SoftBench Message Connector: Customizing Software Development Tool Interactions

Software developers using the SoftBench Framework can customize their tool interaction environments to meet their individual needs, in seconds, by pointing and clicking. Tool interaction branching and chaining are supported. No user training is required.

by Joseph J. Courant

SoftBench Message Connector is the user tool interaction facility of the SoftBench Framework, HP's open integration software framework. Message Connector allows users to connect any tool that supports SoftBench Framework messaging to any other tools that support SoftBench Framework messaging without having to understand the underlying messaging scheme. Users of the framework can easily customize their tool interaction environments to meet their individual needs, in literally seconds, by simply pointing and clicking.

People familiar with the term SoftBench may know it under one or both of its two identities. The term SoftBench usually refers to a software construction toolset.[1] The term SoftBench Framework refers to an open integration software framework often used to develop custom environments.[2] People familiar with SoftBench the toolset should know that underlying the toolset is the SoftBench Framework.

Message Connector can be used to establish connections between any SoftBench tools without understanding the underlying framework. The editor can be connected to the builder which can be connected to the mail facility and the debugger, and so on. Message Connector does not care what tools will be connected, as long as those tools have a SoftBench Framework message interface. The message interface is added by using the SoftBench Encapsulator,[3] which allows users to attach messages to the functions of most tools. Message Connector uses the message interface directly and without modification. To date, over seventy known software tools from a wide variety of companies have a SoftBench message interface. It is also estimated that a much larger number of unknown tools have a SoftBench message interface. Users of the SoftBench Framework can now treat tools as components of a personal work environment that is tailored specifically by them and only takes minutes to construct.

Tools as Components

What does it mean to treat tools as components? To treat a tool as a component means that the tool provides some functionality that is part of a larger task. It is unproductive to force tool users to interact with several individual tools to accomplish a single task, but no tool vendor is able to predict all of the possible ways in which a tool's functionality will be used. Using Message Connector, several tools can be connected together such that they interact with each other automatically. This automatic interaction allows the user to focus on the task at hand, not on the tools used to accomplish the task.

A simple but powerful example is detecting spelling errors in a document, text file, mail, or any other text created by a user. The task is to create text free of spelling errors. The tools involved are a text editor and a spell checker. In traditional tool use, the editor is used to create the text and then the spell checker is used to check the text. In simple notes or files the text is often not checked for errors because it requires interacting with another tool, which for simple text is not worth the effort. When treating tools as components the user simply edits and saves text and the spell checker checks the text automatically, only making its presence known when errors exist. Note that in traditional tool use there is one task but two required tool interactions. In the component use model, there is one task and one required tool interaction (see Fig. 1).

Using Message Connector, a user can establish that when the editor saves a file, the spell checker will then check that specific file. This is accomplished as follows:

1. Request that Message Connector create a new routine (routine is the name given to any WHEN/THEN tool interaction).
2. Select the WHEN: tool to trigger an action (editor).
3. Select the specific function of the WHEN: tool that will trigger the action (file saved).
4. Select the THEN: tool to respond to the action (spell checker).
5. Select the specific function that will respond (check file).
6. Change the WHEN: and THEN: file fields to specify that the file saved will be the file checked.
7. Save the routine (routines are persistent files allowing tool interactions to be retained and turned on and off as desired).
8. Enable the routine.

Now any time the editor saves a file, that file will automatically be spell checked. The focus of creating text free of spelling errors is now the editor alone. The spell checking is driven by editor events, not by the user.

