Integrating, Customizing, and Extending Environments with a Message-Based Architecture

John E. Arnold

US Applied Research Laboratory
Bull HN Information Systems Inc.
300 Concord Road MA30-821A
Billerica, MA 01821

phone: 508-294-2356
fax: 508-294-4848
email: J.Arnold@ma30.bull.com

Steven S. Popovich[1]

Department of Computer Science
Columbia University
New York, NY 10027

CUCS-008-95

Abstract

Message-based architectures have typically been used for integrating an engineer's set of tools as in FIELD and SoftBench. This paper presents our experience using a message-based architecture to integrate complex, multi-user **environments**. Where this style of control integration has been effective for encapsulating independent tools within an environment, we show that these techniques are also useful for integrating environments themselves.

Our experience comes from our integration of two types of process-centered software development environments: a groupware application that implements a Fagan-style code inspection process and a software development process environment where code inspection is a single step in the overall process. We use a message-based mechanism to federate the two process engines such that the two process formalisms complement rather than compete with each other. Moreover, we see that the two process engines can provide some synergy when used in a single, integrated software process environment,

Specifically, the integrated environment uses the process modeling and enactment services of one process engine to customize and extend the code inspection process implemented in a different process engine. The customization and extension of the original collaborative application was accomplished without modifying the application. This was possible because the integration mechanism was designed for multi-user, distributed evironments and encouraged the use of an environment's services by other environments. The results of our study indicate that the message-based architecture originally conceived for tool-oriented control integration is equally well-suited for environment integration.

Keywords: Software Integration, Message-based Architecture, Process Modeling, Federated Systems, Process-Centered Environments

[1]The work reported in this paper was done while Popovich was at Bull HN Information Systems Inc.

1 Introduction

A wide variety of process-centered environments (PCEs) have been developed, based on a range of process modeling formalisms [10, 17, 15]. Many of these systems are based on a client/server architecture, with the process engine(s) in the server(s), but they differ in the degree of centralization, both of the process engine(s) and the process data. Peuschel and Wolf [11] present four architectures for PCEs, based on these distinctions. Figure 1 shows the general structure of each of the architectures, which we now summarize. We have invented names for the architectures; Peuschel and Wolf did not name them, but instead numbered them I through IV, in the same order in which we summarize them below. We also make a distinction between two subtypes of their architecture IV, creating a fifth architecture in our taxonomy.

Before we continue, we should note an important characteristic of PCEs that is not mentioned in the Peuschel and Wolf classification, or indeed in the taxonomy that we present below, for the simple reason that, except in the degenerate case of a single, centralized process engine, it is orthogonal to the environment architecture. That is the distinction between *homogeneous* environments, in which all of the engines share the same process modeling language (but will generally contain different processes written in that language), and *heterogeneous* environments, in which each engine has a different PML. It is important to remember that each of the distributed PCE architectures may be used with either homogeneous or heterogeneous process engines. Figure 2 illustrates the distinction between homogeneous and heterogeneous environments by showing an abstract view of each case. The homogeneous environment contains two instances of a single process engine, while the heterogeneous environment contains one instance each of two distinct process engines.

The *centralized* model is the simplest of the five, with both the process engine and the process data in a single server. Multiple process clients connect to the server, with each performing a particular role in the process. The server maintains the process objectbase, which is not directly accessed by the clients. The MARVEL [1] system exemplifies this model.

One straightforward extension of a PCE into a distributed environment is the *shared objectbase* model. In this model, there may be multiple process engines, each of which may have multiple clients attached to it. The engines share a single objectbase, however. All communication between process engines is through the objectbase. CLF [3] follows this model, with homogeneous process engines (but only one developer per engine). The ProcessWall [7] design follows this model also, with heterogeneous process engines.

The *hybrid objectbase* model is similar to the shared objectbase model in that there is a shared objectbase. There are also, however, local objectbases for each process engine. All communication is still through the shared objectbase, but data that is private to a single engine may be stored locally in order to improve performance. The Merlin [14] system uses this model, with homogeneous engines.

An alternative to the shared object model is the *message-based* architecture. In this model, there are multiple process engines, as in the shared objectbase model. Instead of a shared objectbase, however, there are local objectbases, one for each process engine. Sharing between the process engines is accomplished by communication between the engines. The distinction between the shared objectbase and message-based architectures is analogous to that between concurrent programming languages based on shared memory and on message-based communication.

We can further refine the message-based architecture of Peuschel and Wolf into two architectures, by considering whether the communication between process engines is bus-based or point-to-point. We call these two architectures the *bus* and *point-to-point* architectures. The Oz [2] PCE is based on the point-to-point model, with homogeneous process engines (which, however, enact *heterogeneous* processes). The system that we will present in this paper, ACME, is based on the bus model, with heterogeneous engines.

