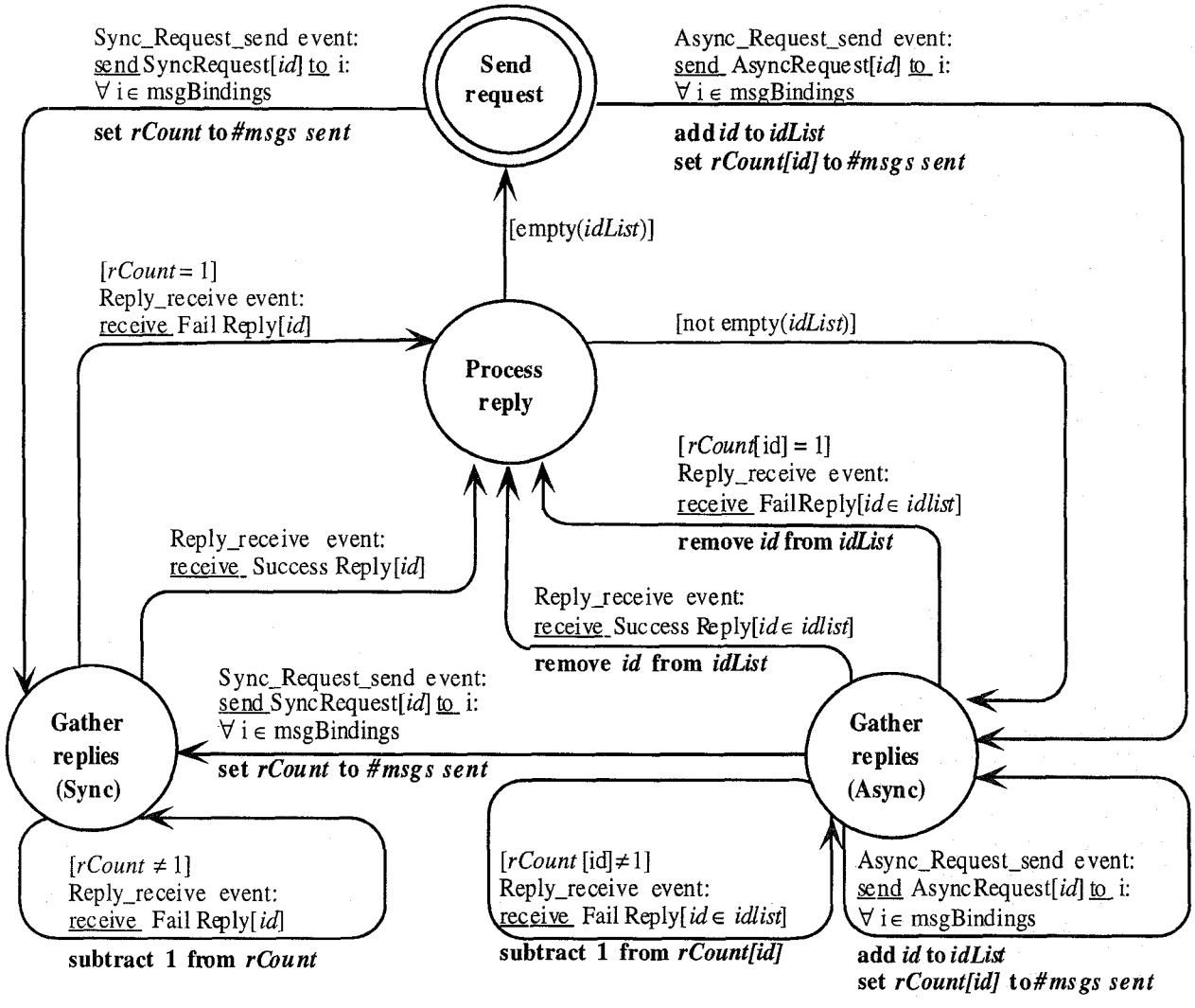

Figure 15. Finite State Machine representing the Request_send and Reply_receive events in Field's MPI.

The previous sections have demonstrated this clarity with descriptions of the Request-Reply component of inter-tool communication in two tool integration frameworks – SoftBench and Field. The language features described will now be compared. Some aspects of the comparison are relatively obvious from the informal descriptions of the language, but others are more subtle and only revealed by a detailed examination such as this.

Both languages permit some degree of dynamic determination of their integration interface and hence the set of tools from which they will accept messages and to which they will send messages. Both tools base this on declared message patterns. In the case of EDL, the matching that occurs between a tool's output message and another tool's input pattern will include the message mode (either Request, Notification or Failure). Field has less restrictive matching requirements which ignore the message mode and thus allows one pattern to be used for Request and Notification messages.