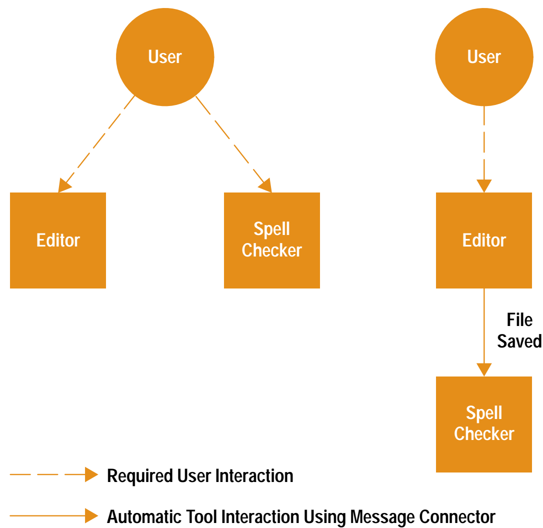

Fig. 1. Traditional tool use (left) compared with tools as task components (right). Using the SoftBench Message Connector, the user can set up a routine so that whenever a file is saved by the editor it is automatically spell checked. The spell checker does not have to be explicitly invoked by the user.

Tool Interaction Branching

While the above example is very simple, it applies equally well to any number of tool interactions. It is also possible to create branching of interaction based upon the success or failure of a specific tool to perform a specific function (see Fig. 2). For example, when the build tool creates a new executable program then display, load, and execute the new program within the debugger; when the build tool fails to create a new executable then go to the line in the editor where the failure occurred.

Interaction Chaining

It is possible to define interactions based upon a specific file type, and it is also possible to chain the interactions (see Fig. 3). As an example, when the editor saves a text file then spell check that file; when the editor saves a source code file then perform a complexity analysis upon that file; when a complexity analysis is performed on a file and there are no functions that exceed a given complexity threshold then build the file; when the complexity is too high, go to the function in the editor that exceeds the given complexity threshold; when the build tool creates a new executable

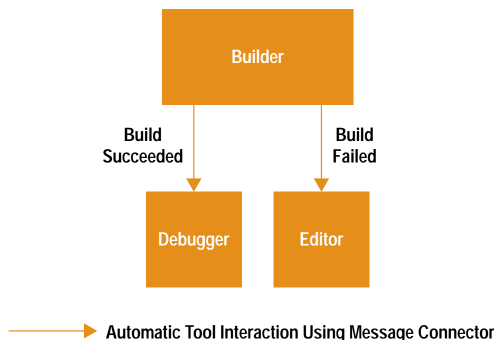

Fig. 2. Message Connector supports tool interaction branching. A different tool is invoked automatically depending on the result of a previous operation.

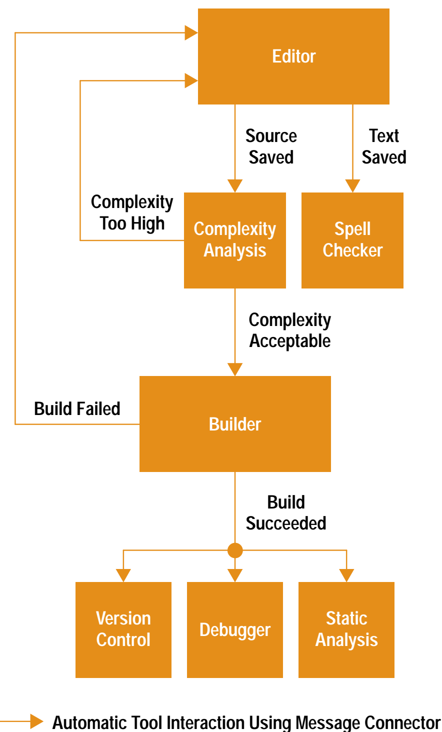

Fig. 3. Tool interaction chaining.

program then display, load, and execute the new program within the debugger, reload the new executable into the static analysis tool, and save a version of the source file; when the build tool fails to create a new executable, then go to the line in the editor where the failure occurred. This example of interaction chaining allows the user to focus on the task of creating defect-free text and source files. The user's focus is on the editor and all other tools required to verify error-free files are driven automatically by editor events, not by the user. The tools have become components of a user task.