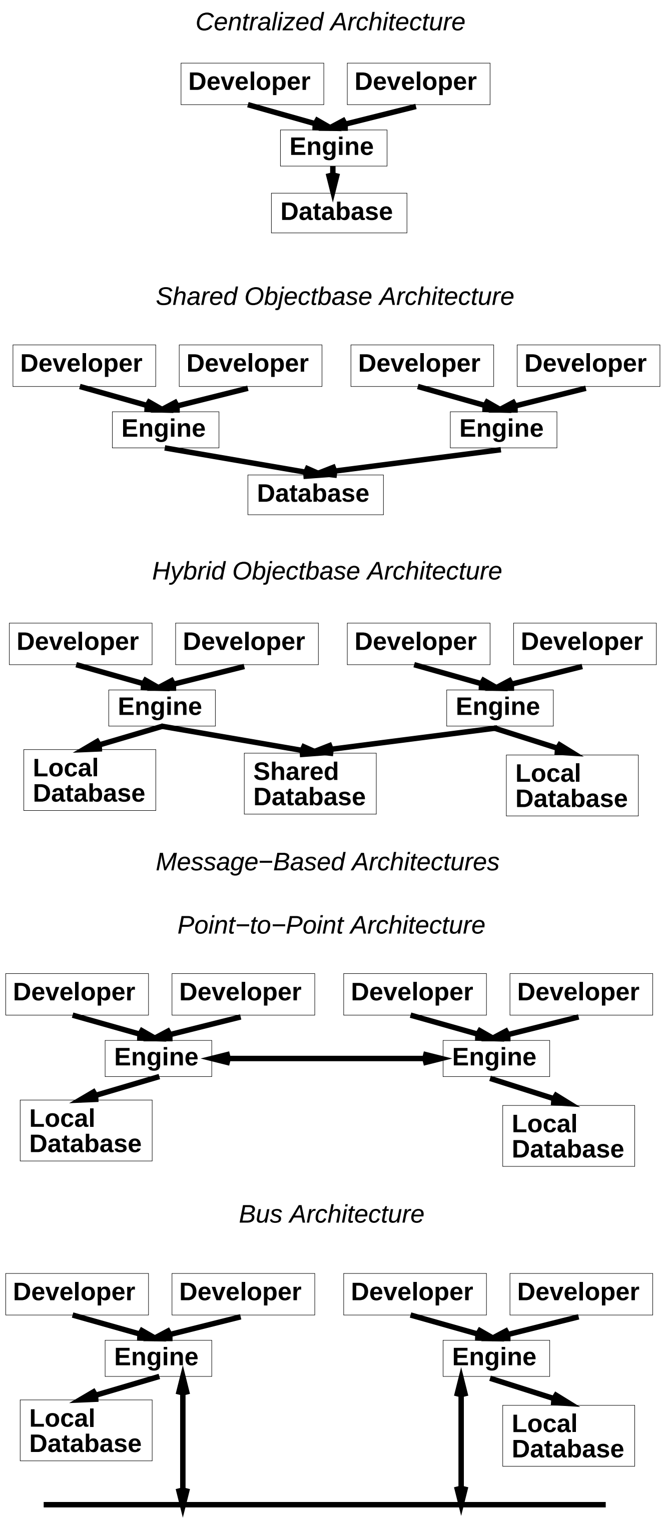

Figure 1: Four PCE Architectures

ACME is an integration of the MARVEL process engine with a collaborative PCE (for a Fagan-style [5] inspection process) based on ConversationBuilder [9] (CB). CB is a toolkit for developing computer-supported collaborative work (CSCW) environments. The CB environment architecture uses the centralized model, with a single server (per environment) controlling all interactions between users, as well as between each user and the

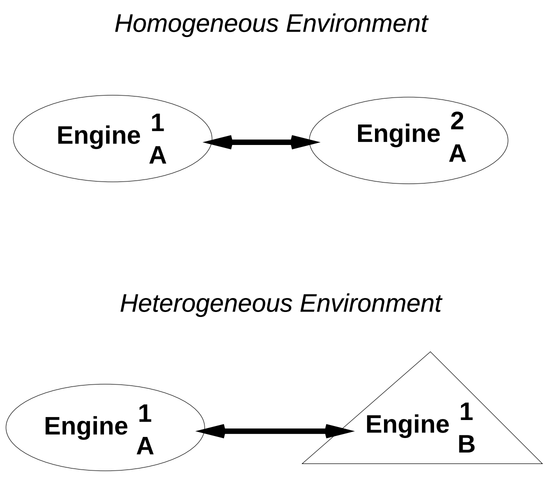

Figure 2: Homogeneous and Heterogeneous Environments

objectbase. The server contains a *collaboration engine*, which enacts a process that defines the allowable interactions in the environment. CB processes, or *protocols*, make use of a speech act model to define process activities as *utterances* that make transitions between the various process states.

Interaction between the CB engine and other environment components is by means of a message bus. Bus-based messaging architectures have become increasingly common in PCEs since their appearance in systems like Field [13] and SoftBench [18]. Originally intended for use as a tool interconnection mechanism in "toaster model" [4] systems, they have found use in a more general fashion as an interconnection mechanism for various types of environment components.

This use of the message bus means that, although CB is centralized by our definition, it is also readily extensible into a message-based environment by integrating (as we have done) an additional process server. This allows the opportunity for synergy between the process engines. The resulting environment can take advantage of the strong points of the two engines, whose process formalisms complement one another: CB offers specialized features for the definition and enactment of processes between people as needed for CSCW applications; Marvel offers specialized features for the definition and enactment of software development processes including the semantics of interactions between programmers and their tools.

We recognized that these two environments did not compete as much as they served as complements. We found that CB could add a recognized need for better collaborative support in MARVEL. Since CB's protocol formalism is very well suited for expressing interactions between people, we wanted to capitalize on these abilities. On the other hand, we wanted the completed application environment to accommodate the rule-based process model that we viewed as more flexible and to include support for process evolution: traits we are accustomed to due to our experience with MARVEL [8].

We did not attempt to add the rule-based process support and process evolution capabilities to CB, however, because we found the challenge of modifying the existing CB and/or the CB application to be quite daunting. To successfully add these capabilities would require high levels of expertise in the applications domain's process, CB itself, and CLOS (Common Lisp Object System - the language in which CB protocols are written).

Thus, we decided to **integrate** the two environments rather than modifying either single environment to add the "missing" features. From this observation, the primary goal of ACME was defined: to explore ways in which an existing process-oriented application could be enhanced and extended with no or minimal changes to the existing application by integrating another process engine via a message bus-style architecture.

This paper will describe how we took advantage of the message-based CB architecture to facilitate the integration of the MARVEL process engine with the CB conversation engine, and how this integration allowed us to implement rule-based agent technology that enabled synergistic interaction between the two process servers.

We begin with a brief overview of the Scrutiny^{TM} [6][2] environment, the existing CB PCE with which the MARVEL engine was integrated, and present background sketches of both CB and the MARVEL process engine. We then describe our ACME environment and show, via an example, how the message-based architecture facilitated the interactions between the process engines.