It is clear from their informal descriptions that Field supports synchronous Request messages with a language feature, and that SoftBench does not support it in this way. It is claimed in the informal literature that synchronous behaviour (that is, delaying the processing of messages until a reply is received) is programmable in EDL. What is not clear, however, is how this can be achieved, nor, indeed, the design decision which lead to only asynchronous messaging being provided. We will address the first issue here and the second will be addressed later in this section.

Consider Figure 9, which depicts the Request-reply sequence of EDL. Three steps are necessary to emulate the semantics of a synchronous request (that is, to delay processing of incoming messages). The first step is to remove the contents of the array *interface* so that no

further Reply_receive events can occur. This will ensure that the encapsulation is in the start state. Secondly, the Request_send event transition from the *Process replies* state must be disabled so that when the transition from the start state to this state occurs, no further Request_send events can be generated. Finally, the desired Request_send event occurs; this places the encapsulation in the *Process replies* state, where the only replies that can be accepted are those which have been specified for this event. Once the required number of Reply_receive events have occurred, the disabled Request_send transition is re-enabled. The encapsulation can remain in this state, or return to the start state if the interface entry for the initial Request_send event is removed. Although it is true that it would have been possible to work out what is involved in emulating a synchronous request in EDL, our modelling technique has made the complexities of doing so particularly apparent.

The descriptions of the Request and Reply communication events highlights differences in the flexibility of the Request-reply process. Field is quite rigid in its approach, allowing the programmer no option – for every Request message, exactly one Reply message will be received. SoftBench, however, exhibits more versatility, providing the integration programmer with many options which include the following extremes:

- a tool does not define any reply events; it requires some service to be provided, but is not interested in the result,
- a tool defines arbitrarily many reply events to capture some specific Notification and Failure replies (and perhaps ignores other possible replies),
- a tool refrains from replying to a Request message,
- a tool replies many times to a Request message,
- a tool receives one reply to a Request message, and then removes all defined reply events for that request, and
- a tool never removes the defined reply events.

As an example of the implication of this versatility, consider the following implementation of the scenario introduced in Section 2.2:

- *ToolA* sends a Request message "`START EDIT prog.c`"[1], which is received by *ToolB*.
- *ToolB* loads *prog.c*, and replies "`EDIT SUCCESS`" to *ToolA*. The file, *prog.c,* is now available for editing (either by a user, or via Request messages from a tool).
- *ToolA*, some time later, sends a Request message "`STOP EDIT prog.c`", which is received by *ToolB*. *Tool A* has possibly defined and published events *S* and *F* to recognise Reply messages associated with this Request message.
- As unsaved editing changes have been made to *prog.c*, *ToolB* asks the user whether to save the file before unloading it. The user responds by telling *ToolB* to cancel the `STOP` request. No Reply messages are sent to *ToolA*.
- *ToolA* continues operation, and does not remove events *S* and *F* from its published interface.

This scenario would appear to indicate that the communication between (at least) two tools cooperating to support the editing of source code is insufficient, as *ToolA* is not informed of the outcome of the `STOP` request. What it demonstrates, however, is the "user-driven" integration style employed by SoftBench, where tools make few assumptions about the fine-grained processes employed by users engaged in software development. The user-driven model of SoftBench expects the user to initiate inter-tool communication by announcing the next required action. To accommodate this interaction style, SoftBench tools incorporate an extensive menu system with which the user can invoke tools and through which the user can interact with other tools. In addition, an Execution Manager tool is provided via which any tool can be invoked or terminated by the user.

We characterise the model of tool interaction employed by Field as "tool-driven". It is assumed in this integration style that the tools are semi or fully autonomous, and that, although the user is in control of the general behaviour of the environment, the tool interaction makes assumptions about the support that the user requires and therefore tools can react autonomously and work in concert. As the tools, then, are instigating certain actions, the initiators of such actions necessarily need to be informed of the result, in order to determine their next courses of action.

The decision of the designers of SoftBench's EDL language to omit a synchronous messaging facility can be understood in the light of the user-driven integration style. In such a style, most operations performed by the environment are initiated by and visible to the user. When something unintended occurs, as in the case of the scenario above, either the user caused that occurrence, or the user is aware of the problem and can therefore react to it.

