# AN EVENT DRIVEN FRAMEWORK FOR SOFTWARE MONITORING

BY

SUMANT TAMBE, B.E.

A thesis submitted to the Graduate School

in partial fulfillment of the requirements

for the degree

Master of Science

Major Subject: Computer Science

New Mexico State University

Las Cruces, New Mexico

August 2005

"An Event Driven Framework for Software Monitoring," a thesis prepared by Sumant Tambe in partial fulfillment of the requirements for the degree, Master of Science, has been approved and accepted by the following:

---

Linda Lacey
Dean of the Graduate School

---

Jonathan Cook
Chair of the Examining Committee

---

Date

Committee in charge:

---

Dr. Jonathan Cook, Chair

---

Dr. Clinton Jeffery

---

Dr. Phillip De Leon

DEDICATION

I dedicate this work to my father Uday, my mother Uma, my brother Siddharth and my friends Archana, Akshay, Sameer, Pushkar, Harshad, Chirag, Saurabh, Omkar, Akash, Avinash, Shrikant, Vijay, Mayur and Naoman.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Jonathan E. Cook, for his encouragement, interest, and patience. Personally, I would like to thank him for sharing his knowledge which has enriched my study in Computer Science and Software Engineering in particular.

VITA

| | |
|---|---|
| April 8, 1982 | Born at Chiplun, India |
| 1999-2003 | B.E., Mumbai University, Mumbai, India |
| 2004-2005 | Research Assistant, New Mexico State University, Las Cruces, USA |

**Field of Study**

Major Field: Computer Science

ABSTRACT

AN EVENT DRIVEN FRAMEWORK FOR SOFTWARE MONITORING

BY

SUMANT TAMBE, B.E.

Master of Science

New Mexico State University

Las Cruces, New Mexico, 2005

Dr. Jonathan Cook, Chair

The Event Tool Framework (ETF) is a framework of modifications and extensions to the dynamic linker that allow developers to have dynamic control over the linking process. The modified dynamic linker is called DDL. DDL allows a tool builder to intercept the binding operation, and allows the tool to take a variety of actions. ETF, which is built on top of DDL, allows powerful control over the linking process, enables the easy construction of runtime monitoring tools and supports the runtime evolution of dynamically linked programs. An "Introspection Suite" is a set of software modules with a common goal of program monitoring. Behind the vision of an "Introspection Suite" is the idea of integrated, co-operating, active tools around the running application. This work shows how event-based integration, in which modules interact by announcing and responding to occurrences of events, can be successfully applied to runtime program monitoring.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# 1 INTRODUCTION

Software systems deployment has made a rapid march from almost no run-time management facilities to more and more dynamic management capabilities. Run-time frameworks have been central in this march, and now many such frameworks include the capabilities for managing and monitoring the applications running on them. The frameworks give the developers the low-level capabilities from which higher level tools and applications can be built. Perhaps best known is the Java environment, with introspection and reflection capability, customizable class loading, and now a programmable "debugger" API that gives detailed control over the application. Yet most of these frameworks support new applications and do not look back to what legacy applications might need. There is still much software running on legacy platforms, and much software still being created for these platforms. This makes support for self-management in the legacy software arena all that much more important.

The existing platform of shared, dynamic link libraries has been long overlooked in its potential for providing to developers such capabilities as management, configurability, and monitoring. With the proper support, the dynamic link mechanisms can be exploited to support many CBSE and software architecture ideas, and can provide a platform for monitoring and self-management capabilities.

Many other uses for introspection and manipulation exist, from debugging and profiling to redirecting invocations to specific versions or even mapping a single call to multiple component versions for fault tolerance and reliability [5]. Some of these uses are general, but many are often very specific to a particular application. Because the effort in building introspection and manipulation tools is very high, projects are often prevented from building application-specific tools or rapidly prototyping new general-purpose tools.

This work is a step towards building a true component framework using shared objects. DDL (Dynamic Dynamic Linker), a customized dynamic linker that offers programmatic access to the linking process, and enables dynamic manipulation of the existing bindings in a running software system. ETF, which is built on top of DDL, allows manipulation of the linking process. ETF helps to build runtime monitoring tools quickly and easily. It also supports the runtime evolution of dynamically linked programs. Instead of having one large monolithic tool to serve all the different needs[1] of runtime software monitoring, a set of integrated software modules, each dedicated for a small subset of the big task and co-operating with each other towards a common goal can be visualized. It is called an "Introspection Suite." The idea of the "Introspection Suite" can be realized by building an extensible and flexible communication mechanism to integrate loosely coupled, active tools around the running application. This work shows how "implicit invocation," in which software modules express their interest in receiving certain types of data that are then routed, usually by a sever, to the appropriate recipients, can be successfully applied to runtime program monitoring. This work also describes how it can support the deployment of CBSE and system architecture composition ideas. The shared object framework can be a true software component deployment platform, and can offer the capability of building self-management into legacy software systems.

[1]Inserting wrappers for tracing, security, assertion checks, generating function call-graph to name a few.

# 2 DYNAMIC LINKING

The fundamental action of a linker is to take multiple separately-compiled pieces of object code and resolve the unknown shared symbols into addresses, so that the object code can execute without any missing pieces of information. Dynamic linking leaves the symbol resolution process to be completed at runtime. The external symbols are still "resolved" during the static link process—however, only a placeholder dependency reference to the dynamic link library that contains the symbol is put into the executable object code.

Shared objects (or dynamic link libraries) delay the binding of externally needed symbols (functions, methods, global data) to the runtime of the program. An extra module, the dynamic linker, is loaded with the program, and accomplishes the dynamic linking necessary for the program to complete its execution

From here on, the terms shared library and/or shared object are used rather than dynamic link library, because these terms are more traditional, and capture the important notion that dynamically linked objects can be shared among processes. The code pages—which typically make up most of a shared library—are read-only and the code in them is compiled to be position-independent[1] and thus it can be mapped into multiple process spaces, even at different addresses.

## 2.1 Dynamic Linking Mechanism

In the Executable and Linking Format (ELF) [13], dynamic linking uses two tables: the Procedure Linkage Table (PLT) and the Global Offset Table (GOT). Calls to external functions (and often to internal functions as well) use these tables to effect an indirect call. Figure 2.1 shows a call to the *printf* function as it goes through the PLT on an already resolved entry. The actual call site in the

[1]With the GNU C/C++ compiler, generating position-independent code must be explicitly selected with the -fPIC option.

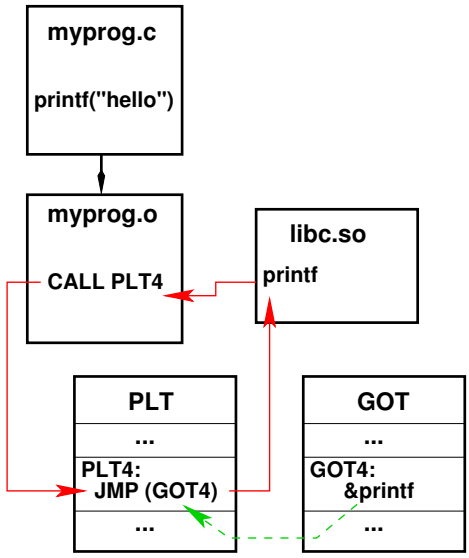


Figure 2.1: Function call through a dynamic link.

program "calls" an entry in the PLT. The PLT is executable code, with basically a jump instruction for each entry. The jump is an indirect jump that uses the corresponding entry in the GOT—an address—to jump to the correct function. Before a symbol has been resolved, the GOT entry (in effect) has the address of the dynamic linker's symbol resolution routine in it. In detail, the PLT entry has a couple of instructions below its main jump. The GOT entry initially points back to these instructions. Their job is to push the symbol name as another parameter and then effect the jump to the dynamic linker, which still goes through yet another PLT entry in order to push the library name as a second parameter for the dynamic linker. After linking is done, only the first jump instruction is used.

Thus, the PLT and GOT tables centralize the code that uses the dynamically bound links and the addresses of those links. The PLT is code while the GOT is data—a table of addresses of functions. The dynamic linker is invoked upon the first call to a function, and its job is to find the symbol (possibly needed to load the shared library into memory), determine its address, load that address

into the correct GOT entry, and effect a jump to the function. The function returns directly to the original call site (since only jumps occurred in between), and all subsequent calls only incur a one-instruction overhead since the GOT now contains the correct address of the function.[2]

For position-independent code, symbols representing global data are also referred to through the GOT, but not through the PLT. The GOT is an address table for all external symbols (and internal globals, as well), while the PLT is specifically for function calls.

## 2.2 C++ and Dynamic Linking

C++ is briefly mentioned here. The dynamic linking mechanism (and static linking as well) only knows about symbol names, it does not have inherent understanding of classes as such. When C++ is compiled, the compiler does *name mangling* to convert the class and method name into a single unique symbol. Because C++ allows method overloading (same name but different parameters), the types of the parameters are also used in the name mangling to produce a unique name for each method, overloaded or not, in a class. In this way, C++ is "invisible" to the dynamic linker, and class methods are only related in that their mangled names all include the same class name. A further complication is in polymorphic behavior, especially the mechanism of virtual methods that allow specific runtime-selected behavior based on the actual object type being used. A class *vtable*—a table of function pointers similar to the GOT—is used to implement polymorphic method calls.