Message Connector Architecture

The architecture of Message Connector follows the component model of use encouraged by Message Connector. As shown in Fig. 4, Message Connector is a set of three separate components. Each component is responsible for a separate function and works with the other components through the SoftBench Framework messaging system. The routine manager provides the ability to enable, disable, organize, and generally manage the routines. The routine editor's function is routine creation and editing. The routine engine's function is to activate and execute routines.

The importance of this architecture is that it allows Message Connector, the tool that allows other tools to be treated as components, to be treated as a set of components. This allows the user, for example, to request that the routine engine enable or disable another routine within a routine. It allows the user to run a set of routines using the routine engine without a user interface. It allows the user to request that the routine engine automatically enable any routine saved by the routine editor. Many other examples of the advantages of the architecture can be given.

The routine manager simply gives the user a graphical method of managing routines. When analyzing the tasks a

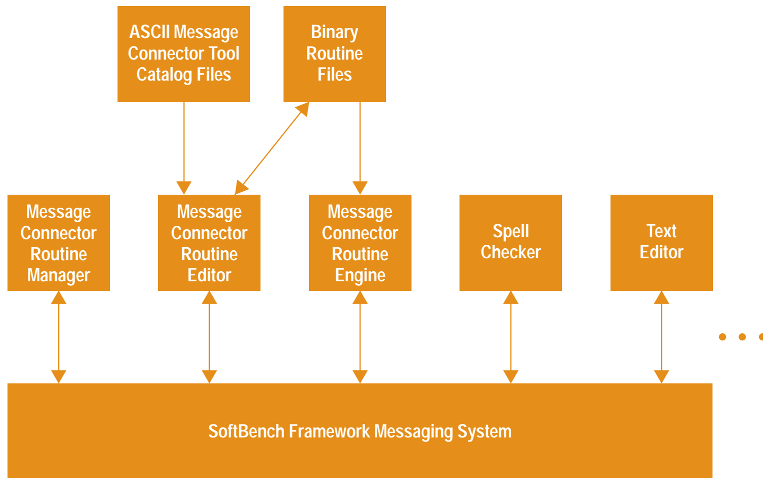

Fig. 4. SoftBench Message Connector architecture. The three major Message Connector modules—the routine manager, the routine editor, and the routine engine—are treated as components like the tools.

Message Connector user would perform, it was concluded that the routine manager would be in the user's environment most of the time. It was also concluded that the routine manager would be an icon most of the time. As a result, the design goal for the routine manager was to occupy as little screen space, memory, and process space as possible. As designed and implemented, a large portion of the routine manager's user interface simply sends a message to the SoftBench Framework requesting that a service be performed. As an example, the Enable and Disable command buttons simply send a message requesting that the routine engine enable or disable the selected routine. The routine manager was designed, implemented, and tested before the implementation of the routine editor and the routine engine.

The routine editor proved to be very challenging. The Message Connector project goal stated that, "Message Connector will provide SoftBench Framework value to all levels of end users in minutes." While a simple statement, the implications were very powerful. "All levels of end users" implied that whatever the editor did, displaying the underlying raw framework would never meet the goal. All information would have to be highly abstracted, and yet raw information must be generated and could not be lost. "All levels of end users" also implied that any user could add messaging tools to the control of Message Connector, so Message Connector could not have a static view of the framework and its current tools. "In minutes" implied that there would be no need to read a manual on a specific tool's message interface and format to access the tool's functionality. It also implied that the routine editor, tool list, and tool function lists must be localizable by the user without disturbing the required raw framework information. "In minutes" also implied that there would be no writing of code to connect tools.

The routine editor underwent sixty paper prototype revisions, eighteen code revisions, and countless formal and informal cognitive tests with users ranging from administrative assistants to tenured code development engineers. It is ironic that one result of focusing a major portion of the project team's effort on the routine editor has been that various people involved with promoting the product have complained that it is too easy to use. Apparently people expect integration to be difficult, and without a demonstration, potential customers question the integrity of the person describing Message Connector. When someone is told that there is a tool that can connect other disparate tools that have no knowledge of each other, in millions of possible ways, in seconds, without writing code, it is rather hard to believe.

