A Layered Operational Model for Describing Inter-tool Communication in Tool Integration Frameworks

Jennifer G. Harvey†*
jennifer.harvey@unisa.edu.au

Chris D. Marlin*
marlin@cs.flinders.edu.au

* Department of Computer Science, Flinders University of South Australia, Adelaide, Australia
† School of Computer and Information Science, University of South Australia, Adelaide, Australia

Abstract

Integration frameworks for building software engineering environments provide at least data, control and presentation integration facilities, together with integration devices which afford access to these facilities by the tools which populate the framework. Typically, an integration device is a specially developed language, or extension to an existing language, in which the integration programmer specifies the desired interactions between the tools comprising the software engineering environment. Surprisingly little effort has been applied to assessing the expressiveness of integration languages, even though the power of such a language limits the level of integration a tool can achieve within the environment. Our work seeks to provide an approach to both assessing and comparing the expressiveness of the integration devices of a range of commercial and research products. This paper presents a layered operational model, based on information structures; this model has been developed for describing the semantics of the inter-tool communication features of integration devices in a precise manner, and in a manner which will facilitate such assessment and comparison.

1. Introduction

Tool integration frameworks offer a reusable facility for the integration of software engineering tools, by providing 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. The expressiveness of an integration device limits the level of integration a tool can achieve within the environment. Nevertheless, we find that, although much work has been done defining and characterising both integration and these integration devices (e.g. [1-4,17]), there is little work which seeks to assess the power 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 by both 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 seeks to provide one approach to assessing and comparing the expressiveness of integration devices. The motivation for this work is described in more detail in [11]. This paper presents a model developed for describing the semantics of the inter-tool communication features of integration devices in a precise manner which will allow such assessment and comparison. It is a formal approach, yielding 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., the ECMA/NIST Reference Model for Tool Integration Frameworks [6] and others surveyed in [11]). The model is a layered information structure model, based on the work of Wegner [24] and Plotkin [22], and in the style of Marlin [13-16], Oudshoorn [18-21] and others (e.g. [7,15,16,18]). It defines a collection of objects, or information structures, which characterise those aspects of interest in an integration framework (for example, messages, tools and inter-tool relationships); the semantics of the integration device features are described in terms of manipulations on the contents of the objects using primitive and other, "higher-order" operations defined by the model.

By developing a model that consists of several layers, it is possible to have a single description that caters for the differing 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 [18] 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.

0-8186-7635-3/96 $05.00 © 1996 IEEE

The paper is organised as follows. Section 2 presents the model. Section 3 illustrates its use by describing and comparing corresponding features of the integration langauges of Field [23] and Hewlett-Packard's SoftBench [5] frameworks. Finally, some conclusions are drawn and future work is discussed in Section 4.

2. The information structure model for inter-tool communication

The various components of the model (information structures, primitive operations, higher-order operations, as well as a communication substrate) fuse into a layered model as illustrated in Figure 1. The horizontal lines indicate that each layer is defined in terms of the layer below. For example, the Communication Substrate layer is defined only in terms of the information structures defined in the lowest layer, whereas the Primitive Operations layer is defined in terms of both the Communication Substrate layer and the Information Structures layer.

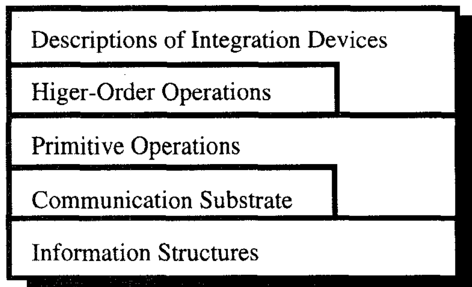

Figure 1. The five layers of the model.

The Information Structure layer describes the objects which represent the abstract domain in terms of state. The Communication Substrate layer defines the underlying elementary communication structure of the model. This layer describes only a tool's delivery and receipt of a communication message; the more advanced aspects of inter-tool communication are progressively specified in the higher layers.