[2]Dynamic linkers do generally support binding modes other than first-call binding. Load-time binding allows resolution before the first call, essential for real-time systems, and suppressed binding forces every single function call to trigger a resolution (which never updates the GOT), which can be important in debugging situations.

# 3 THE DYNAMIC DYNAMIC LINKER (DDL)

Though a full-fledged component framework using shared objects is a distant goal, this work is the first step towards realizing this goal. The modified GNU dynamic linker is called DDL. Interestingly, DDL itself already enables virtually all of the needs of a true component framework. Existing capabilities of the dynamic linker would help understand DDL in detail.

## 3.1 The Dynamic Linker and The Preload Libraries

Current dynamic linkers do allow rudimentary control over the linking process. Most allow an application to *preload* a library, so that symbols in the preloaded library will take precedence over those in libraries loaded later. This library is specified using an environmental variable, LD_PRELOAD. It is typically done for things like wrapping system calls with particular behavior (e.g., a virtual file system) or security checks. However, the preload mechanism is static and is cumbersome to program—the wrappers need to explicitly load and find the *actual* symbols that it is wrapping.

## 3.2 The Modified Linker

DDL is an extension to the GNU dynamic linker, and is extensible itself. Figure 3.1 shows the high-level system architecture that DDL implements. The shaded portions indicate parts of the system that DDL does not modify. The application and application libraries are not modified, at the source or binary level, and the bulk of the system dynamic linker is unmodified. The GNU dynamic linker is an integral part of all the Linux distributions and it has been ported to several other platforms as well. The modifications to the GNU dynamic linker do not depend statically on any libraries. Although, it depends on some functionality

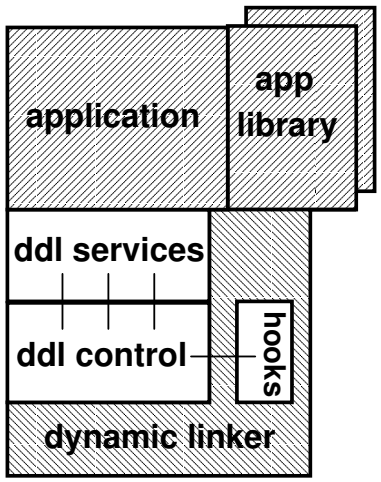


Figure 3.1: DDL system architecture.

provided by the *pthread* library. It resolves the *pthread*-functions dynamically. Therefore, the modifications to the GNU dynamic linker should be portable where the GNU dynamic linker itself and the *pthread* library has been ported.

At the lowest level, hooks are added into the dynamic linker to allow interaction with the linking process. On top of these hooks useful service abstractions are built so that tool builders would not need to start from scratch. Further, application level services are also implemented that provide even higher level interaction for some types of common services—one such service is scripting language support.

## 3.3 Linker Modifications

One of the goals of this project was to have very minimal modification to the GNU dynamic linker itself. The aim was to put as much code outside the linker as possible. Thus, the essential modifications boil down to callback hooks in the linker code itself.

To activate the modifications, linker code checks an environment variable, LD_REDIRECT. If this is not defined, the modifications are ignored and the linker operates normally.

The callback hooks are in the form of function pointers. If LD_REDIRECT is defined, the code attempts to initialize the function pointers by doing symbol lookups—using the dynamic linker code itself. The symbols are of course not defined in the linker or in the application and its libraries. They are defined only *if* a tool library has been preloaded (using the existing LD_PRELOAD capability), and thus the hooks internal to the linker get connected to tool functionality out in a library.

All of this happens once, at application startup. Only if the function pointer callback hooks are initialized properly will DDL act differently than the regular GNU linker. If they are, then at key points in the linking process, the callbacks are invoked and the external tool has the opportunity to interact with and manipulate the linking process. In this, the hooks built into the dynamic linker do not provide an API to external services but rather they use an API provided by the DDL control library. The DDL control library and the tools that use it are thus event-driven and passive unless they spawn their own threads.

An important aspect of modifications to DDL is that it is thread-safe. The regular dynamic linker is by default thread safe because it always updates its global data structures equivalently, so it does not matter if threads interleave their updates. With DDL this may not always be the case, and DDL needs some of its own static data in any case. DDL does support both threaded applications, and client tools that create their own threads. However, DDL does not depend on a thread library. Since the initial tests performed on DDL indicate that overlapping link requests from multiple threads are vanishingly rare, a simple busy waiting technique is used in very small sections of code rather than *pthread*-based library routines.

```
void  redirect_init (void);
int   redirect_isactive (int callback_type, int thread_id);
char* redirect_lookup (char *symbol, char *from_libname,
                       char **force_libname, int thread_id);
int   redirect_definition (char *found_symbol, char *found_libname,
                           void *found_func_address, char *orig_symbol,
                           char *from_libname, void *link_GOT_addr,
                           int thread_id);
int   redirect_offset(char *found_symbol, char *from_libname,
                      int thread_id);
void  redirect_symdef(char *symbol, char *libname, void *address,
                      char *caller_libname, int thread_id);
```

Figure 3.2: DDL callback API.

### 3.3.1 Link Process Manipulation

The fundamental capability that DDL supports is link interception and redirection. This allows DDL and the tools that use it to peer into the dynamic linking process, and control it. This work mostly focuses on intercepting and manipulating run-time bindings, which are generally most of the bindings that regular programs use, and thus the interface below is concentrated in that area. However, currently information gathering on load-time symbols is supported.[1] The callback interfaces also supply the thread ID, in case the client code must perform some thread-specific operations. The callback points are shown in Figure 3.2.

**redirect_init** : This is called once to initialize anything the redirection code needs. It is called before *main()*, so it should not depend on any initializations in the application.

**redirect_isactive** : This is invoked just prior to each of the other callbacks, with the callback type given as a parameter. It should return non-zero if

[1]The run-time and load-time binding mechanisms do need quite different modifications to the core dynamic linker.

redirection is desired. If it returns zero, the other callbacks will not be invoked. This offers dynamic on/off control of the DDL services.

**redirect_lookup** : This hook is called before the linker tries to resolve a function symbol that is being invoked for the first time. As parameters this function receives the symbol of the function to be resolved and the name of the shared object from which this call is coming from (i.e., the main application or some shared library). It returns either a) the new name to which it should be redirected or b) the exact same pointer it received as a parameter, indicating no redirection is taking place. It can also return a library name through force_libname, which will restrict DDL to find the symbol in that library.

**redirect_definition** : This function receives all the information about the symbol and link that have been resolved. For convenience it again provides the original symbol looked up and the name of the shared object the link request is coming from (as in redirect_lookup). It also provides the symbol that was actually resolved (different if redirect_lookup returned a different name), the library that the resolved symbol is in, the address of the resolved symbol, and the address of the GOT entry that is being updated. Through this, analysis tools can keep track of every link request and every symbol definition that is resolved.

**redirect_offset** : This function provides the offset (in bytes) that will be added to the address of the symbol that was used. The parameters are the symbol that was just looked up, and the library name that the call originated from. This allows application-level PLT-style table-based redirection. It should return 0 if table-based redirection is not being used.

**redirect_symdef** : This function provides the information about load-time symbol definitions, many of which are not function symbols and so would not

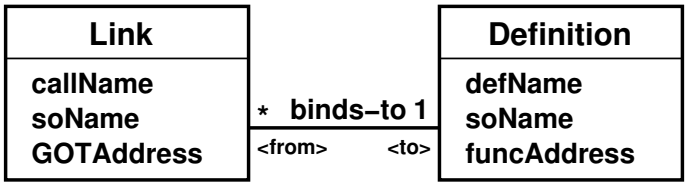


Figure 3.3: Link and definition UML.

appear in the above callbacks. The symbol name, library name in which it occurs, and its resolved address are all provided. The last parameter is unused for now.

Thus, with this interface, tools using DDL can inspect every link request, choose whether to redirect it to another symbol name, and record the information about the resolved symbol and about the link itself.

Redirection capabilities allow for table-based redirection. That is, the redirection code can implement a mechanism similar to the built-in PLT-GOT table-based dynamic linking. The *redirect_offset* callback supports this capability. This will be explained in the next section in more detail.

Figure 3.3 shows how the links (GOT entries bound to functions) and definitions (Functions themselves) relate to each other and the information that uniquely describes each of them.

A link is defined by the original symbol name it is supposed to be linked to, the name of the shared object it is for (the main application or a shared library), and the address of the GOT entry where its binding address is stored.

A definition is defined by its symbol name, the name of the shared object it exists in, and the address at which it exists.

Further information that is useful to save is a reference from the link to its current definition, and in the reverse direction a set of links that currently reference a given definition.

Without using the redirection capabilities, all links will point to definitions of the same name. There are potentially multiple links to the same definition because each shared object that calls that function will have its own link (i.e., its own PLT/GOT and its own unique PLT/GOT entry for calls to that function). If the redirection capabilities are being used, then the called name associated with the link can be different from the defined name of the definition; thus it is important to keep track of these names separately.

## 3.4 Table-Based Redirection

In general, the link interception and redirection capability supports redirecting the calls of each *unique* function to some other *unique* function. While the mechanism does not prevent multiple redirections to the same symbol, in practice this does not make sense, outside of perhaps a few special cases. This is because the called function cannot differentiate between the calls, and thus cannot know which calls map to which original symbols. Thus, only if the arguments from all the calls were the same *and* the desired behavior was the same would it make sense to redirect multiple symbols to the same target.

