SUPPORTING ON–LINE DISTRIBUTED MONITORING AND DEBUGGING

VITOR DUARTE[†], JOÃO LOURENÇO[†], AND JOSÉ C. CUNHA[†]

Abstract. Monitoring systems have traditionally been developed with rigid objectives and functionalities, and tied to specific languages, libraries and run-time environments. There is a need for more flexible monitoring systems which can be easily adapted to distinct requirements. On-line monitoring has been considered as increasingly important for observation and control of a distributed application. In this paper we discuss monitoring interfaces and architectures which support more extensible monitoring and control services. We describe our work on the development of a distributed monitoring infrastructure, and illustrate how it eases the implementation of a complex distributed debugging architecture. We also discuss several issues concerning support for tool interoperability and illustrate how the cooperation among multiple concurrent tools can ease the task of distributed debugging.

Key words. On-line Monitoring, Distributed Debugging, Tool Interoperability, Software Engineering Environments.

1. Introduction. Everybody recognizes the difficulties in developing parallel and distributed applications and the need for tools that can help the programmer in that process. The observation of the behavior of a distributed application (the target application) plays an important role during its development and also during its execution. Monitoring tools are complementary to the use of performance models and simulation tools, as they allow to obtain information on the real execution of an application. Monitoring is also critically important in distributed applications where unpredictable changes may occur in a system configuration, which may require some reaction to achieve specific goals such as load balancing or fault tolerance.

Many monitoring tools and support systems have been developed in the recent past, usually each one dedicated to a particular purpose, for a specific programming language or library, and for specific operating system and hardware platform. Only few of such tools can be considered of general use, but even those are not easy to adapt to new computing platforms or user requirements.

Recently, there has been an increased use of parallel and distributed computing models in a wide range of applications. This has motivated an increased concern on how to design monitoring and control tools which can be used to support a diversity of functionalities for observation and control of parallel and distributed applications, as illustrated in the following list:

- To analyze application performance;
- To observe application behavior;
- To control the execution of an application;
- To support integrated software engineering environments and collaborative working environments, where multiple concurrent tools must be coordinated, so that global consistency constraints must be imposed.

On one hand, it would not be feasible nor even efficient to try to implement a monolithic system encompassing all of the above uses. Instead, a modular design should be promoted:

- To enable the on-line interaction between a tool user interface and the target application. This should guarantee some degree of independence of the user interface and the tool functionality with respect to the low level system architecture;
- To allow an incremental extension of the environment by providing mechanisms to integrate new tools into the system, each tool providing a specific user interface and service;

[†]Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa ({vad,jml,jcc}@di.fct.unl.pt).

- To support the launching of the required services for observation, configuration, data gathering or specific data processing;
- To allow the modular composition of simple individual tools, as a way to achieve more complex functionalities;
- To manage and coordinate the interactions among multiple concurrent tools, having on-line access to the same target application.

From our past work we have identified the need to develop a distributed framework to support monitoring and control services, that could allow the integration of multiple concurrent tools. From our earlier prototypes of the DDBG [7] and PDBG [8] distributed debuggers, we have identified several requirements to support the above objectives. This has led to the design and implementation of the DAMS [5, 8] architecture and the development of the Fiddle [16] debugging tool.

In Sec. 2 we discuss different approaches for monitoring and control architectures. In Sec. 3, the main characteristics of DAMS are reviewed. In Sec. 4, we focus on the description of Fiddle, and compare to previous work. In Sec. 5 we discuss debugging tools interaction when using Fiddle and how such concepts can be supported in terms of DAMS functionalities, to improve tool cooperation in a parallel software engineering environment. In Sec. 6 we report on the current implementation status. Finally, in Sec. 7, we present some conclusions and ongoing work.

2. Monitoring and Control Interfaces. Traditional solutions had rigid objectives and functionalities, and were tied to specific run-time environments (Fig. 1 *a*). They couldn't use more than one tool at a time, and couldn't adapt to distinct objectives or add new functionalities.