2 The Scrutiny Environment

Scrutiny is a groupware application for performing Fagan-style code and document inspections over a network. It was developed by the Bull US Applied Research Laboratory as a ConversationBuilder process. In a Scrutiny inspection, there are a number of inspectors and a single moderator, who controls the progress of the inspection[3].

The Scrutiny process model, summarized in Figure 2, begins with an *initiation* phase in which the document to be inspected is loaded and the participants are selected and notified of the inspection. When the chosen document is electronically distributed to each participant, the *preparation* stage begins. During the preparation stage, inspectors may navigate through the document at their own pace and create annotations to make comments, ask questions, or flag possible defects in the document. Annotations are attached to particular lines or regions in the document. During a normal inspection's preparation stage, only the author of the annotation can see the annotation. By modifying some parameters to the inspection protocol, the moderator can change the protocol to support a less strict document *review* in which participants can see each other's annotations during both preparation and resolution stages.

When preparation has been completed, the Scrutiny process enters the *resolution* stage: the synchronous "meeting" part of the inspection. During resolution stage, the moderator moves the focus of the inspection through the document. Moving the focus causes each participant's view of the document to be focused on the newly-selected area. The participants then read each other's annotations (which are now visible to all partipants regardless of the visibility policy chosen during preparation stage), make further comments, and reach consensus on the identification of defects in the current focus.

Following the resolution stage, the process enters *completion* stage during which reports can be generated, final defect classification is performed, and final metrics for the inspection are collected.

2.1 Extensions to the Scrutiny Process

As our group used the existing Scrutiny application, we discovered a number of user customizations that could be useful but were not explicitly provided. We also recognized that there were opportunities for injecting some process steps into the existing Scrutiny activities. We wanted to experiment with and add these behaviors to Scrutiny without modifying the Scrutiny code, since we did not want to interfere with the on-going development and enhancement of Scrutiny.

The **user customizations** we envisioned were related to attempts to improve the efficiency of reading annotations that have been made on a document being reviewed. For instance, the annotations were visible as

[2]Scrutiny is a trademark of Bull HN Information Systems Inc.

[3]This description of Scrutiny is based on Scrutiny v1.0 as described in [6]. For a more up-to-date view of Scrutiny including some new roles and features, see [16].

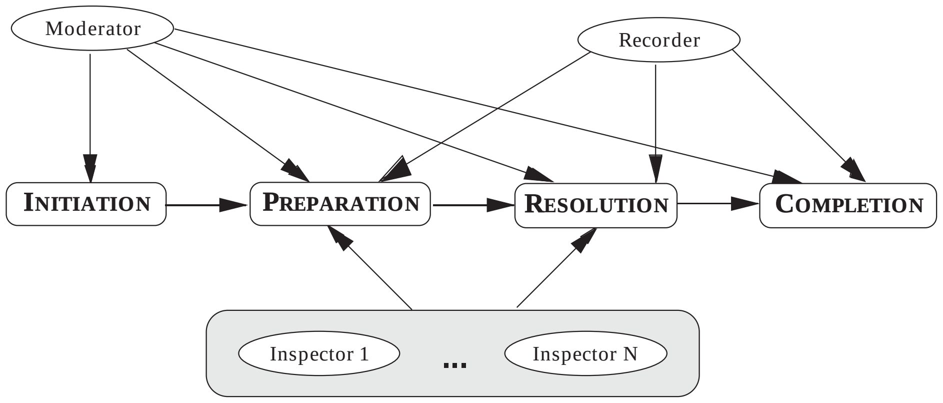

Figure 3: Summary of Scrutiny Process Roles and Phases

hyperlinks on the document being inspected or as summary lines in a window's scrollable list but one needed to then select the hyperlink or the summary line in order to read the text of the annotation. A similar mechanism was used to read defect records that have been created for regions in the document.

The motivation for these customizations came from our experience with these existing interface mechanisms. We found that it was inefficient to navigate long lists of annotations and defect records since the hyperlink pop-up menu or the scrollable list of annotation summaries must be read and the item(s) of interest must be selected. Also, as new annotations are created and added to the menu and scrollable lists, it is increasingly difficult to detect the presence of new (and, therefore, unread) annotations as the number of annotations increases.

We envisioned a customization which would allow each user to select any or all of a number of auto-display options:

- Automatic display of new annotations as they are created,
- Automatic display of new defect records as they are created,
- Automatic display of all annotations in a region of the document when the moderator moves the current focus to that region, and
- Automatic display of all defect records in a region of the document when the moderator moves the current focus to that region.

These are all implemented as part of ACME. A better customization that is being planned is the automatic display of only **unread** annotations in a region when the moderator moves the current focus to that region.

In terms of **injecting new process steps**, we found that we wanted to be able to customize Scrutiny for particular types of inspections. An inspection of a C program was not quite the same as an inspection of a design document, which was not quite the same as an inspection of a technical paper being prepared for submission. An example is that, for C programs, we might want to run a static analyzer such as Lint over the program in order to point out possible problem areas. Similarly, we might want to run an automated spelling checker over the English text of a document to check for typographical errors. Customizing an inspection type to include either of these static analyses amounts to inserting an extra step in the process. Adding these process steps captures the results of these tools and automatically creates annotations containing the possible error (coding or misspelling) and attaches the annotation to the appropriate line in the document. Both of these agents have been implemented in the current ACME.

2.2 Realizing the Extensions to ACME

We turned to MARVEL to address the extensions described in Section 2.1. Specifically, we wanted to customize the Scrutiny process definition without the need to edit the CLOS code that defines it. We considered the possibility of rewriting the entire process in the MARVEL Strategy Language (MSL) and *replacing* the Scrutiny engine with MARVEL, rather than *augmenting* it as we did, but for several reasons, we built ACME as a federated system, using both process engines. The primary reasons for using a federation approach are the following:

- The Scrutiny process was still evolving, and we did not want our process customizations to prevent the user's access to the new Scrutiny functionality.
- Since the Scrutiny protocol is approximately 20,000 lines of code (and, as mentioned above, this body of code was changing), it would be difficult to determine how to best implement the customizations directly in the protocol without overly interfering with the mainstream Scrutiny development effort. We were able to provide all of the extensions mentioned in Section 2.1 (e.g., all of the display customizations and the automated lint and spell agents) in 1000 lines of code which represents about 40 rules and about 15 object classes.
- Processes written in (rule-based) MSL are more easily modified than CB processes written in CLOS code, and MARVEL provides support for process evolution, while CB does not specifically support it.
- We wanted to preserve the use of CB's model of human-to-human interaction. This would have been difficult to handle in MSL rules.