The routine engine turned out to be an object-oriented wonder. The routine engine must be very fast. It stores, deciphers, matches, and substitutes portions of framework messages, it receives and responds to a rapid succession of a large number of trigger messages, and it accommodates future enhancements. The routine engine is the brain, heart, and soul of Message Connector and is completely invisible.

Example Revisited

Walking through the eight steps in the simple editor/spell checker example above will show the interaction within and between each of the Message Connector components.

1. Request that Message Connector create a new routine.

This step is accomplished using the routine manager (see Fig. 5). The routine manager's task is to prompt the user for a routine name, ensure that the name has the proper file extension (.mcr), and then simply send a request to the message server to edit the named routine. The routine manager's role is largely coordination. It has no intimate knowledge of the routine editor. After sending the request to the message server to edit the named routine, the routine manager will await a notification from the message server of whether the edit was a success or a failure. The routine manager then posts the status of the request.

A separate routine editor is started for each routine edit request received by the message server. When the routine manager sends a request to the message server to edit a routine, the message server starts a routine editor and the routine editor initializes itself and sends a notification of success or failure back to the message server. Fig. 6 shows a typical routine editor screen.

2. Select the WHEN: tool to trigger an action.

In the case of creating a new routine, there is no routine to load into the editor and therefore the WHEN: and THEN: fields are displayed empty. The routine editor searches for and displays all possible tools available for Message Connector to manipulate. It is important that Message Connector is actually searching for Message Connector tool catalog files, not the tools themselves. For each file found, the file name is displayed as a tool in the routine editor.

Fig. 5. Typical routine manager screen.

The Message Connector tool catalog files contain three important pieces of information. The catalog files are ASCII files that contain the raw messages required to access the functions of the tool being cataloged. The catalog files also contain the abstractions of the raw messages (these are displayed to the user, not the raw messages) and any message help that may be required by a user. For most tools, the catalog file is provided for the user by the person who added the message interface. If the catalog file does not exist for a particular tool, it can be created using that tool's message interface documentation. The catalog does not have to be created by the tool provider. The catalog files can also be edited by the user to change the abstraction displayed or to hide some of the seldom used functions.

When the user selects a tool from the Message Connector routine editor tool list, the routine editor goes out and parses the tool's catalog file for all applicable message abstractions and displays those abstractions.

3. Select the specific function of the WHEN: *tool that will trigger the action.*

When a user selects a WHEN: function and copies that function to the WHEN: statement, the routine editor reads the function's raw message and the abstraction of the raw message. Only the abstraction is displayed to the user, but both the raw message and the abstraction are temporarily preserved until the user saves the routine.