There are cases in runtime monitoring and program maintenance, however, where it would be useful to have a *concentrator* function that did receive calls for multiple symbols and was able to differentiate them. For example, if one wanted to trace all calls in a program, it would be nice to have a single wrapper do the job rather than a unique wrapper for every function.

To support this functionality, the same basic mechanisms of the system PLT/GOT jump tables is used. The DDL interface that supports this capability of providing an *offset* to be added to the address of a symbol that is resolved.

To use this capability, it needs a jump table. A simple (and non-thread-safe) example of this is shown in Figure 3.4.

```
unsigned int func_id;

void wrapper_plt()
{
   asm("   movl $0, func_id
           jmp wrapper
           movl $1, func_id
           jmp wrapper
           ...
           movl $98, func_id
           jmp wrapper
           movl $99, func_id
           jmp wrapper\n");
}
```

Figure 3.4: Table-based redirection sample code.

This example represents a 100-entry jump table, where each entry is a move/jump instruction pair that sets a global variable to its index value and then jumps to the wrapper function. Note that the program never *calls* the *wrapper_plt* function—rather, it calls directly to one of the entries, which in turn jumps to the wrapper. Use of DDL would redirect each symbol to $(wrapper\_plt + 3 + i * 15)$, where $i$ is the entry assigned to that symbol. To do this it would do symbol redirection to "wrapper_plt," allow the dynamic linker to find that symbol, and then add the offset using the *redirect_offset* DDL callback. At the same time, the DDL extension would save the symbol string in a string table, at the same index being used in the jump table.

The wrapper function, using the global $func\_id$, would have access to the index of the function currently being called, and from there name of the function can be obtained. After doing its tracing behavior (or whatever it is supposed to do), it could use *dlsym()* to resolve the original function and to call it. Functions with different argument vector lengths can still be handled by the same wrapper, since the reverse-calling convention ensures that extra argument data is ignored.

The wrapper only needs to know the maximum argument bytes it needs to push on the stack.

While the jump table must be created in a platform-dependent manner, the basic idea remains essentially the same on most platforms, and through the symbol-plus-offset mechanism that DDL exposes to the user, effective use of a single site for multiple redirected symbols can be accomplished.

As one example, This capability was used to fully trace the SimpleScalar CPU simulator [1, 23].

## 3.5 Runtime Link Modification

Section 3.3.1 describes definitions and links maintained in an internal data structure called *LinkDef* data structure. It maintains resolved symbols, their addresses, addresses of GOT entries and the current definition of the function a link is referring to. Maintaining this information during the runtime of the program allows developers to support dynamic program evolution through runtime link modification.

In order to modify a link, the address in its GOT entry needs to be changed to the address of some other function. All the subsequent calls through that link will be directed to the new function. Note that these calls are from all the call sites in the shared object whose link are modified. Thus, the granularity of program evolution is at the shared object level.

Once the link is initially resolved, however, the linker is never going to be called for that particular link again. Therefore, some way of regaining control over the execution of the program is required to perform runtime link modification. If the DDL-based tool spawned its own threads, then this is immediately possible. But even if the tool depends on the application thread, link modification is still possible. While in the long run some OS support would be needed for generic

framework control of an application, As proof of concept for this, the OS's signal mechanism is employed to accomplish this.

## 3.6 Signal-based Link Modification

In the signal-based implementation, the user must first provide a specification file describing which links they want to modify. While the file does not need to be created when the application is first started, an agreed-upon file name and location *does* need to be known at startup, since there is no way to send such information through a signal. There is a signal handler waiting for USR2 signal in the dynamic linker as part of redirection library. This handler reads in the specification file, traverses the internal data structures of links and definitions, and modifies the GOT entries for the specified links. When the handler is completed and the application regains control, subsequent calls on those links will go to the new definitions.

The lookup performed by the signal handler is fast because hashing is used to index the data structures. It is implemented as hash tables with buckets. The downside of the signal mechanism is that the user process itself might be wanting to use the same signal. If the user process also installs its own signal handler for the same signal, the signal handler in place already will likely never be called.[2] Redirection library will never be called and will practically turn off entire dynamic link modification capability.

In runtime link modification, it can be the case that a new definition for which links should point to may not yet have been loaded by the dynamic linker. This means that it must be able to not only search the existing definitions but also bring in new definitions, possibly even loading new shared libraries. The standard

[2]DDL mechanisms themselves can be used to wrap the signal() call and protect the installed signal handler while still giving the application its desired functionality.

*dlopen()* and *dlsym()* interface can be used to load new libraries. This is UNIX's way for user-level access to shared libraries.

Other work in dynamic program evolution has noted a desire to perform transactional updating—making sure that a module is not being actively used before updating references to or away from it [18]. This often boils down to checking the call stack to see if any functions in the module are active. At this moment, there is no support for such capability. In the current mechanism, existing calls through links being modified have already invoked the old definition, and those will eventually complete.

While trying to revert the redirection, it is quite possible that the original function has not been loaded and therefore there is no address of it to put it back in the GOT entry. This problem is solved by explicitly loading the symbol using *dlsym()*: a part of programming interface to dynamic linking loader. The GOT is modified as usual when *dlsym()* succeeds and returns the address of the desired symbol. The specification file is used when *redirect_init()* is called and setup the internal data structure for possible redirection before program begins execution. Checking whether the wrapper function definition is really present in the library or not could be a worthwhile task.

All the links share a single definition in normal course. The internal data structure allows to redirect calls to the function from different shared objects to different definitions of wrapper functions. For example, a call can be redirected from *putchar()* invoked from some library to *wrap_putchar()* but a call from main may not be redirected. In the later case, it might directly invoke C library *putchar()* function.

# 4 ETF: AN EXTENSIBLE, EVENT-BASED TOOL FRAMEWORK

Behind the vision of an "Introspection Suite" for runtime monitoring is the idea of integrated, co-operating tools around the running application. A serious concern in the construction of such a system of co-operating software modules (tools) is integration. Approaches to integration range from loose, in which modules have little or no knowledge of one another, to tight, in which modules require much knowledge about one another. Loose integration helps reduce the impact on a system when modules are added or changed. Event-based integration, in which modules interact by announcing and responding to occurrences called events, is perhaps the most prevalent loose integration approach.

Event-based programming is at the heart of countless software applications that wait for user-generated events and respond to them. Dozens of software integration systems, such as FIELD [20], also use event-based models in which multiple software modules react to events announced by other modules. Wasserman [28] defined control integration as the ability of software "tools" (modules) to "notify one another of events... as well as the ability to activate the tools under the program control." Thomas and Nejmeh [26] extended Wasserman's work and identified two basic control integration properties: provision of invocable operations ("services") by tools and use of those operations by other tools. A subset of control integration typified by FIELD is also known as "implicit invocation."

Control integration, event-based integration and implicit invocation are not the same, rather they form a hierarchy. Control integration is too broad a term, since it may refer to loose as well as tight integration. ETF is about loose integration. Implicit invocation, originally called "selective broadcast" was pioneered by FIELD [20]. Implicit invocation refers to anonymous multi-casting

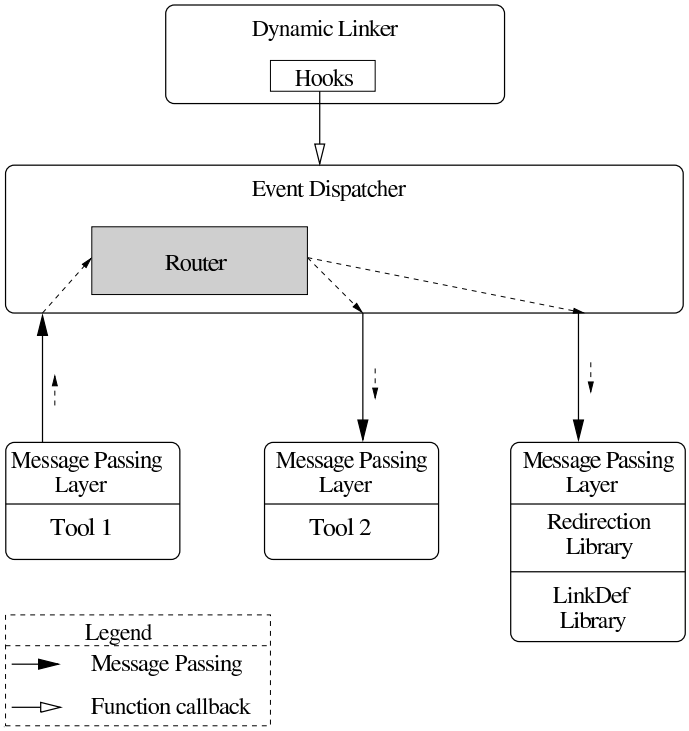


Figure 4.1: An instance of the ETF tool framework.

and is therefore more specific than general event-based integration. Therefore, implicit invocation or selective broadcast is the right term to describe ETF.

## 4.1 Motivation Behind ETF

DDL is a tool that fundamentally opens up ways to control the linking process, but it is quite low level and primitive. Thus, further supporting infrastructure is necessary to truly enable monitoring and management of applications.

To begin with, a typical reusable API approach looks desirable in which potential tools might use the services provided by it. However, it became clear that multiple tools might be needed at a single time. For example, if a management tool is being used to allow reconfiguration, a monitoring tool that collects data from one piece of the system should also be able to be deployed within that system.