3 Background Information

3.1 ConversationBuilder and the Message Bus

ConversationBuilder is a process-based system for writing CSCW applications, where the process is envisioned as a "conversation" between two or more participants. A CB application is written essentially as a finite state machine in which the transitions are process steps. When running the process, users choose process steps from menus that contain all of the currently valid steps that they may execute. In addition to changing the objectbase, these steps may also take actions that affect the global state of the process and the actions available to other users. Thus, a CB process describes a scenario for cooperative interaction among a set of users.

CB is built using a message-based architecture. The process engine and the user interface components all attach to a central message server through which all communications are sent. Thus, in CB, the message bus serves as a general interconnection mechanism for all system components. In ACME, then, we have naturally made use of the message bus to handle all communications between the two process engines. In fact, a clear benefit of CB's use of a message-based architecture is that it makes ACME-style integration not only possible, but quite powerful and simple.

3.2 MARVEL Process Engine

MARVEL is a PCE kernel built around a rule-based process engine. The kernel is configured for a particular environment by loading in a data schema describing both the process and product data, the set of rules describing the behavior of the environment, and a coordination policy defining the actions to be taken in case of concurrent access conflicts in terms of a control rule model. A practical environment will also contain a set of tool envelopes, each of which is a shell script (augmented with parameter passing syntax) providing an invocation mechanism for a tool referenced by the process model.

Each rule in a MARVEL process either encapsulates a particular tool application with a set of conditions and effects on the objectbase, or expresses an inference that can be drawn from the satisfaction of a specified set of conditions on the objectbase. The rule-based process model inherently facilitates extending and evolving the process over time, because new rules can be added at any time, without necessarily having to change other parts of the process.

4 The ACME Environment

In the ACME environment, we used two (heterogeneous) process engines — Scrutiny (the CB application), and a MARVEL server containing the ACME process. These environments, it should be made clear, do *not* cooperate as peers; instead, the MARVEL engine monitors, and models the actions of, the CB engine. The MARVEL engine models an *external* view of the Scrutiny process, since it has access only to the *messages* sent by Scrutiny, and not to its internal state.

Using this model, it is a simple matter to extend the Scrutiny (CB) process by adding to the ACME (MARVEL) process model, but the converse is *not* equally straightforward. This is an appropriate model of interaction for two process engines when one's process is a *subprocess* of the other, for example when an inspection process (*e.g.*, Scrutiny) is to be integrated as an element of a larger software development process. Integrating the Scrutiny inspection process inside of a broader software engineering process such as C/Marvel [12] is a natural and simple extension of this integration and will be supported in the next version of ACME.

The ACME environment contains two programs that handle the communication between Scrutiny and MARVEL. *Harbinger* is a MARVEL client that connects to Scrutiny's message bus **and** to the Marvel process engine. Unlike other MARVEL clients, which act only as user interfaces for MARVEL, Harbinger processes MARVEL commands that are requested through either the user interface *or* that appear on the message bus. Harbinger receives requests to fire particular rules, which are generated by *Monitor*, a mediator program that watches the CB message bus for messages of interest. This architecture is shown in Figure 4. In the implementation, Harbinger is actually connected to the message bus, and message passing between Monitor and Harbinger is done over the bus. This is only an implementation convenience, though; there is no logical connection, since as currently implemented, only Monitor sends commands to Harbinger. We could equally well have connected Monitor and Harbinger with RPC, ToolTalk, or some other mechanism, but made use of the message bus "because it was there".

In ACME, communication *from* Scrutiny *to* MARVEL is accomplished by means of Monitor and Harbinger; this allows MARVEL to listen to the CB message bus, and by so doing, *model* the Scrutiny environment. In order to pass messages *back* to Scrutiny and change its behavior, we use a second communication mechanism. We make use of MARVEL's *envelope* facility, and a message sending program (already provided by CB) to implement this return path with no source code modifications. The return path allows rules that have been fired in MARVEL based on the effects of an action on this model to then "complete the circle" and have effects of their own in Scrutiny.

The link back to Scrutiny is provided by envelopes, which build messages from the arguments passed to them by MARVEL and use the `mbsend` program to send the messages over the message bus. The `mbsend` program, which sends the contents of a file to the message bus, was already provided by CB as a debugging aid, so we did not need to construct any new low-level programs. In fact, we were able to treat Scrutiny completely as a "black box", looking only at the messages appearing at the message bus and affecting Scrutiny only by sending messages. It was perhaps more difficult for us to do this than it would have been to add explicit ACME messages at the appropriate places in the Scrutiny protocol, and it did limit the amount of automation we could provide, since we could use only the information "published" in messages and could perform only actions that were already modeled in Scrutiny as messages. However, since Scrutiny was being extended and maintained by its builders while ACME was being built, the "black box" model simplified our situation considerably by making it