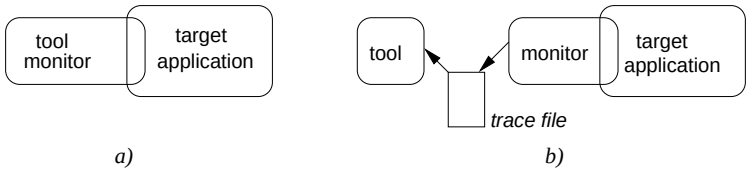

FIG. 1. a) *Monolithic tool;* b) *Trace based tool*

2.1. Tool interaction based on trace files. A simple and very usual approach to achieve tool interaction for monitoring and performance evaluation has been the use of common trace file formats (Fig. 1 *b*). This allows the use of post-mortem visualization and evaluation tools. If new tools are designed compatible with those formats (or if they allow using some conversion step) they can quickly be integrated into the monitoring environment.

The most typical example of a "standard" trace file is the format used by the Portable Instrumented Communication Library (PICL) [9, 28], which has been used by several monitors and tools, directly or with conversion programs. Most of its success is related to its visualization tool, ParaGraph [11], and the more or less neutral semantics of the trace file. The importance of the approach is illustrated by the fact that this trace format has been extended to a new version for the MPI [21] system called MPICL, which uses the standard MPI Profiling Interface to instrument the application.

A more powerful approach was taken by the Pablo [25] project by designing the Self Defined Data Format (SDDF) [1], a meta-format whose philosophy was similar to the one followed later by the XML standard.

2.2. Instrumentation/Control Interfaces. Monitoring systems rely on low level interfaces for instrumenting and controlling a target application. These functions can be supported through several application programming interfaces (API), possibly at different levels in the system (libraries, OS and hardware architecture). The standardization of these interfaces increases the portability of the distributed infrastructure and, as a consequence, of the tools by decoupling the low level dependencies from the monitoring infrastructure. In these approaches, some instrumentation/control functions are separated from the monitoring system, and neutral instrumentation facilities are provided for distinct run-time/OS platforms (Fig. 2).

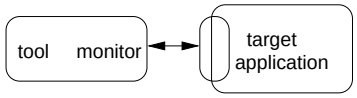

FIG. 2. *Instrumentation and control interfaces*

Within the Parallel Tools Consortium Projects (PTOOLS) there are several efforts targeted at the lower level of the monitoring infrastructure. For example, the Performance API (PAPI) [2] for obtaining the values of hardware counters, which are usually available in modern CPUs. Also the Portable Timing Routines (PTR) [24] for measuring program execution intervals, in terms of wall clock, user CPU, and system CPU times.

One of the most powerful proposals is the DynInst library [3], which was started within the Paradyn project [20], aiming at building a dynamic and configurable monitoring system. The library supports the dynamic code instrumentation of a running process. It uses the OS facilities for inspecting and controlling processes, allowing the user to browse the object code and install code patches at particular points. Those code patches are defined using an abstract description and translated by the DynInst into the native code for the particular architecture where it's running. As several architectures are supported this allows tool portability regarding this aspect.

Such approaches have the benefit of providing more or less standard interfaces supporting low level instrumentation functions, which can then be integrated into full distributed monitoring architectures.

2.3. Monitor/Control APIs and Distributed Monitoring Architectures. The trace-based approach is adequate for the analysis of execution traces after program termination, but it cannot support on-line interaction with a running application. The latter requires specific protocols to be established for asynchronous or synchronous interaction between the tools and the application, libraries, OS and hardware architecture (Fig. 3). This requires a precise interface definition and a flexible architecture for monitoring and control which satisfies the main requirements discussed in Sec. 1.