6. Summary, conclusions and future work

An approach to the precise description of tool integration devices in tool integration frameworks has been described. This approach employs a layered model to describe these devices in a way which can cater to the differing information needs of a range of people who may have an interest in the semantics of the features provided by a tool integration framework for describing styles of inter-tool communication and who may wish to gain an appreciation for the various styles of such interaction which may be supported conveniently by a particular framework. The model has been illustrated by presenting representative aspects of the descriptions of inter-tool communication in two frameworks: SoftBench and Field. The tool integration devices of the two frameworks were then compared, at least insofar as this could be illustrated using those aspects presented in the separate descriptions.

The model presented in this paper allows the precise description of the tool integration devices provided by tool

[1] This is an illustrative message only, and does not reflect the format of a SoftBench message.

integration frameworks, and facilitates a kind of comparison of these devices which has not otherwise been possible to date. Although some aspects of the comparison may well have been clear to the assiduous from the user manuals and other documentation provided to potential users of the systems, the descriptions in terms of our model make possible a range of precise statements about the similarities and differences between the systems.

Plans for future work fall into three broad areas:

(1) Our immediate plans include the description of one more commercial framework: DEC's FUSE [19]. This is expected to yield some points of divergence from the two control-centred frameworks already described and should also give access to another community of users for the future work in (2) below.

(2) As much as one can judge from the literature available, it appears that the design of the integration devices provided in tool integration frameworks has not been influenced by an explicit analysis of what users of these devices (i.e., those carrying out the process of customising the framework to the needs of a particular organisation or project) want to do with them and how they would prefer to express the desired styles of integration. Based on experience in using comparative models of semantics in the design of programming languages [15], we propose to use the model described in this paper as the basis for the design of tool integration devices which allow the convenient expression of the styles of integration which are required in practice. This work will proceed by analysing which features of existing tool integration devices are actually used and how they are employed (whether, for example, they are frequently used to emulate some semantic pattern or protocol which is not provided by the tool integration framework as a primitive). These findings can then be related back to the semantic descriptions of existing devices, and new devices designed and described in terms of the model.

(3) Finally, we plan to be able to generate the inter-tool communication aspects of tool integration frameworks from the semantic descriptions written in terms of our model, in an analogous manner to that used for generating programming language implementations from similar descriptions [22,23]. This would, for example, enable the generation of an implementation of some proposed new set of devices designed as a result of the process described in (2) to be tested in practice, and then further refined.

Acknowledgments

The work described in this paper has been supported from a number of sources. These include Flinders University of South Australia, and the University of South Australia. Preliminary work on the development of the model of inter-tool communication was supported by the Centre de Recherche en Informatique de Nancy (CRIN), France. This work forms part of a collaborative software engineering research program involving the Department of Computer Science at Flinders University and the CSIRO-Macquarie University Joint Research Centre for Advanced Systems Engineering; funding for this work from the CSIRO Institute of Information Science and Engineering is gratefully acknowledged.

We particularly thank our colleagues in the Software Engineering Environments group at Flinders University, especially Bradley Schmerl, Michael Read and Michael McCarthy; they have offered useful advice and their discussions helped in the development of the paper. Thanks are due also to the anonymous referees for their comments.

Bibliography

[1] D.J. Barrett, L.A. Clarke, P.L. Tarr and A. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology,* Vol. 5, No. 4, 1996, pp. 378-421.

[2] A. Brown and P. Feiler. *An analysis technique for examining integration in a project support environment.* Technical Report No. CMU/SEI-92-TR-35, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1992.

[3] A. Brown, P.H. Feiler and K.C. Wallnau. *Understanding integration in a software development environment.* Technical Report No. CMU/SEI-91-TR-31, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.

[4] A. Brown and M. Penedo. "Integration" Working Group summary: SETA2. *ACM Ada Letters,* Vol. XIV, 1994, pp. 85-92.

[5] A.W. Brown. An examination of the current state of IPSE technology. in *Proc. 15th Int. Conf. Software Engineering*, Baltimore, Maryland, 1993, pp. 338-347.

[6] M. Cagan. HP SoftBench: An architecture for a new generation of software tools. *Hewlett-Packard Journal,* Vol. 41, No. 3, 1990, pp. 36-47.

[7] D.H. Freidel. *Modelling communication and synchronisation in parallel programming languages,* Dept. Computer Science, University of Iowa, Iowa City, Iowa, Ph.D. Thesis, Technical Report No. 84-01, 1984.

[8] B. Fromme. HP Encapsulator: bridging the generation gap. *Hewlett-Packard Journal,* Vol. 41, No. 3, 1989, pp. 59-68.

[9] B. Fromme and J. Walker. An open architecture for tool and process integration. in *Proc. Int. Conf. on Software Engineering Environments*, Reading, U.K., 1993, IEEE Computer Society Press, Los Alamitos, California, pp. 50-62.