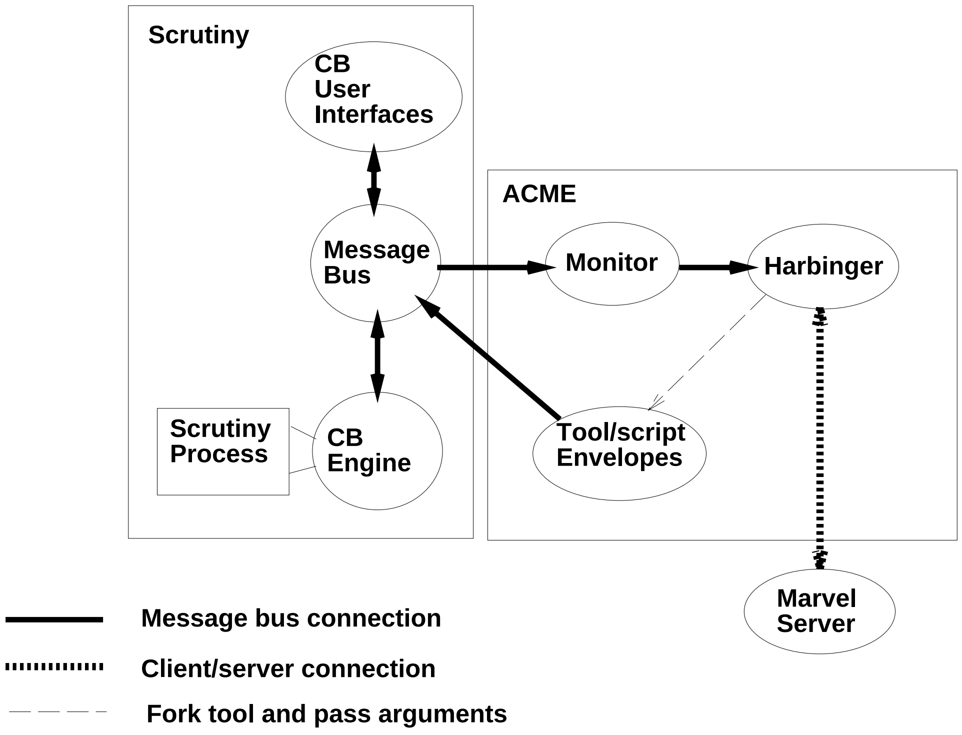

Figure 4: ACME Architecture

possible for us to use subsequent releases of Scrutiny without needing to maintain a set of customizations and re-apply them to each successive version.[4]

Harbinger, the message-bus-aware MARVEL client, is a slight modification of the ordinary MARVEL XView client — only about 100 lines of C code were added. Monitor, the message bus filter that selects messages to be passed to Harbinger, consists of approximately 3200 lines of C, lex and Yacc code. With these programs, we were able to reflect almost all user-visible actions in Scrutiny into MARVEL.

The ACME environment also contains *process* data that provide a MARVEL model of Scrutiny. The MARVEL data model used in ACME is shown in Figure 5. ACME includes a simple model of users, with which the administrator can change the behavior of the Scrutiny environment for specific users by changing preference variables in their user objects. ACME also models sessions, which in addition to their own status information contain instances of the user's preference variables that may be changed to suit a particular inspection. This MARVEL data model and process fully supports the ability for the ACME customizations to be selected (or not selected) by individual users. It also, in principle, can be aware of other Scrutiny sessions when running on behalf of a particular session, though the current implementation does not yet take advantage of this.

The ACME process model consists of about 900 lines of MSL rules, in addition to about 100 lines of process data definitions. The exact mechanics of these rules are not germane to this paper, so we will not describe all of the rules in detail. However, to illustrate the synergy between environments that could be created by two-way communication between two process engines such as in ACME, we present in Section 5 two examples of how ACME uses a message-based architecture to federate the CB and MARVEL process engines to implement some of

[4]Currently, we are working on relaxing the "black box" model to a "gray box" model. The gray box approach uses a Scrutiny that has been enhanced to send 4 ACME-specific notifications related to the creation of new objects and to receive 2 ACME-specific messages that allow the creation of annotations or defect records at a higher level of abstraction than Scrutiny already provided.

the extensions mentioned in Section 2.1.

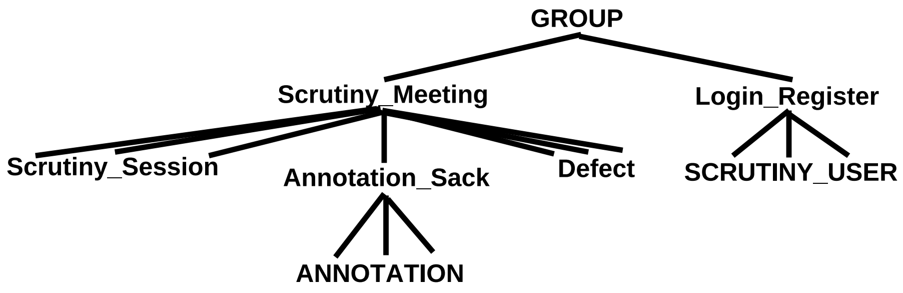

Figure 5: MARVEL Schema for ACME

The most severe limitation that we experienced in building ACME was our self-imposed restriction of treating Scrutiny as a "black box". While it was possible to construct a fairly detailed model of Scrutiny in MARVEL using the messages that were already being sent, Scrutiny was not designed to have requests for actions submitted to it by any means other than direct user interaction. Because of this, only those actions that can be performed directly from the Scrutiny user interface could be requested by MARVEL. This posed an obstacle to us when we added a facility for automatically adding annotations to a document by running a program, *e.g.*, by running Lint on a C file. While our envelope was able to extract all of the necessary information for creating annotations on each line on which a problem was reported, Scrutiny did not provide a message for creating a *filled-in* annotation. It provided only one for adding an *empty* annotation, which would need to be filled in by hand. It is very understandable that Scrutiny would do this, since it is intended as a groupware application where the participants in an inspection are all humans.

Given this design limitation, we had to mimic the human process of making an annotation even though the annotation creation is being undertaken by an automated agent. To accomplish this, we had to follow the three-step process that the human users are led through in order to create an annotation:

1. A start-annotation message is sent to the CB conversation engine. The result of this is that an annotation window is displayed.
2. The 'form' in the annotation window is filled in. Instead of people typing the information in, the ACME agent generates strings and files that represent the content of the annotation.
3. The information is acknowledged to be stored in the CB. Real users do this by pressing a button which send a store-annotation message. The ACME automated agents just send the store-annotation message which closes the window and stores the information in the CB object base.