The next two layers provide the operations that will be used in the uppermost layer, the Descriptions of Integration Devices layer, to describe the semantics of the integration devices of the integration frameworks under investigation. The Primitive Operations layer defines information structure manipulation operations, such as insertion and deletion, together with communication primitives, such as send and receive operators. The Higher-Order Operations layer utilises these primitives to define more complex operations, such as establishing a relationship between tools (as a communication binding). 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.

2.1. The Information Structures layer

The information structures model the relevant characteristics of the system under consideration. In this model, there are two information structures which are representative of the basic building blocks of an integrated environment – messages and tools. The `Message` information structure contains a number of attributes, as shown in Figure 2. The first three attributes are self explanatory. The attribute `messageMode` indicates the message communication mode – three message communication modes are defined by the model: Notification, Request and Reply (`"Not"`, `"Req"` and `"Repl"`). A Notification messsage is sent to inform interested tools of some event within the tool. A Request message specifies a service that the sending tool requires from the environment. A Reply message is sent in response to a Request messsage by a tool which has serviced that request, and indicates the success or otherwise of the requested service. The attribute `messageData` provides the data to be transmitted.

There is one `ToolCommunications` information structure for each tool in the integrated environment. It identifies the tool and designates legal inter-tool communication relationships; an example is depicted in Figure 3(a). Each `ToolCommunications` structure is identified by a `toolID` field. Each structure includes a list, `inputMsgs`, in which a tool publicises the notifications that it wishes to receive, and its services that it provides to the environment, so that those services are visible to other tools. This list thus defines the message interface of the tool. In addition, each `ToolCommunications` structure includes a list, `outputMsgs`, which contains the set of messages that can be emitted by the tool, including the notifications it will send and the requests for services provided by other tools in the environment. A tool publicises its available services, and the notification data it wishes to receive, by specifying pattern strings (in its `inputMsgs` list) which are used to match Request or Notification messages specified by other tools in their `outputMsgs` list. In this way, inter-tool relationships are created. For example, if *tool1* can emit a message *M*, and *tool2* has specified a pattern string *P* in its `inputMsgs`, which *M* matches, then a relationship is created between the message *M* and the pattern *P*, so that when *tool1* emits a message *M*, it is received by *tool2*.

To support the publication of services and notification data in the `ToolCommunications` structures, each entry of a tool's `inputMsgs` list is a structured entity containing

senderID	recipientID	messageID	messageMode	messageData

Figure 2. The *Message* information structure.

the attribute `pattern` for the published interface to the tool, and the attribute `patternMode` which indicates whether the pattern represents a Request ("`Req`") or a Notification ("`Not`") message in which the tool is interested. Note that Reply messages are special, being generated only in response to the receipt of a Request message, and therefore do not contribute to a tool's interface.

Likewise, each entry of the `outputMsgs` list of a `ToolCommunications` structure contains the attribute `msg` which contains the message data, and `msgMode` to indicate the message communication mode (once again, via the values "`Req`" or "`Not`"). Thus, Figure 3(b) extends Figure 3(a), by including the structure of `inputMsgs` and `outputMsgs`. The two attributes at the top of Figure 3(b), the binding lists, are discussed below.

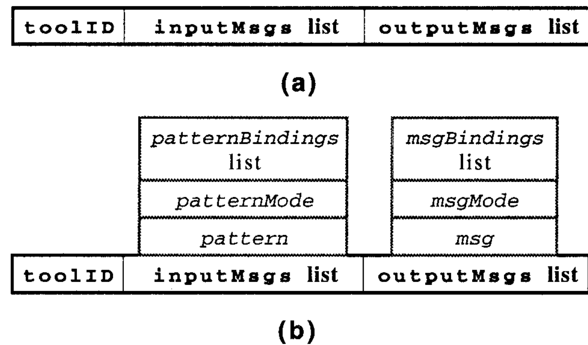

Figure 3. A *ToolCommunications* information structure.