[10] D. Garlan and E. Ilias. Low-cost adaptable tool integration policies for integrated environments. in *ACM SIGSOFT'90: Fourth Symposium on Software Development Environments*, Irvine, California., ACM SIGSOFT Software Engineering Notes, 15,6, Dec., 1990, pp.1-10.

[11] J.G. Harvey. *Software Engineering Environments: classifications and models.* Technical Report No. CS-93-002, School of Computer and Information Science, University of South Australia, Adelaide, South Australia, 1993.

[12] J.G. Harvey and C.D. Marlin. *Describing Inter-Tool Communication in Tool Integration Frameworks.* Technical Report No. CS-95-012, School of Computer and Information Science, University of South Australia, Adelaide, South Australia, 1995. (*also* Dept. Computer Science, Flinders University of South Australia, Technical Report No. 96-01.)

[13] J.G. Harvey and C.D. Marlin. Towards a formal description of tool integration frameworks. *Australian Computer Science Communications,* Vol. 17, No. 1, 1995, pp. 199-207.

[14] E. Ilias. *Policies for tool integration in integrated programming environments,* Oregon Graduate Institute of Science and Technology, Oregon, Masters Thesis, Technical Report No. CR-90-04, 1990.

[15] C.D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation.* Lecture Notes in Computer Science 95, Springer-Verlag, 1980.

[16] C.D. Marlin. *A methodical approach to the design of programming languages.* Technical Report No. 83-05, Dept. Computer Science, University of Iowa, Iowa City, Iowa, 1983.

[17] C.D. Marlin and D.H. Freidel. *A model for communication in programming languages with buffered message passing.* Technical Report No. 83-09, University of Iowa, Iowa City, Iowa, 1983.

[18] C.D. Marlin and D.H. Freidel. Comparing communication in two languages employing buffered message-passing. *Journal of Systems and Software,* Vol. 12, No. 2, 1990, pp. 87-105.

[19] K. Michaels. Defining an architecture for control integration. in *SEE'93: Proc. 6th Conf. Software Engineering Environments*, Reading, U.K., 1993, IEEE Computer Society Press, Los Alamitos, California, pp. 63-71.

[20] B. Nejmeh. *Characteristics of Integrable Software Tools.* Technical Report No. INTEG_S/W_TOOLS-89036-N, Version 1.0, Software Productivity Consortium, Herndon, Virginia, 1989.

[21] M.J. Oudshoorn. *ATLANTIS: A tool for language definition and interpreter synthesis*, Dept. Computer Science, University of Adelaide, Adelaide, South Australia, Ph.D. Thesis, Technical Report No. TR 92-04, 1992.

[22] M.J. Oudshoorn and C.D. Marlin. Language definition and implementation. *Australian Computer Science Communications,* Vol. 11, No. 1, 1989, pp. 26-36.

[23] M.J. Oudshoorn and C.D. Marlin. Interpretive language implementation from a layered operational model. in *Proc. 5th International Conference on Computing and Information*, Sudbury, Ontario, Canada, 1993.

[24] M.J. Oudshoorn, K.J. Ransom and C.D. Marlin. Generating an implementation of a parallel programming language from a formal semantic definition. *Australian Computer Science Communications,* Vol. 14, 1992, pp. 641-654.

[25] G.D. Plotkin. *A structural approach to operational semantics.* Technical Report No. 085/091, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[26] S. Reiss. *FIELD: A Friendly Integrated Environment for Learning and Development.* Kluwer Academic Press, 1994.

[27] S.P. Reiss. *Integration mechanisms in the Field environment.* Technical Report No. CS-88-18, Computer Science Department, Brown University, Providence, Rhode Island, 1988.

[28] S.P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software,* Vol. 7, No. 4, 1990, pp. 57-66.

[29] S.P. Reiss. Interacting with the Field environment. *Software - Practice and Experience,* Vol. 20, No. S1, 1990, pp. 89-115.

[30] K.J. Sullivan and D. Notkin. Reconciling environment integration and component independence. in *ACM SIGSOFT'90: Fourth Symposium on Software Development Environments*, Irvine, California., 1990, ACM SIGSOFT Software Engineering Notes, **15**, 6, pp. 22-23, pp. 22-33.

[31] A.I. Wasserman. Tool integration in Software Engineering Environments. in *Proc. Software Engineering Environments: An Int. Workshop on Environments*, Chinon, France, 1989, Springer-Verlag, pp. 138-149.

[32] P. Wegner. Data structure models for programming languages. in *Proc. Symposium on Data Structures in Programming Languages*, 1971, ACM SIGPLAN Notices, 6,2, Feb., 1971, pp. 1-54.