Since the black box integration required us to follow this multi-step process, we had to implement a single, logical ACME action as an envelope which processed all three steps. Since the existing Scrutiny also was programmed to allow any given user to make only a **single** annotation at a time, this envelope required the addition of timing-related steps to insure that an agent doesn't attempt to make a second annotation until the creation of the first annotation has been completed.[5]

The message-based architecture of this style of integration (as seen in Figure 1) explicitly does not share objectbases between the two process engines. Thus, in ACME, the MARVEL objectbase and the CB objectbase remain distinct components. This style of architecture can present problems in terms of the effort that must be taken to insure that the two objectbases do not get their contents 'out of sync'. However, it also simplifies the problems of concurrent interaction between the two process engines since they never need to access the same data

[5]It was issues like these that motivated the gray box integration approach that we are currently pursuing.

objects.

The synchronization problem is especially important in terms of our ability to provide the ACME added-value to Scrutiny. In order to use the MARVEL rule mechanisms to write our process customizations, we needed to mirror the relevant objects of the Scrutiny process' CB data base in the MARVEL objectbase. This is due to a MARVEL restriction that allows rules to match and fire only on objects in the MARVEL objectbase.

Since we were writing rules to match on particular patterns of attributes and values for the various users, document, annotations, defect records, and the state of the inspection process itself, Monitor needs to watch all message bus activity that relates to these CB objects/states. The behavior at the core of the ACME-style integration can be summarized as follows:

1. A normal message is sent on the message bus in the normal course of running a Scrutiny inspection.

2. If the message has a message pattern that has been defined as relevant to some change that needs to be recognized in the MARVEL objectbase, the message is processed by Monitor:

   - Monitor interprets the message syntax to derive the semantics of the message (e.g., the CB u-store-object utterance in a certain context is syntax that indicates the creation of an annotation object).

   - If the message is one that has been defined as something that Harbinger wants to know about, the appropriate command message for Harbinger is created.

   - The command message is put on the message bus to be received by Harbinger. (Though not required, we use the same message bus as the normal Scrutiny session is using.)

3. Harbinger receives the message sent by Monitor. The MARVEL command embedded in the body of the message is executed by Harbinger (a MARVEL client with connections to the user interface **and** the CB message bus).

4. The MARVEL command that was executed typically creates a new object in the MARVEL objectbase or modifies an attribute of an object in the MARVEL objectbase to mirror a related object in the Scrutiny/CB objectbase. For instance, a Scrutiny message that created a new annotation would result in a MARVEL command to create the relevant annotation in the MARVEL objectbase. Similarly, a Scrutiny message that modified the annotation type from "question" to "defect" would result in a MARVEL command to modify the attribute in the related MARVEL object.

5. The creation or modification of objects in the MARVEL objectbase causes the MARVEL rule engine to match and (if possible) fire additional rules by forward chaining. The MARVEL commands of step 3 above implicitly cause the rules that implement the ACME extensions to Scrutiny to be fired. For instance, the creation of a new annotation in the MARVEL objectbase would cause the auto-display-on-create rules to match and be fired.

6. The firing of this rule would result in the activation of the envelope which would then use the relevant object/attributes in the MARVEL objectbase to construct a message that, when sent to the message bus, causes the appropriate behavior. For instance, the envelope triggered by the auto-display-on-create rule would use the new annotation object, its attributes, and the appropriate user object and its attributes to build a message that will cause the appropriate user's interface components to automatically open the annotation window that will display the relevant CB annotation object to the user.

5 ACME Examples

We will now show, by some simple examples, how the two disparate process engines in ACME interact synergistically. We will consider two related features -- a facility for automatically displaying new annotations as they are created (at the user's option), and another for displaying all annotations in a region whenever the moderator changes the group's focus to that region. These options are expressed as two attributes of the SCRUTINY_USER (for a user's default options) and Scrutiny_Session (for a user's options in a particular inspection) classes. The `anno_display_create` attribute, a boolean, specifies whether or not new annotations are to be displayed to that user, while the `anno_display_zone` attribute, a three-valued enumerated type, has options for no display of annotations in the "zone" under consideration, displaying each annotation in the "zone" in a separate window, and displaying only a list of annotations from which the user may choose one or more to view[6].

Figure 6 summarizes the communication and rule chaining that occur in our two examples.