It is apparent from the contents of the `ToolCommunications` structure that valid communication relationships between tools can be established by binding the entries of the `inputMsgs` list to matching entries in the `outputMsgs` lists of the tools. Indeed, the contents of the two lists can be determined statically, and this suggests that communication connections between tools can also be determined statically. However, while the range of output messages remains constant for the execution lifetime of the tool, the range of valid input messages for a tool varies during its execution lifetime; that is, a tool may wish to accept a certain message for a short period of its operation only. For example, an editor may initially accept requests to edit source documents for *projectX* and not for *projectY*. However, later the user may restrict editing to source documents of *projectY* in this case, the tool can no longer accept messages related to *projectX*. The set of communication bindings is therefore dynamic, as bindings are constructed upon an `inputMsgs` list entry becoming active and removed when an entry is no longer active. Figure 3(b) illustrates the binding lists for the input patterns and for the messages – these are reciprocal, such that where a pattern in *tool1* is bound to an output message in *tool2*, the same message in *tool2* will be bound to the pattern in *tool1*.

As an example, consider Figure 4. Three tool entries are shown, with sample active `inputMsgs` list entries and their `outputMsgs` list entries – the `inputMsgs` list of each tool is depicted between the first and second bold lines, and the `outputMsgs` list below the second bold line. Communication bindings are shown as connecting lines between matching `inputMsgs` and `outputMsgs` list entries, such as between `outputMsg 1` of tool `DEBUG` and `inputMsg 1` of tool `EDIT`. This is a snapshot of the communication bindings at one instant in time during the operation of the software engineering environment. The contents of any `inputMsgs` list can alter during the execution lifetime of the tool, and hence a different set of communication connections can be established.

2.2. The Communication Substrate layer

Tools represent the senders and recipients of information, and messages provide the flow of information through the environment. The purpose of the Communication Substrate layer of the model is to define a basic communication mechanism which is enacted when a message is emitted by one tool and received by one or more other tools. This mechanism is utilised in the Primitive Operations layer and the Higher-Order Operations layer to describe more complex inter-tool communication.

As a tool transmits a message, a `sendm` operation is executed, and the `receivem` operation is executed when a tool requests and waits for delivery of a message. Hence, a tool can be described in abstract terms as shown in Figure 5.

The guard in the `behaviour` section of the description specifies that a tool can be sending or receiving a message, or idle in terms of inter-tool communication.

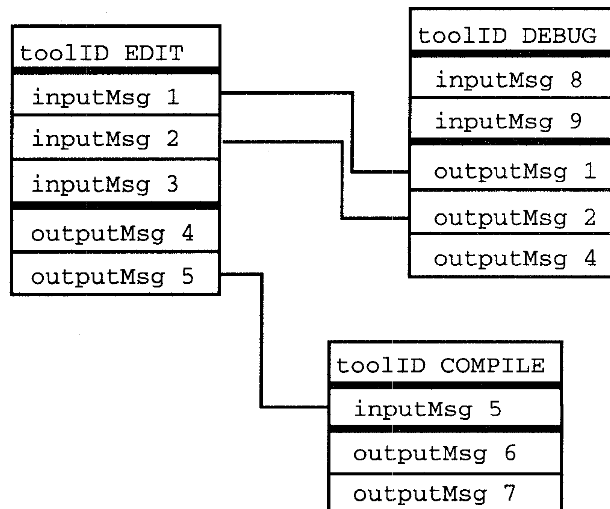

Figure 4. A snapshot of the communication bindings between *ToolCommunications* structures.

```
operation Tool is {
  out operation sendm(m : Message);
  in  operation receivem(m : Message);
  structure
     msgQ : unbounded buffer;
     bufferState : [open, closed] ← open;
  behaviour
     [ receivem(m)
     | sendm(m)
     | nil ];
}
```

Figure 5. An abstract description of a tool.

```
out operation sendm(m : Message) is {
  behaviour
    r : Tool ← I.recipient;
    r.msgQ::append(m);
}

in operation receivem(m : Message) is {
  behaviour
    when not msgQ::empty and
           bufferState == open then
        msgQ::removeHead(m);
    end when;
}
```

Figure 6. Descriptions of *sendm* and *receivem* operators.

The `structure` section describes two local variables. An unbounded buffer, `msgQ`, stores incoming messages which have yet to be delivered to the tool; the operations `append`, `removeHead`, `empty` and `length` are defined for the unbounded buffer type. The second variable, `bufferState`, determines whether messages can currently be removed from the queue and delivered to the tool. It is initialised to `open`, meaning that messages can be delivered.