To this end, ETF, an extensible, event-based tool framework was created for allowing multiple tools access to DDL capabilities. This is shown in Figure 4.1. Tools, each embodied in a shared object, register with the event dispatcher to be notified of link request events. The event dispatcher is what interacts with the base DDL support, rather than the tools themselves.

The idea of an "introspection suite" consists of a set of collaborating, management or monitoring tools. This idea, of course, is not new, being used as far back as the FIELD environment [20], albeit at a different level. In ETF, if one wants to plug in a tool that watches a particular class and checks invariant properties, this tool should not interfere with other tools that might be doing other things.

ETF is an event-driven framework for developing tools which monitor and/or intercept dynamic linking of function calls in an application. Fundamentally, in ETF, an event is nothing but a dynamic link request generated by application. ETF makes it easy for tool developers to *listen* to dynamic link resolution events and *redirect* function calls to different symbols. ETF also consists of Redirection Library, an infrastructure library which provides commonly used services such as, redirection of symbols.

## 4.2 Introduction To ETF

The GNU dynamic loader generates link resolution events. There are events corresponding to every dynamic link request generated by application binary or the externally linked shared objects it is dependent on. It includes system C library as well as programmer defined shared libraries (.so files). The modifications to the dynamic linker are known as hooks. ETF gets access to link resolution information through these hooks. This information consists of the symbol getting resolved, the GOT entry of the caller library, the definition symbol and its address. The Modified dynamic linker, (DDL (Dynamic Dynamic Linker)) is itself a

reusable piece of software as every hook has fixed behavior and the code invoked by hooks is external to the linker.

The Event Dispatcher which is central to ETF realizes the *Hollywood Model* of interaction: "Don't call us, we'll call you." The mechanisms provided by the Event Dispatcher make it an extensible framework. These mechanisms are supported by two components of ETF: the Registrar and the Router. ETF allows the tools as well as the Redirection Library to register themselves with the Event Dispatcher and receive notification of the dynamic link resolution events. The tools may optionally get notifications of these events through a predefined callback interface implemented by each tool. For every event of interest to the tool, it has to register a pattern corresponding to the event with the Event Dispatcher. The Event Dispatcher guarantees a sequence of calls to these callback functions.

The tools are the artifacts of users of ETF. The possibilities of tools are really boundless. A developer might want build a simple program introspection tool or else a program visualization tool to show a graphic display of the dynamic linking process. Developers might also use the services provided by the Redirection Library to manipulate linking and build some manipulation tool. Physically, a tool is nothing but a dynamically linked shared object file (.so file). ETF reads a file listing all the tools to load during initialization and allows these tools to initialize themselves. Every tool shared object has to export a fixed, predefined interface. The Event Dispatcher and the tools interact with each other though this fixed interface and of course, the Event Dispatcher API.

The crux of symbol redirection lies in the Redirection Library. The Redirection Library listens to link resolution notifications produced by the Event Dispatcher and builds an internal data structure of links and definitions for future use. The Redirection Library offers an easy to use API to redirect a symbol to another symbol or to a table in case of the table-based redirection. A message passing layer on top of this API enables communication with the other loaded

tools. This message passing layer also accepts redirection requests sent by tools through the mechanisms provided by ETF. The Redirection Library is also treated as a tool and therefore it also exports interface of a tool to receive notifications from the Event Dispatcher. It is in fact a special tool called a master tool. In the context of ETF, a master tool is the one which actually has the authority to redirect a symbol at link resolution time and make changes in the GOT entries of the loaded shared objects. The Event Dispatcher treats the Redirection Library as the only master tool in ETF.

The Redirection Library maintains an internal data structure to provide its services. This internal data structure is knows as the *LinkDef* data structure. The LinkDef data structure maintains the entire history of the dynamic link requests made by the running application as well as any future redirection requests. The Redirection Library manipulates the sole instance of LinkDef data structure through a broad API provided by the LinkDef library. The tools can also manipulate the same instance through this API although it is not recommended in general sense.

This completes an overview of the components of ETF. The following sections describe these components and their interaction in great detail.

## 4.3 Components of the Event Tool Framework (ETF)

ETF consists of the hooks inserted in the Dynamic Dynamic Linker (DDL), the Redirection Library, the Event Dispatcher and the tools. The Dynamic Dynamic Linker and the hooks are explained in detail in the previous chapter. A detailed description of the remaining three components is given below. The terminology used here has been borrowed from the EBI framework [3] described by Barrett, Clarke, Tarr and Wise.

### 4.3.1 The Event Dispatcher

The mechanisms supported by the Event Dispatcher to allow communication between software modules (tools) is known as "implicit invocation"(a.k.a. selective broadcast). Garlan and Shaw describe implicit invocation [8] systems: "The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event 'implicitly' causes the invocation of procedures in other modules."

The Event Dispatcher supports the participant interaction through two framework components: the Registrar and the Router. The Registrar and the Router together accomplish the task of communication between the participants of ETF. Each represents a basic aspect of integration. Participants are the interacting tools. Registration distinguishes tools that can communicate from those that cannot. Routing transmits data among participants.

Participants of ETF are called informer tools and listener tools. The informer tools detect events and generate messages corresponding to them. The Dynamic Dynamic Linker (DDL) is analogous to an informer. The callback hooks inserted into DDL invoke specific functions in the Event Dispatcher. The Event Dispatcher is adapted to receive information from DDL. Thus Event Dispatcher itself becomes an informer. The Event Dispatcher generates one event corresponding to every invocation of the callback hooks. It generates a REDIRECT_LOOKUP event to correspond to an invocation of the redirect_lookup hook, a REDIRECT_DEFINITION event to correspond to an invocation of the redirect_definition hook and a REDIRECT_OFFSET event to correspond to an invocation of the redirect_offset hook. The information associated with every event

is passed in a message. Detailed description of the message format is given in the section 4.3.2.

### 4.3.2 Events, Messages and Patterns

An event is an occurrence such as a link resolution activity initiated by an application, an invocation of a function wrapper, the change of a participant's state or the sending of a message. Informers detect events. A message is the information emitted by an informer in response to an event or events. Thus, messages are the manifestations of events in the framework. Every message belongs to some registered pattern. A pattern is a plain string and represents the format of the corresponding messages. A pattern string is made up of predefined DDL data types. A message must have as many data values as data types defined in the pattern. They also need to be arranged in the exact same order. This enables new tools to interpret messages generated by the other tools.

Patterns are of two types: input patterns and output patterns. An input pattern represents messages that a tool will be notified of. An output pattern represents messages that a tool will generate. Thus, a listener tool must register at least one input pattern and an informer tool must register at least one output pattern. Two tools are allowed to communicate with each other when an input pattern name matches to that of the output pattern registered by the other tool.

Messages in ETF can be of three different types, *request*, *reply* and *notification*. A simple announcement is modeled by notifications. There is no reply expected from the listening tools if a notification is received. Although, a tool may send another request or notification on receipt of a notification. On the other hand, a reply must correspond to a request. ETF does not accept a request type of message if there is no tool registered to process the request and deliver a valid reply. A requesting tool may or may not be interested in the reply. In such a case, requesting tool does not have to register an input pattern to listen

to the corresponding reply. If for some reason, there is no reply generated for a request (it could be because of a failure in the request processing) the Event Dispatcher sends a predefined failure reply to the requesting tool. Therefore, for every requesting tool, the Event Dispatcher automatically registers a failure reply pattern on behalf of the tool. The tool must export the necessary interface to receive the reply. Readers are directed to section 4.3.4 for more information on the tool's interface.

Every message also has a status: *success* and *failure*. The message status describes either a successful operation or a failure in the operation. There is no semantics attached to a status as far as message delivery is concerned. It is just for the sake of convenience. The Event Dispatcher uses the failure status flag if a request fails to generate a valid reply.

Every message, no matter what type, also has a mode of delivery: *synchronous* or *asynchronous*. In case of a synchronous message, the Event Dispatcher suspends the execution of the calling thread until the message is delivered. If the message is a synchronous request then the caller is suspended until a valid (failure/success) reply is delivered to the requesting tool. There can also be synchronous replies.

*Per instance* specification is chosen over *per type* specification of the above attributes for increased flexibility when messages are sent, since the former specification can model the latter, but not the vice-versa.

Every message has a unique message id. Every request has a unique message id as well as a unique request-reply id. A reply corresponding to a request has the same request-reply id but a different unique message id. A message also has a sender id associated with it. A message can carry arbitrary amount of data with it. Order and data types of the data values associated with a message are described in the message pattern. Every message carries a reference to the pattern it belongs to.

Table 4.1: Data types allowed in an ETF pattern.

| Keyword | C Data Type | Represents |
|---|---|---|
| DDL_LINKFNAME | char pointer | a function name |
| DDL_LINKLIBNAME | char pointer | the caller library name |
| DDL_DEFFNAME | char pointer | a function name (definition) |
| DDL_DEFLIBNAME | char pointer | the library name which contains the definition |
| DDL_STRING | char pointer | any arbitrary string |
| DDL_OFFSET | int | an integer offset |
| DDL_ADDRESS | unsigned int | any arbitrary address |
| DDL_INT | int | any arbitrary integer |
| DDL_CHAR | int | any arbitrary character |
| DDL_DOUBLE | double * | a pointer to an arbitrary double precision floating point value |
| DDL_NEXTDATA | unsigned int | a pointer to a dynamically allocated (additional) chunk of data. |

A pattern is a plain null-terminated string. Every pattern begins with a name. Followed by the name, a pattern may contain zero or more "DLL data types." The data types defined by the Event Dispatcher and their respective meaning is given in table 4.1.