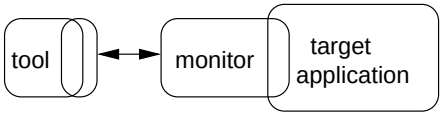

FIG. 3. *Decoupling tool and monitor*

A related effort on improving the accessibility to low level instrumentation and control interfaces is the Dynamic Probe Class Library (DPCL) [22]. DPCL defines an API that uses

DynInst for process instrumentation. DPCL is based on a standard client/server infrastructure that allows an individual tool to interact with several DPCL servers (one on each machine). This allows the tool to control all the processes of the application. Also, each DPCL server can accept more than one client, allowing several tools to use the DPCL infrastructure simultaneously (Fig. 4). There is no support to solve possible interferences and conflicts among the tools, or for coordinating their cooperation. Another example is given by FIRST [23] which relies upon CORBA [10] to implement a distributed monitoring system. FIRST also uses the DynInst library for process instrumentation and uses the PTR routines for time measuring.

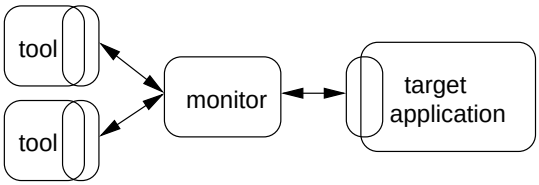

FIG. 4. *Flexible monitor infrastructure*

OMIS [19] is a systematic effort towards supporting a standard monitoring interface and an open framework for developing monitoring tools and services. The OMIS specification defines a generic interface library for inspecting and controlling distributed processes and for the tools to interact with the monitoring system. The inspection can be customized for each particular use/tool, by defining the interesting events and the actions each one should trigger. There is provision for extensions to the supported events and actions. A distributed monitor and control system can support more than one simultaneous client tool and allow some interoperability. An implementation of the OMIS specification was developed as a monitoring architecture (OCM/OMIS) [18]. It has been used as a basis to develop debugging and visualization tools (new versions of the DETOP and VISTOP [27] tools) and to support their interoperability.

In the following section, we describe the DAMS approach for distributed monitoring.

3. DAMS: A Distributed Application Monitoring System. The DAMS [5, 8] approach proposes a distributed infrastructure for monitoring and control of parallel and distributed applications. Its distinctive characteristic is being based on a software architecture which allows a clear separation between the low-level mechanisms for distributed observation and control, and the high-level services provided by monitoring and control tools, such as debuggers, performance analyzers or resource managers (Fig. 5).

3.1. Architecture. Instead of defining a rigid monitoring interface, DAMS only provides the mechanisms to integrate new services into its distributed architecture, handling service identification, registration and localization, communication, services activation and control, and concurrent tool interaction.

The DAMS architecture is neutral concerning the computational model of the target application, and also concerning each tool specific functionalities (Fig. 6). A DAMS configuration is built of a set of services which can be accessed by the tools in order to implement the tool specific functionalities.

In order to integrate a new service into the DAMS a given Service Module must be defined and registered as an available service. From the clients point of view, the Service Module provides an interface to a set of functions (entry points into the module) and is named through an unique global identifier. From the DAMS point of view, the Service Module is responsible

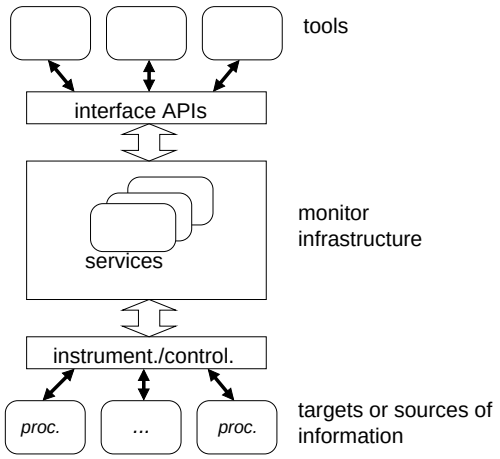

FIG. 5. *Layer view of a monitoring and control environment*

for the implementation of the service which can handle requests from multiple concurrent clients. It is responsible for the interpretation of each request, the supervision of the execution of the required actions, and the sending of the corresponding replies back to the client. For such purpose, a Service Module can rely upon other services, possibly located at remote machines. An internal DAMS communication layer supports the communications between services and with the tools.