The message transmission process is represented by the descriptions of the `sendm` and `receivem` operations shown in Figure 6. A `sendm` operation places the message in the recipient tool's message queue, and a `receivem` operation removes a message from the tool's message queue. In the descriptions, attributes of the information structures are selected with a dot, and operations are selected using a double colon, as in `r.msgQ::append(m)`, which applies the `append` operator of the `msgQ` field of recipient tool.

This layer also includes the operations `clearm` to selectively remove messages from a buffer, `blockm` to prevent receipt of messages, and `unblockm` to reopen a buffer. The description of these operations is trivial and omitted here for brevity.

The description of the operation `Tool`, and the descriptions of `sendm`, `receivem`, `clearm`, `blockm` and `unblockm`, form the Communication Substrate layer of the model. Note that the substrate does not define synchronous or asynchronous communication modes – while the operations such as `sendm` and `receivem` are constant across the descriptions of various tool integration frameworks, the semantics of the communication modes vary between frameworks and so are described at the higher levels of the model.

2.3. Primitive Operations layer

The model defines a set of primitives for manipulating the information structures, providing insertion, deletion, update and search operations, as shown in Table 1. The syntax of the primitives reflects the syntax style used for all primitives and higher-order operations in the model,

insertion	insert item in *infoStructure* where {*infoStructureField* ← *expr*, ...};	returns a pointer to the inserted item
deletion	remove [all] item[s] from *infoStructure* [where {*infoStructureField* == *expr*, ...}];	
update	change item in *infoStructure* where {*infoStructureField* == *expr*, ...} to {*infoStructureField* ← *expr*, ...};	returns a pointer to the updated item
search	find [all] item[s] in *infoStructure* where {*infoStructureField* == *expr*, ...};	returns a pointer to a list of located items, or NULL if the search is unsuccessful
match	*pattern_or_string* matches *string_or_pattern*;	returns boolean
iterate	for all [unique] *i* in *str* do *expr*; ... end for all;	

Table 1. Primitives for the manipulation of information structures.

send	send message to *toolID* where {*messageField* ← *expr*, ...};	
receive	receive message [from *senderID*];	returns message
remove outstanding messages	clear messages where {toolID == *T*};	
(un)block incoming messages	block messages to *toolID*; unblock messages to *toolID*;	

Table 2. Message primitives.

indicating the operation being invoked (e.g. find item), the information structure to which the operation is applied (e.g. *infoStructure*), and the parameters being transmitted (e.g. where {*infoStructureField* == *expr*, ...}). As an example, a typical primitive statement might be:

```
A ← find item in thisTool.inputMsgs where
        {pattern == P};
```

This statement locates an item in the ToolCommunications information structure for the tool, where the attribute pattern is equal to the value of the variable P, 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 ← insert item in thisTool.inputMsgs
        where {pattern ← P};
```

In this case, a new entry is inserted into the inputMsgs list of the ToolCommunications information structure for the tool, and the pattern attribute of this entry is given the value of the variable P.

A match operator and an iterator are also defined, as shown in Table 1. The matches operation determines the equivalence of a pattern which may contain wild-card entries, and a string, as in

```
C ← "EDIT * * *" matches
     "EDIT NOTIFICATION SAVE a.c";
```

(which returns True). The for all iterator is designed to iterate over the entries contained in an information structure or a subset thereof. The precise semantics are not described here; further details can be found in [9].

The Primitive Operations layer also defines communication primitives corresponding to operations in the Communication Substrate layer; these are shown in Table 2.

2.4. Higher-order operations layer

Associated with the ToolCommunications structure are two higher-order operations, described in Table 3. Higher-order operations are defined in terms of the primitives and information structures of the model – details are omitted here, but may be found in [11].

The operation, create msgInterface, inserts a new interface component (a message pattern) into a tool's inputMsgs list and creates the associated communication bindings. The second operation, remove msgInterface, is the inverse of create msgInterface, destroying the communication bindings for a specified interface component, and then removing that component.

create msgInterface where {toolID == *T*, pattern ← *P*, patternMode ← *M*};	Inserts an item into the inputMsgs list of the ToolCommunications structure for the tool *T*. If no such structure exists, one is created. Next, communication bindings are created in all cases where an entry in an outputMsgs list of a tool matches the newly created inputMsgs list entry.
remove msgInterface where {toolID == *T*, pattern == *P*, patternMode == *M*}; remove msgInterface where {*ptrToInterface*};	The first version of the syntax of this operation locates an entry within the inputMsgs list of the ToolCommunications structure for the tool *T*, where the pattern field has the value *P* and the patternMode field has the value *R*. The second version provides a pointer to such an entry. Both versions then remove all communication bindings to that inputMsgs list entry (effectively closing those communication paths) and, finally, remove the entry from the inputMsgs list.

Table 3. Higher-order operations for the *ToolCommunications* structure.