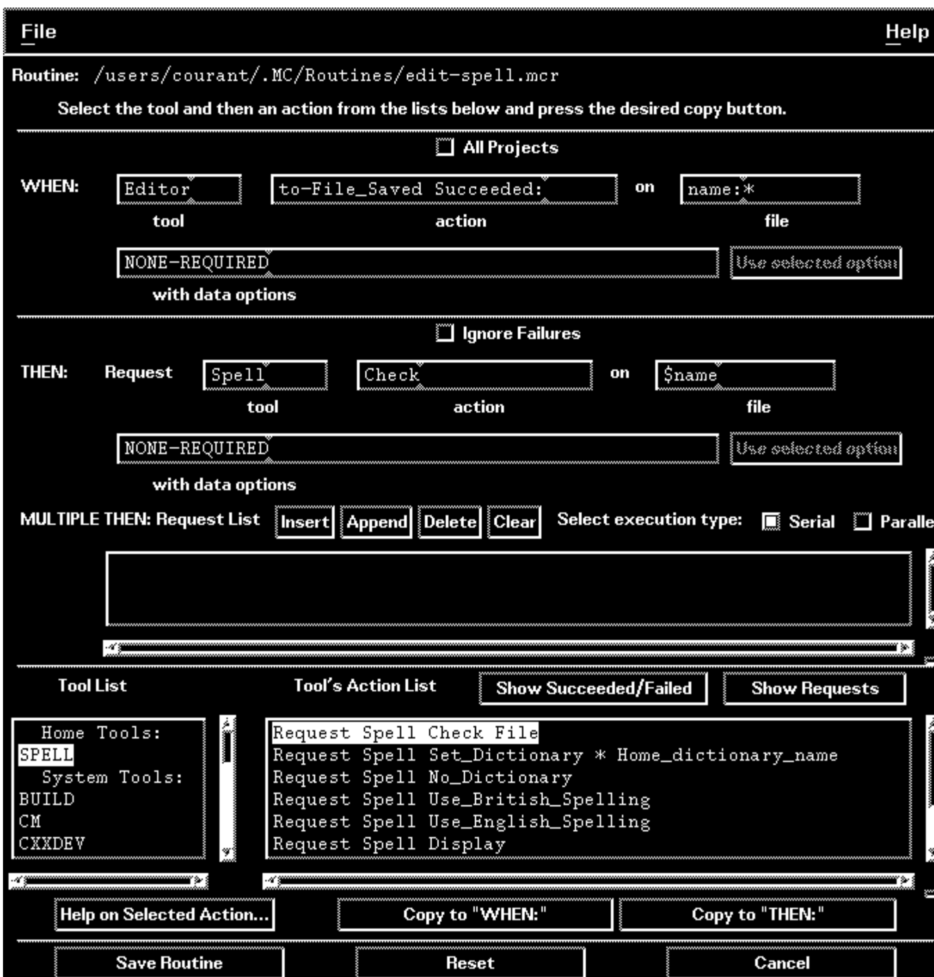

Fig. 6. Typical routine editor screen.

4. Select the THEN: tool to respond to the action.

5. Select the specific function that will respond.

These steps are similar to the WHEN: steps.

6. Change the WHEN: and THEN: file fields.

This simply allows the user to change the values displayed on the screen. For these values, what is seen on the screen is what will be used when the user selects Save Routine.

7. Save the routine.

This step takes all of the raw messages, the message abstractions, and the screen values and assembles them into an internal routine file format which both the routine editor and the routine engine are able to read. The routine editor then writes out a binary data file into the routine file being edited and then quits.

8. Enable the routine.

This step is driven by the routine manager, but is performed by the routine engine. The user selects the routine of interest, then selects the Enable Routine button on the routine manager. Again, the routine manager's primary role is coordination. When the user selects the Enable Routine button, the routine manager simply finds the routine selected and sends a request to the message server to enable the named routine. The routine engine receives the enable request from the message server and reads the named routine. After reading the routine, the routine engine establishes the WHEN: message connection to the message server. This WHEN: connection is as general as required. If the user uses any wildcards in the WHEN: statement, the routine engine will establish a general WHEN: message connection and then wait until the message server forwards a message that matches the routine engine's message connection. If the message server forwards a matching message, the routine engine sends a request for each of the THEN: statements to the message server.

Development Process

Message Connector's transformation from a concept to a product was a wonderful challenge. The two most important elements of this transformation were a cross-functional team and complete project traceability. A decision was made before the first project meeting to assemble a cross-functional team immediately. To make the team effective, all members were considered equal in all team activity. It was made clear that the success or failure of the project was the success or failure of the entire team. This turned out to be the most important decision of the Message Connector project. The team consisted of people from human factors, learning products, product marketing, research and development, promotional marketing, and technical customer support. Most of the team members only spent a portion of their time on the Message Connector project. However, a smaller group of full-time people could never have substituted for Message Connector's cross-functional team. The collective knowledge of the team covered every aspect of product requirements, design, development, delivery, training, and promotion. During the entire life of the project nothing was forgotten and there were no surprises, with the exception of a standing ovation following a demonstration at sales training. The team worked so well that it guided and corrected itself at every juncture of the project.