3.2. Built-in Services. In order to support the above aspects, the DAMS includes built-in Service Management. It provides functions for registering new services and for service identification and localization. The Service Management allows the tools to request access to the interfaces of the registered services by connecting to instances of the Service Modules. This access can be made through specific client level libraries which may provide transparent user interfaces.

DAMS also offers a built-in Resource Management service that is used to manage the hosts and processes configuration. This includes adding and removing hosts from the DAMS environment, launching processes and getting status information.

DAMS provides a built-in Event Service for asynchronous event notification based on a publisher/subscriber model. This mechanism can be used to detect and react to events generated by the target application, for example, to react to execution level exceptions. This mechanism can also be triggered by the processing of requests made to the DAMS services, when invoked by a client tool, in order to enable asynchronous interactions at the tool user interface level (e.g., non-blocking, event-driven semantics for client requests in a graphical user interface).

3.3. Tool Interaction. Support for concurrent tools can improve the expressiveness of a parallel software engineering environment, by allowing the user to exploit the complementary roles of distinct tools, having access to the same target application. When multiple concurrent tools have access to the same target application there is the problem of interference among tools. At the level of the DAMS architecture, some support for tool synchronization is ensured due to the use of common service modules to access the same target-processes. This corresponds to the so-called *structural conflicts* [27]. Additionally, it may be necessary to ensure consistency among multiple concurrent tools at a logical level. The event mechanism can be

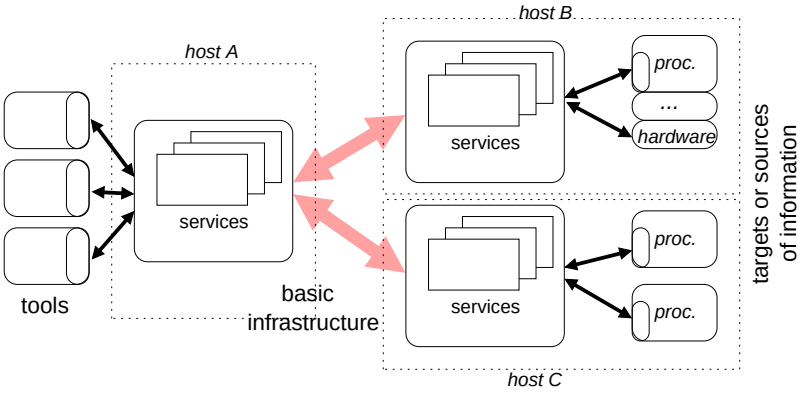

FIG. 6. *DAMS architecture*

used to help the coordination of such concurrent tools. Events generated by a target application can be disseminated to the set of tools which are interested in observing the application evolution, so those tools can coordinate their actions accordingly.

Additionally, tools themselves can explicitly generate events, allowing other tools to be aware and adapt to such events. For example, the user can select in one tool to observe a different aspect of the application, then other tools can follow that change.

3.4. Conclusion. DAMS basic architecture can be used to develop specific services for observation and control. In general, it is possible to develop specific tools and services (e.g., the ones based on Fiddle) in a stand-alone fashion, i.e., without requiring any basic support. However, we claim it is preferable to rely upon a low-level infrastructure providing a minimal set of functionalities for monitoring and control. In such an approach, the tool/service developer can concentrate on specific design issues of each functionality, thus benefitting from increased software development productivity. On the other hand, by having a common underlying platform, it becomes easier to support multiple concurrent tools and manage their interactions. In the following sections we first discuss the functionalities and architecture of a complex debugging service (Fiddle), and then we discuss how it maps into the DAMS framework.

4. Fiddle: a Tool for Distributed Debugging. Some parallel debuggers have evolved into commercial tools, e.g., TotalView [26], made available through parallel machine and tools vendors. The architectures of such parallel debuggers usually follow a monolithic approach, combining a debugging engine and the user interface into a single large program. More flexible parallel debuggers use a client/server model, e.g., p2d2 [12], which separates the user interface from the debugging engine. This approach improves the ability to adapt to different requirements and environments, and to customize the user interface to better fit a specific environment. The generality of the parallel debuggers also use considerably small and dumb agents in each machine node, to support the distribution of the debugging functionalities, relying in a central server to do all the processing.