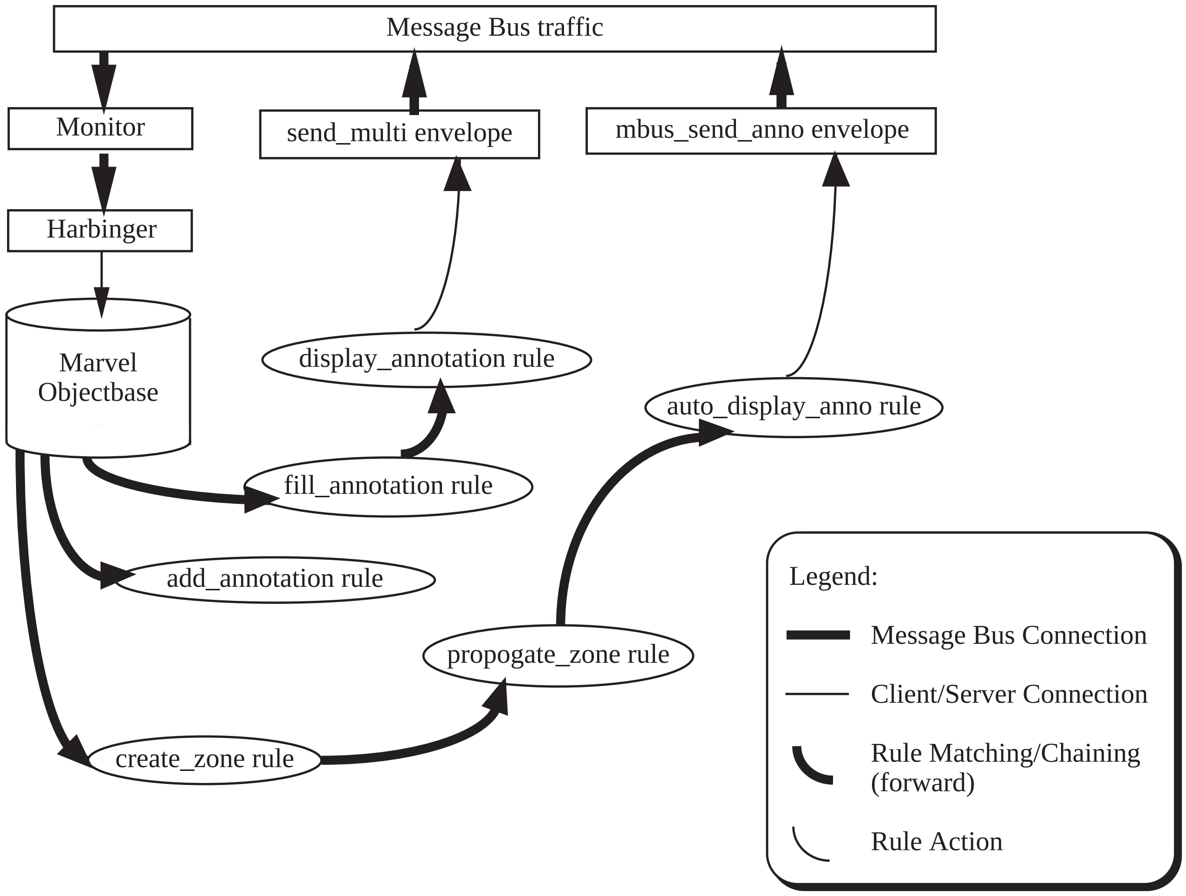

Figure 6: ACME Example Summary

5.1 Displaying New Annotations

The first customized behavior that we added to Scrutiny was to allow users to specify that they would like all newly-created annotations to be automatically displayed. This was accomplished by having Monitor watch for message traffic patterns that indicate the creation of a new annotation, taking appropriate action in Marvel, and affecting the user's display by creating relevant messages and sending them to the Scrutiny session's message bus. It starts when the annotation is created and Scrutiny sends the following message to update the list of annotations in each user's display.

[6]This last option is not yet implemented.

```
("update" "ws.stoops.ui.cb" (action "node177.header"
  :label "r 001  Re:      Line(s): 12 - 16    By: stoops
     Type: Question      Do we need to change name?")
```

Monitor parses this message to extract all of the relevant information: The name of the node (`node177`), the annotation ID (`001`), the range of line numbers in the document to which the annotation applies, the author of the annotation, its type, and its short title. Monitor derives the pathname to the file containing the full contents of the annotation from its node name, and then invokes two rules, in order, by sending messages to Harbinger. The first, `add_annotation`, creates a new annotation object, with the specified annotation ID as its name, to describe the newly created Scrutiny annotation. The second, `fill_annotation`, enters the rest of the information into the new annotation object.[7]

There is no chaining between these two rules. The reason why chaining cannot be used here is that `fill_annotation` needs to have all of the annotation content information passed to it as literal-valued parameters.

Chaining does occur from `fill_annotation` to the `display_annotation` rule, though. That rule finds all active sessions that have the `anno_display_create` option enabled. It then uses an envelope to send one message for each of those sessions, telling it to display the new annotation.

The envelope formats a `u-display-node` message, imitating the message that CB normally sends in order to display an annotation, for each one of the sessions. It then opens a connection to the message bus and sends those messages, resulting in the annotation being displayed on all the appropriate screens.

5.2 Displaying Existing Annotations

The action starts when the moderator moves the current focus, or "zone", within the document being inspected. To accomplish this, Scrutiny sends a message like this one over the message bus:

```
("CB" "server.cb" (U-store-zone :node "node51" :ps "ps43" :ss "ss24"
  :agent "stoops" :zone-start-line 2 :zone-end-line 2 :zone-pix-pos 46
  :zone-start-char 45 :zone-end-char 52))
```

Monitor recognizes this message as moving the focus, and sends a message to Harbinger in order to invoke the `create_zone` rule.

The precondition of this `create_zone` rule distinguishes it from another `create_zone` rule that applies to individual inspectors during the preparation stage (when annotations may not be visible to other inspectors). The postconditions record the start and end lines of the new zone in the inspection object which causes chaining to the `propagate_zone` rule.

The `propagate_zone` rule gets all of the participants' sessions in a particular inspection that has just had a new zone set, and copies the start and end lines from the local zone in the moderator's session to the global zone in the inspection. It then sets attributes in the sessions through which chaining will take place.

The chaining here is to the `auto_display_anno` rule for each session that has its `anno_display_zone` set to `display_each`. There is another rule, `no_auto_display_anno`, that handles sessions where `anno_display_zone` is set to `no_display`.

This rule invokes another message sending envelope that causes annotations to be displayed by Scrutiny. It

[7]This had to be done in a separate rule, because the version of MSL that was current at the time did not allow creating an object and referencing its attributes in the same rule. Since then, a facility has been added to the MARVEL process language to allow binding newly created objects to variable names. This would allow writing these two rules as one.

sends messages that display all of the annotations in the range of line numbers in the new zone.

6 Lessons Learned

The message-based nature of the Scrutiny environment allowed us to easily construct an environment with multiple communicating process engines (CB and MARVEL) operating synergistically, using a message-based model (a bus architecture subtype of Peuschel and Wolf's Architecture IV). The flexibility of MARVEL's rule and activity (envelope) languages allowed us to "complete the circle" -- that is, to have actions in Scrutiny result in rules being fired in MARVEL, which in turn result in further actions being taken in Scrutiny -- without requiring any actual changes to either the Scrutiny environment, which we treated as a "black box" throughout the ACME project, or the MARVEL process engine. The resulting ACME environment has the user interface and inter-user interaction (groupware) facilities of the original Scrutiny environment, while also making possible extensions to the relatively inflexible Scrutiny process through the MARVEL engine. Our approach demonstrates the possibility for synergy between several process engines communicating over a shared message bus.