The Event Dispatcher interprets the data values associated with a message simply as a block of memory. A simple convention is followed while sending data via messages. Except integers and characters, data part of a message holds pointers to respective data values. By default, maximum size of data part of a message is equal to *MAX_DATAVALUES*, an implementation defined constant. The size of data values occupied is stored in the message as *datacount*. Therefore, data values are available in the range *data[0]* to *data[datacount-1]*. Among the DDL data types defined in Table 4.1, DDL_NEXTDATA requires a special attention. It is a way of linking more than one data blocks together to carry arbitrary amount of data with a message. If a message has more than *MAX_DATAVALUES* of data, then the last datavalue, i.e. *data[MAX_DATAVALUES-1]*, of the first data block is a pointer to the next block of data. This block is allocated dynamically

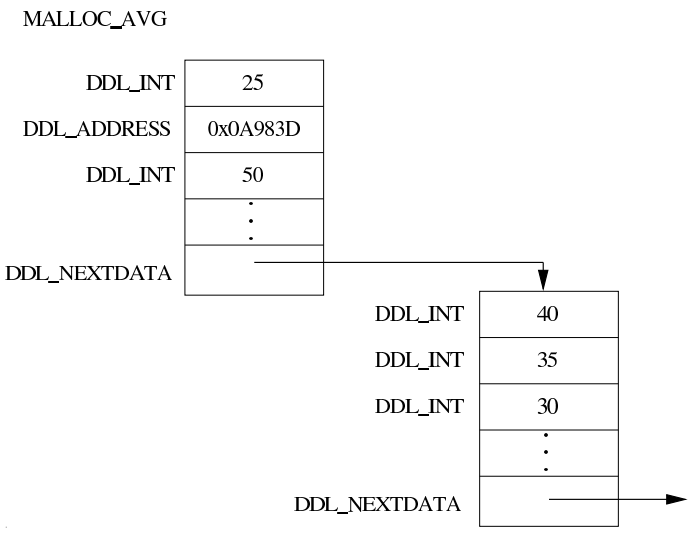


Figure 4.2: An example of a pattern and the corresponding message data.

by the sender of the message. Thus, a sequence of datablocks can be formed using DDL_NEXTDATA each of size *MAX_DATAVALUES*. Releasing memory allocated by every such message is a responsibility of the sender of the message. The Event Dispatcher invokes a function provided by the sender of the message upon completion of processing of the message.

For example, imagine an introspection tool monitoring memory allocations performed by the running application. This tool computes an average size of dynamically allocated memory per invocation of *malloc()*. This tool also produces a notification message after every 50 invocations of *malloc()* library function. Every such message has a name, followed by the average size of the dynamically allocated memory of all the previous calls to the *malloc()* function, followed by the address of the largest allocation, followed by a sorted list of last 50 memory allocations in a decreasing order. A typical message pattern for such a kind of message would be: "MALLOC_AVG DDL_INT DDL_ADDRESS DDL_INT .... DDL_INT DDL_NEXTDATA DDL_INT .... DDL_NEXTDATA DDL_INT." A

typical instance of a message corresponding to the output pattern is shown in the Figure 4.2.

MALLOC_AVG represents the name of message. The listening tool must match this name exactly in its input pattern.[1] The first occurrence of DDL_INT represents the average size of dynamically allocated memory. The DDL_ADDRESS represents the address of the biggest allocation. The next occurrence of DDL_INT represents the size of largest memory allocation. DDL_NEXTDATA represent the pointers to additional block of data allocated by the sender tool.

Thus using the "DDL data types" a variety of message patterns can be constructed to carry a variety of different types of data values.

### 4.3.3 The Registrar

Before a tool can send or receive any messages, its intent to do so must be registered with the Registrar. The Registrar reads a file called *toolfile* and loads all the tools listed in the file. Physically, a tool is a shared object file (.so file). The Registrar initializes every tool by invoking a fixed interface exported by every tool. The Registrar invokes the *tool_init* function of every tool and allows the tool to initialize itself. The Registrar assigns an identifier to every tool and every tools is expected to use the same identifier while communicating with the Registrar. The tools register patterns of messages with the Registrar during initialization. Dynamic (during tool execution) registration is also supported. Unregistration and re-registration are also supported. An instance of the Registrar is passed to every tool to invoke services provided by the Registrar. An instance of the Router can also be obtained from the instance of the Registrar by a simple method invocation.

The Registrar maintains a so called *ToolCommunications* [10,11] structure to maintain a list of active tools and their respective patterns. This data structure

[1] All the registered output message patterns are available to the interested listening tools through a simple API provided by the Registrar.

of the Registrar adheres pretty closely to the *Information Structures* described in the Layered Operational Model [10, 11] described by Harvey and Marlin.

### 4.3.4 The Router

The Router's purpose is to receive messages from the informer tools and deliver it to the listener tools. As tools register input/output message patterns, the Registrar instructs the Router to compute the communication bindings between the tools. The Router uses the *ToolCommunications* structure defined in the Registrar to compute the bindings. For every informer tool, the Router computes a list of listener tools interested in that type of messages. The binding happens when the input and the output pattern names match. When a tool sends a message, it gets added in a queue. The Router picks the messages in the queue and delivers them to the listener tools.

An important aspect of the operation of the Router is that, it runs in a separate thread called the *dispatch* thread. The Router supports *active* type of message delivery model. In the active model, listener tools register input patterns and defines a function to receive messages corresponding to those patterns. The dispatch thread invokes the interface exported by the tool. This interface consists of three functions: *notification_receive*, *request_receive* and *reply_receive*. These functions are also referred as the "receive" functions. The active model supported by the Router is the key to the event-based operation of ETF. The Router accepts request type of messages only if there is a tool registered to accept that type of request. If more than one tool can process a particular request, then the Router dispatches the request to next interested tool only if all the earlier tools fail to process the request. If none of the tools could process the request then, the Router delivers a *REQUEST_FAILURE_NO_REPLY* type of reply to the requesting tool. Thus ETF generates no more than one reply per request. A successful reply

generated by any tool can be delivered to any other tool interested in that type of reply. This may or may not include the tool which generated the request.

In many systems, the functionality of the Registrar and the Router is incorporated into a single component: the ORB of CORBA and the broadcast message server of FIELD are two examples. In ETF, however, the Registrar and the Router have different semantics to justify their separation. Registration is the act of obtaining permission to communicate whereas, routing is the act of carrying out communication. The Router deals with messages whereas, the Registrar deals with message patterns. Registration occurs much less frequently than sending messages.

### 4.3.5 The Redirection Library

The Redirection Library is a the largest part of ETF (excluding the linker). The Redirection Library offers different services to the tools. Table 4.2 shows the Redirection Library API functions and the service offered by each function. These services are integrated into ETF. Users can access these services by sending requests corresponding to it. The Redirection Library can stand alone as a part of ETF and is capable of intercepting the application link resolution process based on the user provided redirection specification. User can provide redirection specifications in a plain text file. The Redirection library interprets this file and remembers the redirection requests enlisted in the file. It redirects the listed symbol when it receives the notification of the event of resolution of link of that particular symbol.

The basic functionality offered by the Redirection Library is symbol/function redirection. For example, a call to *printf()* from *main* can be redirected to *wrap_printf()* defined in some library say, wrapper.so. When a library or the main program invokes a dynamically linked function, it is searched in the dependent shared libraries and resolved before the actual invocation. The Redirection

Table 4.2: API functions exported by the Redirection Library

| API function name | Description |
|---|---|
| redirect_symtosym | Redirects a symbol to another symbol |
| redirect_symtotable | Redirects a symbol to a table |
| redirect_ignore_next | Ignores the next linkup request for a certain function name from a certain library |

Library grabs this opportunity to redirect the original symbol to a different symbol if the user wants it to.

Table-based redirection is another feature supported by the Redirection Library. Table-based redirection allows using a *jump-table* type of redirection. Basically, symbol lookup is redirected to the symbol representing the beginning of the table, and then *get_offset()* function is used to add an offset to the symbol's address. Detailed description of Table-based redirection is given in section 3.4

A message passing layer sits on top of the Redirection Library API. The message passing layer interacts with ETF. It registers two request input patterns: REDIRECT_SYM2SYM and REDIRECT_SYM2TABLE. Tools can communicate with the Redirection Library in the form of requests of above two types. The Redirection Library sends replies, namely, REPLY_SYM2SYM and REPLY_SYM2TABLE on successful processing of the redirection request. Apart from accepting requests from the other tools, Redirection Library accepts three types of notifications generated by the Event Dispatcher: REDIRECT_LOOKUP, REDIRECT_DEFINITION and REDIRECT_OFFSET. It uses these notifications to build its internal data structure of links and definitions.

#### 4.3.5.1 Signal-based Redirection

During the early phases of the project there was no support for active tools and all the tools including the Redirection Library were just passive tools listening to the link resolution events. There was no way to regain the application thread of

control once the callback hook has returned. In order to regain the control, OS's signal mechanism is used. With the support for active tools now, need of signal-based redirection has reduced, if not vanished. Though tool developers should not depend on signal-based redirection anymore, the technique has been documented here.