Prior to describing Fiddle, we will summarize our previous work on distributed debugging and how it motivated the design of this new system.

4.1. Previous Work on Distributed Debugging. Previous work on DDBG [7] allowed to experiment with two tool integration scenarios concerning the distributed debugging ac-

tivity: integration of a Graphical Parallel Application Editor and the Debugger [13]; and integration of Testing and Debugging tools [17].

We describe the main characteristics of each case study, concerning tool interactions and their coordination, the most relevant limitations that were found in the DDBG prototype and the major improvements required.

4.1.1. Visual Programming and Graphical Debugging. In a parallel software engineering environment, a graphical editor for a visual programming language can help the user in the design of a parallel or distributed application. In such an environment, the user develops the distributed application by specifying graphical high-level entities and their composition. A graphical program, consisting of such graphical entities, for example, representing processes and communication channels, will then be automatically compiled into a textual source code which may be amenable to a parallel execution. The usage of a classical distributed debugger, operating with the automatically generated code, provides little help to the user in the understanding of the (graphical) program behavior.

To overcome these difficulties the debugger should allow the user to work mainly with the graphical program constructs and the abstractions that were used during application development. This requires, for example, to highlight the entities in the graphical representation and their corresponding lines of (the generated) source code in the textual program representation, and to allow the user to step through both the graphical entities or the source code.

Within the SEPP/HPCTI [6] European projects, a successful experiment involving the integration of the GRED [13] graphical editor and DDBG was achieved [13]. In this prototype, GRED and DDBG could establish a two-way interaction, corresponding to the invocation of the DDBG debugging methods by GRED, and its replies reporting the changes in the target application state. However, in order to enable asynchronous user interaction at the graphical editor level, it was necessary to extend the GRED/DDBG interaction with an ad-hoc scheme for deferred delivery of the replies from the debugger to the graphical environment.

4.1.2. Integrating Testing and Debugging. A distributed debugger may contribute to the detection, localization and correction of bugs in an application, but still strongly depends upon the user interpretation of program correctness. The use of an interactive *testing tool*, which partially automates the identification and localization of suspect program regions, can improve the process of developing correct programs.

STEPS [15] is a testing tool developed at Technical University of Gdansk, Poland, within the SEPP/HPCTI [6] European projects and is able to identify potential critical program flow paths in a C/PVM program. When integrating such a tool with a distributed debugger, one must ensure that the program will behave as predicted by the testing tool.

Within the above mentioned projects, another successful experiment involving the integration of STEPS and DDBG was achieved [17]. In this prototype, the composition of the testing and the debugging tools allows the following iterative steps:

1. STEPS generates specific testing scenarios;
2. An intermediate tool, Deipa, reads a scenario and builds a semantic tree with its contents;
3. By interacting with Deipa, the user may select one of the points in the scenario (a *global breakpoint*) for testing;
4. Deipa [17] generates a set of debugging commands, which are sent to DDBG, to enforce each target-process to follow a specific path until the global breakpoint is reached;
5. DDBG drives the execution of the target-processes according to Deipa directives;
6. At that global breakpoint (a suspect program location), the user may use DDBG

to enter an interactive debugging session, to inspect and fine control the target-processes;

7. Finally, the testing may be resumed by selecting another point (step 3).

In this case, there is a one-way interaction between STEPS and DDBG, through an intermediate file which contains the testing scenarios. This experiment illustrated the need to include an intermediate tool to manage the interaction between the testing and the debugging tools, and the need of this tool to coexist with other (lower-level, in this case) debugging interfaces.

4.1.3. Major Improvements Required. The above experiences suggested the need to provide support for:

- Asynchronous interactions between the debugger and the other tools, supported by events;
- The inclusion of new tools, to provide complementary functionalities for application development and possibly act as intermediaries between the debugger and the other tools;
- Synchronization of the client tools sharing the access to the target application.