```
Notification_send →
        1   A ← find item in thisTool.outputMsgs where {      /* phase 1 */
        2     msg == NotificationData,
        3     msgMode == "Not"};
        4   for all I in A.msgBindings do                     /* phase 2 */
        5     send message to I.recipient where {
        6        senderID ← thisTool.toolID,
        7        messageMode ← "Not",
        8        messageData ← NotificationData};
        9   end for all.
```

Figure 7. Description of the Notification_send communication event in SoftBench's EDL integration language.

2.5. Descriptions of Integration Devices layer

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 inter-tool communication event corresponding to inter-tool communication language features in the integration devices. These descriptions 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(s) it describes. The descriptions are Pascal-like, using constructs of the language in conjunction with the model primitives and higher-order operations; a more complete description is provided in [10]. A Pascal-like syntax was chosen as the basis for the algorithmic descriptions because the set of programming language constructs required is small.

In addition, two variables are defined by the model. The first, `thisTool`, returns the name of the tool in which the language feature being described is executing, as in:

```
send message where {
     senderID ← thisTool, ....};
```

Similarly, the variable `lastMsgID`, returns the ID of the last message received by this tool, as in:

```
send message where {
     ..., messageID ← lastMsgID, ....};
```

3. Describing integration devices in tool integration frameworks

We present an example of the descriptions in the uppermost layer of the model by comparing the description of an inter-tool communication event in the Event Description Language (EDL) of Hewlett-Packard's SoftBench tool integration framework, and of the MSG Program Interface (MPI) of Field. 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.

Because of limited space, only one example of these features can be presented. For the purposes of illustration, the communication event (3) Notification_send is selected.

The SoftBench EDL statement

```
send_message(Notify, NotificationData);
```

is the language feature corresponding to the Notification_send communication event. Figure 7 shows the algorithmic description of this event. It consists of two phases. Phase 1 establishes the set of communication bindings that are associated with the Notification message by locating the entry in the tool's `outputMsgs` list that matches the message mode and the contents of the NotificationData field (the matching, therefore, is quite specific to this message). This occurs in lines 1 to 3. If there are no communication bindings associated with the

```
Notification_send →
  1   A ← find item in thisTool.outputMsgs where {     /* phase 1 */
  2      msg == NotificationData};
  3   if A =/= NULL then
  4      B ← find item in A.msgBindings where {        /* phase 2 */
  5         patternInd == "Pri"};
  6      if B == NULL then                              /* phase 3 */
  7         for all I in A.msgBindings do
  8            send message to I.recipient where {
  9               senderID ← thisTool,
 10               messageMode ← "Not",
 11               messageData ← NotificationData};
 12         end for all;
 13      end if;
 14   end if.
```

Figure 8. Description of the Notification_send communication event in Field's MPI integration language extension.

Notification message, no further action is taken. Phase 2, in lines 4 to 9, occurs when a list of communication bindings exist. Here, the message is generated and sent to each tool bound to this Notification message.

Field types its message patterns that form the integration interfaces as either Normal or Priority; a message emitted by a tool will not be delivered immediately to a recipient if there is any message pattern of type Priority currently bound to the message. To accommodate Normal and Priority message patterns in the model, a `patternInd` attribute is introduced into the `msgBindings` list of each tool to indicate whether the binding is to a Normal (`"Nor"`) binding or a Priority (`"Pri"`) binding.

The statement