When initialized, the Redirection Library installs a signal handler to act upon the SIGUSR2 signal. Upon arrival of the USR2 signal, it reads the specification file pointed by the LD_REDIRECT environmental variable, interprets the LINK and/or TABLE specifications given in the file and issues corresponding API calls to carry out the operation. Practically, it is equivalent to calling the API functions of the Redirection Library. The USR2 signal can be sent using the *kill* command. For example, "kill -USR2 pid." It is very important to note that the specification file must be updated each time before the signal is sent. If there are two conflicting specifications (same symbol getting redirected to two different symbols), effectively, the Redirection Library *remembers* only the one which comes later in the specification file.

#### 4.3.5.2 Scripting Support Using Tcl

This is an esoteric feature that allows dynamic analyses to be written at the scripting level. Scripting allows new dynamic analysis ideas to be prototyped in a high level scripting languages (Tcl), and enables even project-specific analyses to be developed cost-effectively.

Tcl support is fully integrated into a version of the Redirection Library. It supports Tcl-based wrappers – that is, wrapper functions written in Tcl. To do this, C-code wrappers that call the Tcl functions are automatically generated from a prototype-like definition. Moreover, Tcl scripts can invoke the services provided by the Redirection Library. To enable this, Tcl has been extended to provide new *commands* corresponding to the API provided by the Redirection

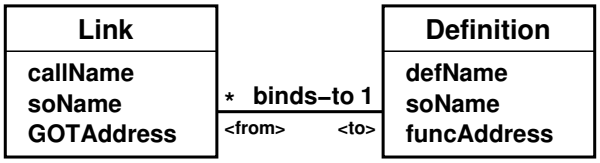


Figure 4.3: Link and Definition UML.

Library and the LinkDef library. This task is largely automated by SWIG [24], a Simplified Wrapper and Interface Generator. SWIG is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and Tcl. Though, at the time of this writing complete Tcl support has not been integrated into ETF, it can be done using a dedicated proxy tool. A separate version of the Redirection Library has been developed which allows dynamic analyses to be written in Tcl. It is not a part of ETF as of yet. Chapter 6 talks about the the Tcl integration into the Redirection Library in great detail.

#### 4.3.5.3 Centralized Link/Definition/Redirection Management

Along with the link interception, The Redirection Library maintains an internal data structure of resolved symbols (definitions), and the bindings or links that refer to them. Maintaining this information during the runtime of the program allows developers to support dynamic program evolution through runtime link modification (redirection). This is done in the Redirection Library and is made available to other tools through the message passing layer. Since this capability is generally useful, it should not be re-built in every tool.

Figure 4.3 shows how the links and definitions relate to each other and the information that uniquely describes each of them.

The link and definition structure is implemented using two hash tables with buckets for fast lookups. The hash tables along with the operations on them such as add and erase, make up the LinkDef library. The LinkDef library also

provides iterators to traverse through all the link and definition entries. A link is identified by the caller library name and the function to be called. A definition is identified by the function name and the library name in which it is defined. Therefore, a link holds the address of the GOT (Global Offset Table) entry and the definition holds the address where the function is loaded in memory.

In order to modify a link, the address that is in the jump table entry needs to be changed to the address of some other function. All the subsequent calls through that link will be directed to the new function. Note that jump tables are allocated per shared object (the main program and other shared libraries), and so these calls are from all the call sites in the shared object whose link is just modified. Thus, the granularity of program evolution is at the shared object level.

## 4.4 Tools

Tools are analogous to browser plug-ins which fit into ETF when they conform to the required interface. Tools can be viewed as clients of ETF. Tools are of two types, an informer tool and a listener tool. A tool can be both an informer as well as a listener. Tools are implemented as dynamically linked shared object files (.so files). The Registrar reads a *toolfile* during initialization and allows these tools to initialize themselves. The Registrar invokes *mastertool_init()* (for the master tool) and *tool_init()* (for other tools) to initialize them. Every tool shared object has to export a fixed, predefined interface. The Event Dispatcher and tools interact with each other though this fixed interface and of course, the Event Dispatcher API. Depending upon the types of input patterns registered by a tool, it needs to define some or all of the following functions: *notification_receive()*, *request_receive()* and *reply_receive()*. These functions are also referred as the "receive" functions.

The Redirection Library is called the master tool. Master tool is the one which has the authority to redirect a symbol during link resolution. For more information on the Redirection Library see the section 4.3.5.

Shared object files in which, tool modules are defined must be listed along with their absolute path in a *toolfile.* The Registrar reads this file. The information concerned with the interaction of the Event Dispatcher and tools is given in section 4.3.1.

### 4.4.1 Tool Events

An informer tool detects system events which is outside the scope of ETF. An informer tool constructs a message corresponding to the event and it is then delivered to the Event Dispatcher for a broadcast. Before a tool can send a message, it must register an output pattern corresponding to the message. A set of messages correspond to a registered pattern. As mentioned earlier, the message must conform with the message pattern and the data types exactly. In order to receive the events corresponding to a output pattern, the listener tools must register an input pattern having the exactly same name. Only then the tools are allowed to communicate. The Router delivers the message to all the interested listener tools by invoking the "receive" functions of each tool. In this way, the broadcast of a message causes an "implicit invocation" of the "receive" functions of the other tools. The informer tool as well as the listener tools do not know how many tools are getting informed about the event.

This allows the constructive ability to forge new capabilities without rewriting tool capabilities. The main issues in supporting the tool events are descriptive power and efficiency of delivery. While link requests and reconfiguration events and the like are not occurring too often, if a tool decides to publish events for every call to some often-used function, or for every read of some variable, efficiency does become a concern.

Because the tools all share the same process space, the current design is to have a read-only event description structure whose reference is passed to the event handler: the "receive" functions, along with a reference to the event data. In this way the event data does not need parsed every time, only traversed to extract the information desired by any given tool.

### 4.4.2 Tool Conflicts

One issue with allowing multiple tools to operate is handling conflicts in their requests on application monitoring or management.

In monitoring, two tools might wish to wrap the same function. If the wrappers are read-only, they can be stacked and will not interfere, other than with more overhead. If tools install a wrapper that might modify parameters or return values, this would possibly conflict with another tool's needed monitoring or management. In run-time modification, two tools might ask to modify the same link in different ways; or one tool might request to modify a link that another tool has already wrapped for monitoring.

At the time of this writing tool conflicts are not handled in ETF. One way to handle these conflicts is to require tools to declare a priority for themselves, and to require tools to declare whether a link redirection will be a read-only wrapper that will support the original functionality. If tools are only installing read-only wrappers, allow these to be stacked. Otherwise, higher priority tools will supersede requests by lower priority tools. Tool conflicts might also be treated as an error.

### 4.4.3 Tool Threading and External Tools

The dynamic linking foundation, and the framework on top of it, necessarily sits within application threads. In regular dynamic linking, the thread initiating the as-yet-unresolved call implicitly invokes the dynamic linking func-

tionality, which resolves the needed symbol and then jumps to the actual function. While extending the linking functionality, adding wrappers or intermediaries between required and provided services, more overhead is added directly into the application threads of control.

While there is no way to completely avoid overhead within the application threads, ETF supports the creation of in-thread tool proxies, which then interact with separately threaded tools. In this manner, significant tool functionality can be added without "unneeded" application slowdown. Obviously, some monitoring and management tools will need to control the execution of the application, and even force synchronous interaction (such as stopping an invocation until security checks on parameters are done), but other tools, such as visualization tools, that have external needs asynchronous to the application computation can indeed be constructed on top of ETF.

Tool proxies can also be used to communicate with tools external to the application process. Whether the tool is too heavyweight to combine with the application or there are existing tools that can take a data feed from another system, proxy tools on top of ETF can be built that respond to events by passing them on through IPC or other communication facilities. One possible use of such a capability is the remote monitoring and management of a system not originally built to support such.

## 4.5 The Thread Model

The modifications to the dynamic linker and ETF are thread-safe. ETF and the Redirection Library can handle multiple applications threads. Moreover, ETF itself creates a separate thread called a *dispatch* thread which actively invokes "receive" functions defined by tools to deliver messages to the listener tools. It is guaranteed that the dispatch thread is the only thread which invokes these functions. Tools themselves can create threads and can assign variety of task

to them, such as a GUI. There are no restrictions imposed by thread model on tools except an obvious restriction that tools may not wait indefinitely in the "receive" functions such as, a call to *scanf()* with no input. This will stop the entire event processing mechanism and events generated by the application such as, link resolution events will pile up in the message queue.

The thread model becomes important in case of the synchronous messages. If a synchronous notification is sent by any thread except the dispatch thread, ETF suspends the execution of that thread. It is only when the message is delivered to the recipient tool that the suspended thread resumes its work. In case of synchronous requests, the suspended thread resumes only when a valid (success/failure) reply is delivered back to the requesting tool. That means, the tools making requests should expect to get their *reply_receive()* function invoked while *send_message()* function has still not returned.

An interesting scenario arises when a tool wants to send a synchronous message from within the "receive" functions, i.e. in the dispatch thread. Even in this case, the *send_message()* function does not return as long as the message is not delivered. But this happens without suspending the dispatch thread. This special case is handled using recursion. *send_message()* function invokes *dispatch()* routine recursively and returns only when all the messages in the queue are delivered to their respective recipients. Therefore, when the dispatch routine is invoked recursively, the other application threads should not be allowed to insert new messages into the message queue. Solving this problem using recursion gives rise to another scenario: infinite circularity.