4.2. Overview of Fiddle. The experiments described above have motivated the design of a more advanced debugging tool: Fiddle. Its software architecture was designed to be able to fulfill the major improvements identified above and, being functionally backwards compatible with DDBG, it could replace the latter in those experiments, with benefits.

The main functionalities of Fiddle are (the entries marked with (†) are new to Fiddle):

- *Debugging of multiple target application processes*, executing in the local or remote machines;
- (†) *Debugging of multi-threaded processes*, if such functionality is supported by the node debuggers being used by Fiddle to act upon the target-processes;
- *Simultaneous access by multiple client tools* to the same target application processes;
- (†) *Support for multi-threaded client tools*, to ease the control of the asynchronous interactions between the debugger and those clients;
- (†) *Deferred replies to the services requested* by client tools, based on an event/call-back mechanism, for improved support for asynchronous interactions;
- (†) *Event notification*, providing basic support for tool interoperability and tool co-ordination services;
- (†) *Tool synchronization events*, to support shared views of the target application by the multiple client tools.

Fiddle software architecture improvements (over DDBG) include:

- *A layered software architecture*, to provide limited debugging functionalities with reduced overhead;
- *A many-clients/many-servers model*, with the clients acting as global debugging interfaces, and the smart local servers on each node having full local debugging capabilities;
- *The central server* is used only to support multiple clients simultaneously and to provide some global debugging functionalities.

In its current version, a language-dependent library provides access to Fiddle methods. These methods are categorized as:

- *Management methods.* Observe and/or change Fiddle internal state;
- *Inspection methods*. Observe but do not change the target application state;
- *Control methods.* Change the target application data and/or execution states.

A Fiddle client tool is a possibly multi-threaded program, that was linked to the Fiddle library, and uses Fiddle to interact with the target application. A Fiddle method invocation

follows a *remote procedure call* (RPC) semantics, blocking the calling thread until the service is executed.

Fiddle manages multiple client tool connections simultaneously, allowing them to act upon the same target application. These concurrent tools may provide distinct views of the target application and will, typically, explore complementary approaches to the debugging activity, such as a *source code graphical debugging interface* and a *3D data visualizer*.

4.3. Fiddle Software Architecture. Fiddle is structured as a hierarchy of 5 functional layers, which implement an incremental set of functionalities, as summarized in Tab. 1.

	Layer 0_s	Layer 0_m	Layer 1_m	Layer 2_m	Layer 3_m
Multiple target-processes	Yes	Yes	Yes	Yes	Yes
Multi-threaded target-processes	Yes	Yes	Yes	Yes	Yes
Multi-threaded clients	No	Yes	Yes	Yes	Yes
Number of nodes	1	1	Any	Any	Any
Number of clients	1	1	1	Any	Any
Events and call-back	No	No	No	No	Yes

TABLE 1
Fiddle layers and their functionalities

The interface provided by each layer $\mathcal{L}_i$ is used by the layer immediately above $\mathcal{L}_{i+1}$, in order to implement higher level functionalities. A client tool can also directly use any layer $\mathcal{L}_i$, but this usage is exclusive with the usage of the upper layers $\mathcal{L}_{j_{(j>i)}}$, the lower layers $\mathcal{L}_{k_{(k<i)}}$ are used implicitly by $\mathcal{L}_i$.

• *Layer* 0_s. The software architecture of Layer 0_s is presented in Fig. 7. It provides a function-based interface to access a set of node debuggers. A node debugger may be a text-oriented sequential debugger, such as the GNU GDB or DBX, or any other library with debugging capabilities, such as DynInst [3].