```
MSGsend(NotificationData);
```

corresponds to the Notification_send event in Field's MPI. Figure 8 shows the algorithmic description of the Request_send communication event corresponding to this feature. It comprises three phases. Phase 1, in lines 1 to 2, establishes the set of communication bindings that are associated with the Notification message by locating the entry in the tool's `outputMsgs` list that matches the contents of the NotificationData field. If there are no communication bindings associated with the Notification message, no further action is taken is taken (line 3). Phase 2 searches for communication bindings of type Priority (`"Pri"`). This occurs in lines 4 and 5. If such bindings exist, no further action is shown in this description. Phase 3, lines 6 to 13, proceeds if no Priority communication bindings exist. Here, the Notification message is generated and sent to each tool bound to this Notification message.

The complete description of this feature also describes the semantics of the Notification_send communication event where Priority bindings are involved, but has been omitted here for simplicity and brevity.

When comparing the integration devices of tool integration frameworks, some aspects of the comparison are clear from informal descriptions of the integration capabilities and the language features, but it is only through the detailed examination made possible by an approach such as that described in this paper that more subtle details and differences are revealed. Field's Priority messages are described by Garlan [8], Ilias [12] and Reiss [23]. However, their precise semantics and the relationship between the Priority messages and message transmission is ambiguous. An examination of the description of the semantics of the Notification_send communication event in Figure 8 clarifies the latter ambiguity – a Notification message is transmitted only if a Priority binding for that message does not exist. Subsequent descriptions of the semantics of Priority messages elucidates the remaining ambiguities in the descriptions of Garlan, Ilias and Reiss.

Both integration languages permit some degree of dynamic determination of their integration interface and hence of the set of tools from which they will accept messages and to which they will send messages. Both languages 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 Reply). Field has less restrictive matching which ignores the message mode and thus allows one pattern to be used for both Request and Notification messages.

Our investigations into the expressiveness of integration languages have lead to the identification of various integration styles. In particular, Field exhibits a "tool-driven" style of integration, where tools are semi or fully autonomous, and make assumptions about the fine

grained processes employed by a user engaged in software development and hence about the support that the user requires. This style of inter-tool communication in Field is manifested in this example in the existence of Priority messages. Although this example does not expand on the behaviour of Priority messages, but it does indicate the advantages of a feature which allows the integration designer or specifier to manipulate any message emitted by a tool before it is delivered to its destination. Such a feature can be used to examine emitted messages before delivery, to take actions depending on the message type and/or to replace the emitted message with one or more alternative messages; in this way, it can be used to ease the tool integration task. Some examples of possible use include

- where the priority processing provides a simple examination of an emitted message, various data gathered from the examination can be forwarded to a metrics tool;
- where the priority processing takes some actions depending on the message type, this processing can be used to invoke the recipient tool if it is not currently part of the environment;
- where the priority processing replaces the emitted message, it can be used to provide a translation facility where the interface of a tool being introduced into the environment does not conform with the interface of another tool with which it needs to communicate.

4. 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 five-layer model to describe these devices in a way which can cater to the different information needs of a range of people with an interest in the semantics of the features of tool integration languages, and who wish to consider the various styles of inter-tool communication which can be supported conveniently by a particular framework. The model has been illustrated by presenting an aspect of the descriptions of inter-tool communication in two frameworks: SoftBench and Field. The tool integration devices of the two frameworks were then compared 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 integration frameworks and facilitates a kind of comparison of these devices which has not been possible to date. The sample communication event descriptions in Figures 7 and 8 demonstrate the utility of such semantic descriptions, and illustrate the ease with which comparisons can be made between the semantics of the features of the integration devices of different tool integration frameworks. Firstly, the detailed descriptions have served to clarify ambiguities in the informal descriptions of Priority messages. Although the example is brief, it has also shown that Field's MPI integration language assigns types to the communication bindings between tools, and that these types influence the delivery of messages. This supports a "tool-driven" style of integration. The example presented is too brief to indicate whether SoftBench adopts the same integration style, but our work indicates that it evidences an alternative approach.