### 4.5.1 Infinite Circularity

Circularity can occur in event driven systems like ETF when two or more tools send synchronous requests in a cyclic manner to process another synchronous request. For example, Tool T1 sends a synchronous request to tool T2 and while

processing the request, T2 sends another synchronous request to T1. And in turn T1 again generates the same request to tool T2. If such a chain of request has no end then it is termed as an infinite circularity. This situation is analogous to deadlocks(or more accurately, livelocks). A pile of unprocessed synchronous requests goes on increasing and none of the tools can do useful work. This situation can occur in ETF if programmers are not careful. Because ETF deals with the nested synchronous requests using recursion, if proper care is not exercised then it will soon consume the entire available stack and bring down the system. Therefore, ETF does not allow circularity to go beyond a certain hard-coded limit known as *the circularity threshold* (say 50). If recursive calls go beyond the circularity threshold then ETF simply drops the latest synchronous request and reports a failure to the requesting tool. The recursion unwinds afterwards and all the earlier tools can resume their operations.

# 5 DEPLOYING ARCHITECTURE IDEAS USING ETF

The area of software architecture codified the ideas of specifying and reasoning about the large-scale structure of a system. Central in this work are the notions of components and connectors. While much of the architecture work delves deeper than these two simple ideas, the notion of connecting up components is core [16].

In traditional programming languages, these ideas are still foreign. Components in programming languages, be they functions or classes or packages, are not able to refer to their external dependencies using their own internal namespace. Connections are bound by global name agreement, because the undefined symbol from one object is resolved by finding the exact same symbol in another object. This makes controlling the global namespace very important.

Dynamic linking, being an offshoot of traditional program linking, has taken the same view of a system that programming languages have. The ETF tools break that barrier and open up the linking process to allow new mechanisms for system composition.

Using the Dynamic Dynamic Linker (DDL) and ETF on top it, one can view a shared object's external symbol dependencies as locally named port declarations. The dynamic linker's job, then, is to bind these ports with other locally named ports on other shared objects, using some type of connector. The ability to bind a port at link time to a different named port is provided by DDL. ETF builds some reusable functionality on top of this such as, rebinding the ports at runtime, binding multiple required ports to a single provided port. ETF also makes this functionality very easy to use.

The simplest connector is a one-to-one null connector, which would be accomplished by simply replacing the undefined symbol being looked up with a

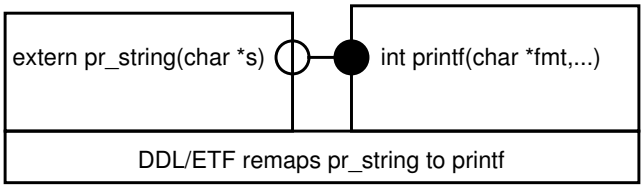


Figure 5.1: Direct null connector by re-mapping symbols.

different one, namely the symbol from the shared object providing the service. This is shown in Figure 5.1. Thus, the undefined symbol becomes a required port name, and the exported symbol becomes a provided port name, and ETF enables the mapping between them. Of course, both ports need to have the exact same call and invocation format in order for this mapping to work.

True architecture support necessitates the capability of supporting complex connectors. Such connectors can have computational capabilities, whether to transform some data, enforce contracts, or handle incompatibilities between endpoint interfaces. A complex connector can almost be viewed as a component itself, but the difference is generally that the connector does not embody application logic.

In ETF, complex connectors must be embodied in some functional code, and have their own provided ports and required ports. To insert a complex connector using dynamic linking and ETF, the symbol re-mapping is done in such a way that the connector is placed between the required and provided ports of the application components. This is shown in Figure 5.2.

## 5.1 Dynamic Reconfiguration

Although the mechanisms supporting dynamic linking have been used up to now in a static fashion—that is, linking is done once and is stable for the rest of the execution—it does not need to be limited to this.

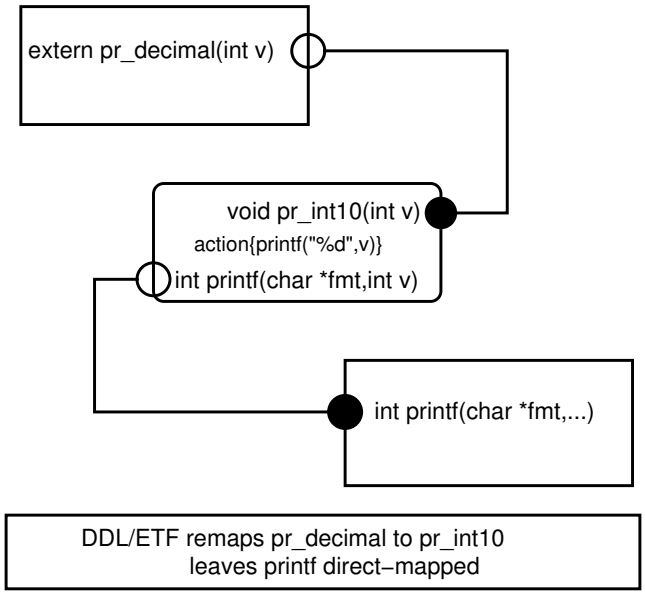


Figure 5.2: Complex connector by interposing a connector component.

The basic support that shared objects give the dynamic linker is a *jump table* that is filled in by the dynamic linker with addresses of resolved symbols. This jump table serves as a centralization point for external references, and thus it can also serve to allow for dynamic reconfiguration. Each shared object has its own jump table, and thus is independently (re-)configurable from the other shared objects.

In ETF, information is maintained on all link sites (entries in jump tables) and all symbol definitions, and the current associations between the two. This allows developers to dynamically change any link at any time during the system's execution. This moves the dynamic linker framework one step closer to being a dynamic component management framework.

Using this capability, monitoring capability can be inserted and removed. It can add a new version of a shared object and update the references and can support other forms of dynamic systems management.

# 6 REDIRECTION USING A SCRIPTING LANGUAGE: TCL

Some situations call for dynamic analyses to be written at the scripting level. Scripting allows new dynamic analysis ideas to be prototyped in a high level scripting languages (Tcl), and enables even project-specific analyses to be developed cost-effectively. Scripting languages are well suited for rapid prototyping where performance is not a concern. Therefore, a scripting language support for runtime monitoring and introspection tasks may save the developer from the drudgery of writing analysis tools in C. Ideally, a developer would want to quickly get away with a short script for program analysis, identify the points of improvements in the system and get back to the job of system development. With this idea in mind, scripting support has been developed for Tcl on top of the existing Redirection Library.

An important point to be noted here is that, Tcl scripting support work is independent of the ETF. Tcl support can be integrated into ETF by using a dedicated proxy tool developed in C. This tool can invoke methods written in Tcl using Tcl C Extension API. Following discussion talks about a stand-alone version of the Redirection Library with Tcl support built on top of it.

## 6.1 Selection of the Scripting Language

There are several scripting languages to choose from for such an extension and embedding task. Python [19], Ruby [21], Lua [14], Tcl [25] are a few of them. The selection of language was based on the following four criteria.

- The thread model supported or imposed by the language: This is important because this system should be usable in multi-threaded environment also.

- The ease of extending the language using C and ease of embedding it in an application written in C: Both the models are important here to enable analysis to be done entirely in a scripting language.
- The ease of development of a GUI.
- The active user community of the language and the documentation availability.

Python, Ruby, Lua and Tcl were compared on these grounds and Tcl was chosen among them. The reasons in favor of Tcl can be enumerated as follows:

- Beginning with the 8.1 release, the Tcl core is thread safe. It supports Tcl into multi-threaded applications without customizing the Tcl core.
- An important constraint of the Tcl threads implementation is that only the thread that created a Tcl interpreter can use that interpreter. In other words, multiple threads can not access the same Tcl interpreter. This constraint was not prohibitive. Tcl thread constraints fit well into the thread model as long as Tcl interpreter is used by the same thread every time.
- Tcl was built to be extended and embedded in system languages. The API is well documented. Tcl/Tk allows rapid construction of GUIs. Moreover, there are open source, interface generators such as SWIG [24] which connects programs written in C and C++ with scripting languages such as Tcl in an almost fully automatic way. For these reasons, Tcl outweighed the other languages.

## 6.2 Building the Tcl Scripting Layer

The task was to incorporate two different models of interaction of an application and Tcl: the extend model and the embed model. The extend model is

about invoking the Redirection Library services in the form of new Tcl commands. The embed model is about invoking Tcl methods from a C program. The Redirection Library and the *LinkDef* library exports the services provided by them in the form of an easy to use API. SWIG [24], a Simplified Wrapper and Interface Generator was used to generate Tcl wrappers corresponding to every API function exported by the Redirection Library and the LinkDef library. The advantage of using SWIG is that new API functions, if added, can be quickly extended as Tcl commands. Moreover, the wrappers SWIG generates are type safe and therefore add an extra layer of security.

The other part, embedding Tcl, was little tricky due to the fact that the Tcl threads implementation allows only the thread that created a Tcl interpreter to use that interpreter. To overcome this limitation, a dedicated *tclthread* is created which is responsible for invoking Tcl methods from within a C program. Tcl has a nice, robust API to embed Tcl in a C program. Every time when the application thread wants to invoke the Tcl methods corresponding to hooks in the modified linker, the Redirection Library suspends the execution of application thread and notifies the *tclthread* to invoke the corresponding method. Data is passed back and forth between two threads in a thread safe way.

This version of the Redirection Library, called "Tcl Redirection Library" is packaged separately and can be used for quick program analysis at the scripting language level.