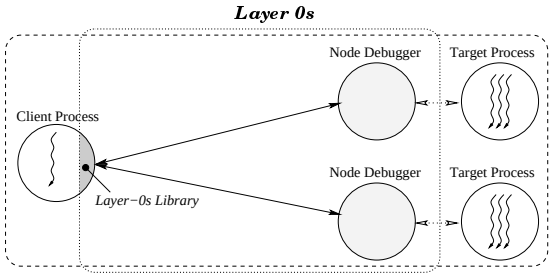

FIG. 7. *The Layer* 0_s *architecture*

This layer manages only one single-threaded client, but is able to control multiple (single- or multi-threaded) target-processes running in the local machine. It allows the starting of new instances of the node debugger as needed, to generate the commands for the node debuggers in the appropriate format, and to collect, parse and extract the relevant data from their responses.

Since Layer 0_s clients are single-threaded programs, the invocation of a method blocks the client until its completion.

• *Layer* 0_m. This layer extends Layer 0_s to support multi-threaded client tools (Fig. 8). Method invocation at Layer 0_m also blocks the calling thread but, as the remaining threads in the client stay active, it may still interact with the user or with Fiddle.

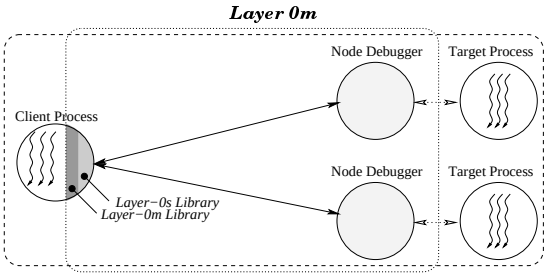

FIG. 8. *The Layer 0_m architecture*

In this way, concurrent requests to distinct target-processes are processed in parallel, while concurrent requests to the same target-processes are serialized and performed one after another.

• *Layer 1_m*. This layer extends Layer 0_m, so that the target-processes can also execute in remote machines, and not only in the local machine (Fig. 9). The software architecture for Layer 1_m contains, in each node, an instance of Layer 0_m and a daemon (*$L0_m$ server*) which is a Layer 0_m client.

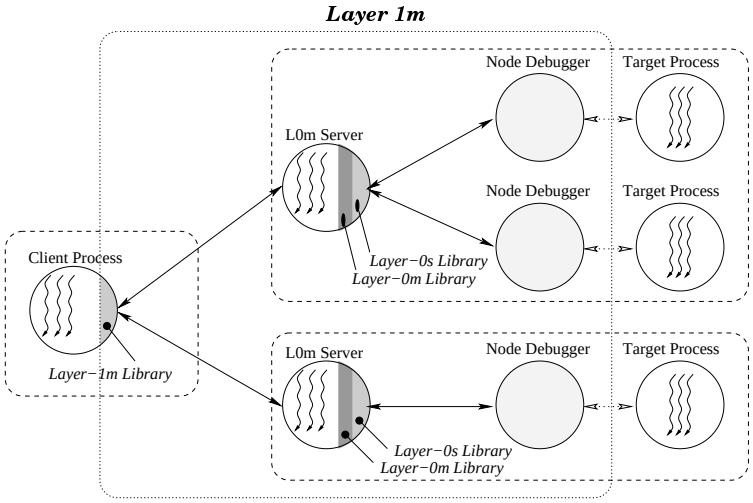

FIG. 9. *The Layer 1_m architecture*

• *Layer 2_m*. This layer extends Layer 1_m to support multiple simultaneous client tools (Fig. 10). These tools may be concurrently issuing debugging commands to the same set of target-processes. The software architecture for Layer 2_m contains an instance of Layer 1_m and a daemon (*$L1_m$ server*) which is a Layer 1_m client. Besides the implicit structural coordination which results from the sharing of the same servers (*$L0_m$ server* and *$L1_m$ server*) when accessing the same target-processes, this layer does not provide any other support for the coordination among client tools.