The broader research area of this work is the investigation of the ways in which increased expressiveness of a suitable kind can be incorporated into the tool integration devices of tool integration frameworks, and the identification of better integration styles among sets of tools. Our efforts so far have concentrated on the development of the model and the descriptions of the integration devices of Field and SoftBench. Our immediate plans include the description of another commercial product: the ENCASE integration language employed by DEC's COHESIONWorX tool integration framework. We suspect that this will provide an alternative style of inter-tool communication.

It appears that the design of integration devices in tool integration languages has been informal to a degree, and has not been influenced by an analysis of user requirements. Our work will proceed by analysing which features of existing tool integration devices are used and how they are employed. 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. We plan to generate the inter-tool communication aspects of tool integration frameworks from the semantic descriptions of the model, in a manner similar to that used for generation programming language implementations [18-21]. This will enable the generation of an implementation of the proposed new set of devices designed as a result of the user analysis, which can be tested in practice and further refined.

Acknowledgements

The ongoing work described in this report represents part of a long-term collaborative software engineering research programme 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 following sources is gratefully acknowledged: the CSIRO Institute of Information Science and Engineering, Flinders University's URB grant scheme, the Centre de Récherche en Informatique de Nancy (Nancy, France), the Institute for Computer Systems Engineering and Assurance, and the University of South Australia's Cathie Funds. The anonymous reviewers' comments halped to improve this paper.

References

[1] Brown, A. & Feiler, P. (1992) *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.

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

[3] Brown, A. & Penedo, M. (1994) "Integration" Working Group summary: SETA2. *ACM Ada Letters,* **XIV**, (Fall), pp. 85-92.

[4] Brown, A.W. (1993) An examination of the current state of IPSE technology. *Proc. 15th Int. Conf. Software Engineering*, Baltimore, Maryland (May 17-22), pp. 338-347.

[5] Cagan, M. (1990) HP SoftBench: An architecture for a new generation of software tools. *Hewlett-Packard Journal,* **41**, 3 (June), pp. 36-47.

[6] ECMA/NIST (1991) *Reference model for frameworks of software engineering environments.* Special Publication Report No. ECMA TR/55, 2nd Ed., European Computer Manufacturers Association, National Institute of Standards and Technology.

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

[8] Garlan, D. & Ilias, E. (1990) Low-cost adaptable tool integration policies for integrated environments. *ACM SIGSOFT'90: Fourth Symposium on Software Development Environments*, Irvine, California. (Dec. 3-5), ACM SIGSOFT Software Engineering Notes, **15**,6, pp.1-10.

[9] Harvey, J.G. (1995) A layered model for the description of tool interactions in integrated software engineering environments. *Proc. 2nd Annual JRCASE/Flinders University Collaborative Research Workshop on Software Engineering*, Macquarie University, North Ryde, Sydney, Australia.

[10] Harvey, J.G. & Marlin, C.D. (1995) *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 and Department of Computer Science, Flinders University of South Australia, Technical Report No. 96-01.

[11] Harvey, J.G. & Marlin, C.D. (1995) Towards a formal description of tool integration frameworks. *Australian Computer Science Communications,* **17**, 1 (Feb.), pp. 199-207.

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

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

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

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

[16] Marlin, C.D. & Freidel, D.H. (1990) Comparing communication in two languages employing buffered message-passing. *Journal of Systems and Software,* **12**, 2 (May), pp. 87-105.

[17] Nejmeh, B. (1989) *Characteristics of integrable software tools.* Technical Report No. INTEG_S/W_TOOLS-89036-N, Version 1.0, Software Productivity Consortium, Herndon, Virginia.

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

[19] Oudshoorn, M.J. & Marlin, C.D. (1989) Language definition and implementation. *Australian Computer Science Communications,* **11**, 1 , pp. 26-36.

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

[21] Oudshoorn, M.J., Ransom, K.J. & Marlin, C.D. (1992) Generating an implementation of a parallel programming language from a formal semantic definition. *Australian Computer Science Communications,* **14**, (No. 1, Part B), pp. 641-654.

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

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

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