One critical reason the team worked so well was the second most important element of the project—complete project traceability. There was not a single element of the project that could not be directly traced back to the project goal. This traceability provided excellent communication and direction for each team member. In the first two intense weeks of the project, the team met twice per day, one hour per meeting. These meetings derived the project goal, objectives (subgoals by team definition), and requirements. The rule of these meetings was simple: while in this portion of the project no new level of detail was attempted until the current level was fully defined, understood, and challenged by all members. As each new level of detail was defined, one criterion was that it must be directly derived from the level above—again, complete project traceability. The project goal was then posted in every team member's office to provide a constant reminder to make the correct trade-offs when working on Message Connector. This amount of time and traceability seemed excessive to some people outside of the team, but it proved to be extremely productive. All of the team members knew exactly what they were doing, what others were doing, and why they were doing it throughout the life of the project.

The project goal was made easy to remember, but was extremely challenging: "Message Connector will provide SoftBench Framework value to all levels of end users in minutes." At first glance, this seems very simple. Breaking the goal apart, there are three separate, very challenging pieces to the goal: "SoftBench Framework value," "all levels of end users," and "in minutes." As an example of the challenge, let's look more closely at the "in minutes" portion of the project goal. "In minutes" means that there is a requirement that the user find value in literally minutes using a new product that uses a rather complex framework and a large number of unknown tools that perform an unknown set of functionality. How would Message Connector provide all of this information without requiring the user to refer to any documentation? "In minutes" made a very dramatic impact on the user interface, user documentation, and user training (no training is required). These three pieces of the goal also provided the grounds for the project objectives. The project objectives then provided the basis for the project and product requirements. At each new level of detail it was reassuring to the team that there was no effort expended that did not directly trace back to the project goal. The team ownership, motivation, creativity, and productivity proved to be extremely high.

Conclusion

Using Message Connector, users of the SoftBench Framework can easily customize their tool interaction environment to treat their tools as components of a task, in literally seconds, by simply pointing and clicking. This was all made possible by immediately establishing a cross-functional team to own the project and requiring complete project traceability. An interesting fact is that early users of Message Connector developed two new components that are separate from the Message Connector product but are now shipped with it. One component (named Softshell) executes any specified

UNIX* command using messaging and can return the output of the command in a message. This allows a Message Connector user to execute UNIX commands directly as a result of an event of any tool. For example, when the user requests the editor to edit a file, if the file is read-only then execute the UNIX command to give the user write access. The second component (named XtoBMS) converts X Windows events into messages that Message Connector can use to request functionality from any component automatically. This means that when any tool maps a window to the screen, the user environment can respond with any action the user defines. This has been used extensively in process management tools so that the appearance of a tool on the screen causes the process tool to change the task a user is currently performing.

Acknowledgments

The transformation of Message Connector from concept to product was the result of a very strong team effort. While there were many people involved with the project at various points, the core Message Connector team consisted of Alan Klein representing learning products, Jan Ryles representing human factors, Byron Jenings representing research and development, Gary Thalman representing product marketing, Carol Gessner and Wayne Cook representing technical customer support, Dave Willis and Tim Tillson representing project management, and the author representing research and development. The cross-functional team approach caused roles and responsibilities to become pleasantly blurred. The entire team is the parent of Message Connector.

References

1. C. Gerety, "A New Generation of Software Development Tools," *Hewlett-Packard Journal,* Vol. 41, no. 3, June 1990, pp. 48-58.

2. M.R. Cagan, "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools," *ibid,* pp. 36-47.

3. B.D. Fromme, "HP Encapsulator: Bridging the Generation Gap," *ibid,* pp. 59-68.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.