# 7 RELATED WORK

The DITools project [22] is the closest related work to the DDL project. They used a similar approach to link interception and modification, and supported redirecting a link to a wrapper and also an event notification mechanism where each monitored call was not wrapped but did generate an event to a fixed-interface callback. It does not appear that they addressed the issues surrounding C++, nor did they do non-function symbol resolution nor runtime link modification.

Ho and Olsson [12] describe *dld*, a tool for "genuine" dynamic linking. Their tool provides the capability to load and unload shared libraries, breaking links when a library is unloaded and relinking them to new code when new libraries are loaded. However, it does not appear that they ever supported redirection of links to different symbol names.

Hicks et. al [9] work on binary software updating from a formal perspective. Their methods use typed, proof-carrying assembly code from which they can verify that an update will be safe. Their infrastructure includes special languages and compilers to generate the annotated assembly code, and a runtime framework that uses it.

Additional systems that provide instrumentation capabilities on executable binaries exist. Dyninst [4] can patch custom code into pre-existing executable code, and has provided a platform for several research tools. Valgrind [27] provides a complete simulated CPU and execution space to the program under inspection, and is extensible, thus allowing new dynamic analyses to use it as a foundation.

There is much work in dynamic introspection and modification of Java programs, but it is in a very different environment than this work. Some representative references are [2, 7, 15, 18].

In the commercial world, .NET seems to offer extended capabilities beyond simple dynamic linking [17]. .NET "solved" the DLL incompatibility problems by requiring shared objects to reference exact versions of other shared objects, and even allowing multiple shared objects in an application to use different versions of some other shared objects. External rules can allow a different version to be declared compatible, so that application upgrading is possible, but tightly controlled. .NET also has strong introspection capability, and a debugging API that gives control over an application, so it is possible that they support a large amount of what DDL/ETF does. However, no indication has been seen that .NET would allow programmatic symbol redirection (although the debugger API does support run-time CLR binary editing), and there are some indications that .NET still suffers from some versioning problems [6].

# 8 LIMITATIONS AND FUTURE WORK

Using shared objects and dynamic linking does have its limitations in terms of fully deploying component-based system ideas.

For one, dynamic linking occurs within a single process. As such, shared objects are not truly protected from each other, and many well-known accidental or malicious anomalies can occur. Shared objects have implicit access to each other's address space, and although the linker can arbitrate between symbol-based access requests, it cannot prevent or monitor implicit or accidental access.

The invocations between shared objects, since they are essentially just function calls, are synchronous. While ETF framework can build on top of this to implement asynchronous interaction, this would have to be designed into the shared objects that used it. Legacy shared objects, expecting synchronous interaction, would probably not support such large interaction changes.

Dynamic linking, like static linking, is done without regards to type signatures. It is assumed that the compiler has checked this already, and so when resolving symbols, only the symbol name matters. By opening up the dynamic linking process and allowing diverse binding possibilities, interface checking (at least type checking but also perhaps semantic checking) becomes important.

Currently this issue has not been tackled. However, there are several possibilities. Most executable formats are extensible; debuggers take advantage of this to include much information about a program, including type information. A similar approach can be taken to include only the type information about the external interface of the shared object. An alternative would be to require a separate specification of the interface type signatures, similar to a header file. Methods such as how C++ embeds type information into symbol names could also be used, although this would need to be augmented with type relations.

Granularity at which DDL/ETF performs could be a limitation for certain types of monitoring needs. DDL/ETF functions at function call granularity. Some program monitors need finer grain access, e.g. to monitor loops or conditionals.

At the time of this writing, ETF supports exactly one dispatch thread and a message queue. In a multi-threaded application, single dispatch thread might get overwhelmed by a flood of asynchronous messages generated by several application threads. If the message queue length is not controlled then it is possible that dispatch thread might never get enough CPU share to cope up with the ever increasing length of the message queue. One way to handle this situation is to continually watch the size of the message queue and suspend the execution of the application threads till the message queue size becomes manageable.

Finally, not all shared object interactions need to be mediated by the dynamic linker. Implicit interactions might occur. Addresses, including functions pointers, can be passed between shared objects and then used to directly access some "unknown" exported behavior or data. Some limitations on what a shared object could do to be considered a manageable component would be required, and most of these could probably be checked with some static analysis techniques.

# 9 CONCLUSION

This work has describe an extensible event-based mechanism for manipulation and monitoring of an application built on shared libraries through interacting with the dynamic linker.

Shared, dynamically linked libraries have been around for quite some time, and yet they have been ignored as a platform for CBSE (Component Based Software Engineering) ideas. This ubiquitous platform can support much more dynamic behavior and component management than it currently does. The ultimate hope is to influence the direction of future dynamic library infrastructure to include the support needed to make shared libraries true manageable components.

As with any opening of an application framework, security does become a concern. However, since some dynamic library platforms already allow redirection through the preload mechanism, this work can be seen as *enabling* further security measures rather than opening new holes. An authentication mechanism can be implemented to ensure that shared objects are from a trusted source. In this way management and manipulation can be allowed and at the same time have confidence that the manipulation is not being done by a malicious tool.

The current focus is in the deployment of the HERCULES framework on top of DDL/ETF, but DDL/ETF is also being used for dynamic analysis work (especially scripting language support), for dynamic behavior adaptation, and other applications. HERCULES is a framework for reliable evolution of a system where multiple versions of components can be active in a system at any given time [5]. An early prototype of this is already working, where components are C++ classes.

Presently, DDL and ETF are stable and both are freely available for research use at http://www.cs.nmsu.edu/please/ddl/index.php.

REFERENCES


[1] N. Abbas, S. Tambe, R. Srinivasan, and J. Cook. Using DDL to understand and modify SimpleScalar. In *Proc. 2004 Working Conference on Reverse Engineering*, page to appear, Oct. 2004.

[2] M. Dahm, 2002. The Byte Code Engineering Library for Java. http://jakarta.apache.org/bcel/.

[3] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, Alexander E. Wise, 1996 A framework for event-based software integration In *ACM Transactions on Software Engineering and Methodology (TOSEM)* Volume 5 , Issue 4 (October 1996) Pages: 378 - 421

[4] B. Buck and J. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000. www.dyninst.org.

[5] J. Cook and J. Dage. *Highly Reliable Upgrading of Components.* In proceedings of the 21st International Conference on Software Engineering, (May 1999), pages 203-212

[6] A. Berglas, 2004 Warning: .NET Hell and Version Control, unstable, irreproducable bugs. http://www.codeproject.com/Purgatory/DotNetHell2.asp.

[7] S. Eisenbach and C. Sadler. Changing Java Programs. In *Proceedings of the 2001 International Conference on Software Maintenance*, pages 479–487, Nov. 2001.

[8] David Garlan and Mary Shaw. An Introduction to Software Architecture. *CMU Software Engineering Institute Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21.* January 1994

[9] M. Hicks, J. Moore, and S. Nettles. Dynamic Software Updating. In *Proc. 2001 ACM Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[10] J. G. Harvey and C. D. Marlin. 1996 A layered operational model for describing inter-tool communication in tool integration frameworks. In *The Australian Software Engineering Conference*, pages 55-63.

[11] J. G. Harvey and C. D. Marlin, 1997. Comparing inter-tool communication in control-centred tool integration frameworks. In *8th Conference on Software Engineering Environments*, pages 67-81.

[12] W. Ho and R. Olsson. An Approach to Genuine Dynamic Linking. *Software Practice and Experience*, 21(4):375–390, 1991.

[13] John R. Levine, 1999 Linkers and Loaders. published by Morgan-Kauffman in October 1999.

[14] The Programming Language Lua, http://www.lua.org.

[15] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proc. European Conference on Object-Oriented Programming*, pages 337–361, 2000.

[16] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[17] Microsoft .NET Framework Developer Center, 2004.

http://msdn.microsoft.com/netframework

[18] A. Orso, A. Rao, and M. Harrold. A Technique for Dynamic Updating of Java Software. In *Proc. 2002 International Conference on Software Maintenance*, pages 649–658, Oct. 2002.

[19] Python programming language, http://www.python.org.

[20] Reiss, S. P. 1990. Connecting tools using message passing in the FIELD environment. IEEE *Software*, 7, 4 (July), pages 57-67.

[21] Ruby scripting language, http://www.ruby-lang.org/en.

[22] A. Serra, N. Navarro, and T. Cortes. DITools: Application-level Support for Dynamic Extension and Flexible Composition. In *Proc. 2000 Usenix Technical Conference*, pages 225–238, June 2000.

[23] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. *Technical Report 1342, Computer Sciences Department, University of Wisconsin*, June 1997.

[24] Simplified Wrapper and Interface Generator (SWIG). http://www.swig.org.

[25] Dr. John Ousterhout, 1988. Tcl, Tool Command Language. University of California Berkeley, 1988.

[26] Thomas, I. and Nejmeh, B. A. 1992. Definition of tools integration for environments. IEEE *Software*, 9, 2 (March), pages 29-35.

[27] J. Seward. Valgrind, an Open-Source Memory Debugger for x86-Gnu/Linux. Technical report. valgrind.kde.org.

[28] A. I. Wasserman, 1989 Tool integration in software engineering environments. *The International Workshop on Environments (Software Engineering Environments)*, volume 647 of Lecture Notes in Computer Science, pages 137-149. Springer-Verlag, Berlin, September 1989. Chinon, France.