We constructed an environment in which one process engine monitors, models and affects the actions of another process engine, which is unaware of the first engine, except for the commands that it sends on the message bus. Our methods should be equally applicable to homogeneous and heterogeneous environments, since nothing in the ACME architecture depends on the internals of either process engine. It would be less straightforward, but should be feasible, to use the ACME architecture in an environment based on the point-to-point communication model; the message-based nature of the environment is more essential than the actual message bus, although the message bus did make communication more convenient. For that matter, it should also be possible, in theory, to apply our methods to a shared or hybrid objectbase model environment with triggers to detect changes to the data, since the triggers could serve the same purpose as messages did for us.

The most direct opportunity for future work here is to continue to expand the extent to which Scrutiny is modeled, and to add new and more complex forms of automation through MARVEL.

An intriguing possibility, however, might be to explore the message bus framework, not as an interaction mechanism for process engines, but as a mechanism by which a process may be incrementally migrated to a different process engine supporting a different process language. As it stands now, we have only *augmented* Scrutiny with MARVEL; we have not *replaced* any part of it, but we have not foreclosed that possibility. For example, it might be feasible, in the end, to model Scrutiny to such an extent that its process engine is no longer necessary -- while we would still be using Scrutiny's user interface, all of the messages to the process engine would be handled by the Monitor/Harbinger framework, and would result in messages being sent back to the user interface from MARVEL rules. This would take many months, of course, but slowly, message after message could be migrated to the new process engine, leaving us eventually free to remove the old engine completely.

7 Acknowledgements

The authors thank Mike Gelman and Melanie Jutras for their work on the implementations of ACME as well as George Heineman for his current contribution. The ACME team would also like to thank the Scrutiny team at Bull HN Information Systems Inc., especially John Gintell, Roland McKenney, and Mike Houde, for providing the application that inspired ACME and for their help with ACME's development.

The authors also thank Gail Kaiser, George Heineman, Roland McKenney, Mike Houde and Melanie Jutras for their comments on earlier versions of this paper.

References

[1] Israel Z. Ben-Shaul, Gail E. Kaiser and George T. Heineman.
An Architecture for Multi-User Software Development Environments.
Computing Systems, The Journal of the USENIX Association 6(2):65-103, Spring, 1993.

[2] Israel Z. Ben-Shaul and Gail E. Kaiser.
A Paradigm for Decentralized Process Modeling and its Realization in the OZ Environment.
In *16th International Conference on Software Engineering*, pages 179-188. IEEE Computer Society Press, Sorrento, Italy, May, 1994.

[3] CLF Project.
CLF Manual,
USC Information Sciences Institute, 1988.

[4] Anthony Earl.
Principles of a Reference Model for Computer Aided Software Engineering Environments.
In Fred Long (editor), *Software Engineering Environments International Workshop on Environments*, pages 115-129. Springer-Verlag, Chinon, France, September, 1989.

[5] Michael E. Fagan.
Design and Code Inspections to Reduce Errors in Program Development.
IBM Systems Journal 15(3), 1976.

[6] John Gintell, John Arnold, Michael Houde, Jacek Kruszelnicki, Roland McKenney and Gerard Memmi.
Scrutiny^{TM}: A Collaborative Inspection and Review Systems.
In Ian Sommerville and Manfred Paul (editors), *4th European Software Engineering Conference*, pages 344-360. Springer-Verlag, Garmisch-Partenkirchen, Germany, September, 1993.

[7] Dennis Heimbigner.
The ProcessWall: A Process State Server Approach to Process Programming.
In Herbert Weber (editor), *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 159-168. Tyson's Corner VA, December, 1992.
Special issue of *Software Engineering Notes*, 17(5), December 1992.

[8] Gail E. Kaiser, Peter H. Feiler and Steven S. Popovich.
Intelligent Assistance for Software Development and Maintenance.
IEEE Software 5(3):40-49, May, 1988.

[9] Simon M. Kaplan, William J. Tolone, Alan M. Carroll, Douglas P. Bogia and Celsina Bignoli.
Supporting Collaborative Software Development with ConversationBuilder.
In Herbert Weber (editor), *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 11-20. Tyson's Corner VA, December, 1992.
Special issue of *Software Engineering Notes*, 17(5), December 1992.

[10] Takuya Katayama (editor).
6th International Software Process Workshop: Support for the Software Process.
IEEE Computer Society Press, Hakodate, Japan, 1990.

[11] Burkhard Peuschel and Stefan Wolf.
Architectural Support for Distributed Process Centered Software Development Environments.
In Wilhelm Schafer (editor), *8th International Software Process Workshop*. Wadern, Germany, March, 1993.
Position paper.

[12] Programming Systems Laboratory.
Marvel 3.1 Sample Environments.
Technical Report CUCS-010-93, Columbia University Department of Computer Science, March, 1993.

[13] Steven P. Reiss.
Connecting Tools Using Message Passing in the Field Environment.
IEEE Software 7(4):57-66, July, 1990.

[14] Wilhelm Schafer, Burkhard Peuschel and Stefan Wolf.
A Knowledge-based Software Development Environment Supporting Cooperative Work.
International Journal on Software Engineering & Knowledge Engineering 2(1):79-106, March, 1992.

[15] Wilhelm Schafer (editor).
8th International Software Process Workshop: State of the Practice in Process Technology.
IEEE Computer Society Press, Wadern, Germany, 1993.

[16] US Applied Research Laboratory.
Scrutiny User Manual.
Technical Report USARL/94-1, Bull HN Information Systems Inc., May, 1994.

[17] Ian Thomas (editor).
7th International Software Process Workshop: Communication and Coordination in the Software Process.
IEEE Computer Society Press, Yountville CA, 1991.

[18] Gary Thunquest.
Supporting Task Management & Process Automation in the SoftBench Development Environment.
In Ian Thomas (editor), *7th International Software Process Workshop: Communication and Coordination in the Software Process*, pages 133-135. IEEE Computer Society Press, Yountville CA, October, 1991.

Table of Contents

List of Figures