• *Layer 3_m*. This layer adds event-based *Tool–Fiddle–Target-processes* interactions and call-back capabilities to Layer 2_m. In contrast to the previous layers, the invocation of a method in this layer does not block the calling thread and immediately returns a *request*

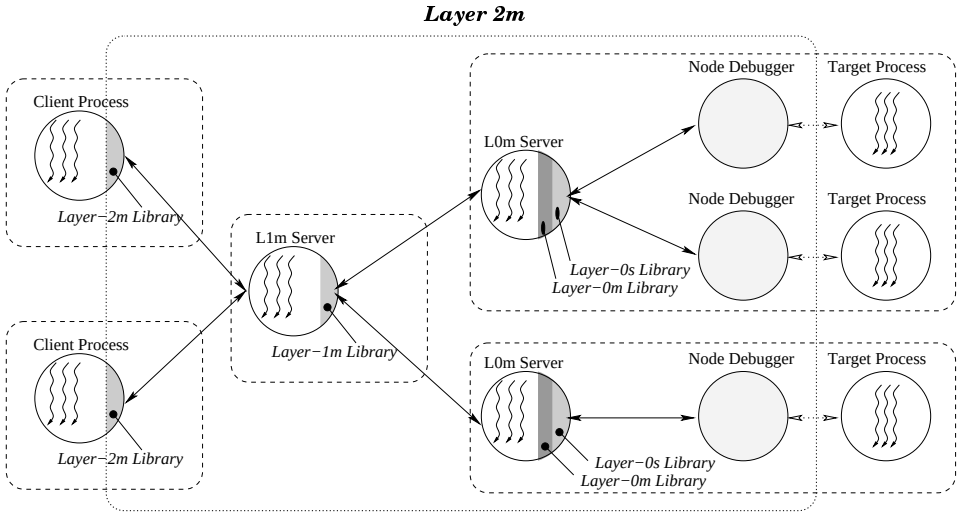

FIG. 10. *The Layer 2_m architecture*

identifier. Upon completion of the request, a previously specified handler (*call-back*) will be triggered by Fiddle and invoked to process the reply. The *request identifier* is also passed to the handler on its activation, allowing a single handler to be used to process different kinds of events.

5. Tool interaction in Fiddle. In this section we first discuss tool interoperability aspects concerning Fiddle, and then we relate them to several tool integration scenarios.

5.1. Interaction events in Fiddle. As Fiddle accepts multiple clients simultaneously, each providing a possibly different interface/view to the target application, there may exist the following classes of interaction events:

- **Method invocation events** are associated with Fiddle method invocation. This corresponds to a client tool calling a Fiddle method which may act or not upon a target-process;
- **Method reply events** are associated with the replies and/or success status of the method execution, which is reported to the client tools;
- **target application events** are associated with changes in the data or execution state of a target-process.

Each tool can subscribe to certain classes of events in which the tool is interested, by specifying an handler to process those events. An handler in a client tool $\mathcal{T}_a$ may be activated in one of the following situations:

- When any tool $\mathcal{T}$ invokes a method;
- When the processing of a method invoked by a tool $\mathcal{T}$ is terminated;
- When one of the target-processes changes its internal data state (e.g., data value);
- When one of the target-processes changes its execution state (e.g., stops).

In this way, a Fiddle's method invocation may originate different types of events. For example, when a client tool $\mathcal{T}_i$ calls the "continue()" method, the following events are generated by Fiddle and sent to all client tools $\mathcal{T}_j$ which have subscribed to that class of events (maybe including $\mathcal{T}_i$ itself):

1. A *method invocation event* to inform about the service requested, namely who in-

voked which method, and which is the target-process;
2. A *method reply event* with the reply from Fiddle, in this case just reporting if the service requested was accepted or not by the node debugger;
3. A *target application event* reporting a change from <stopped> to <running> execution state;
4. Another *target application event* event will eventually be generated, reporting a change from <running> to <stopped> execution state.

5.2. Supporting Fiddle services over DAMS. In order to develop a distributed application, cooperation among multiple distinct tools is needed. Such tools must be able to access the state of the target-processes as well as act according to their evolution. Typical tools in a software engineering environment can act as observers or controllers of the application, so their coexistence requires the coordination of their interactions.

DAMS provides basic mechanisms allowing multiple tools to connect to common Service Module instances, thus enabling their access to the same target-processes. The state changes of those processes can also be observed by the tools through the DAMS event mechanism.

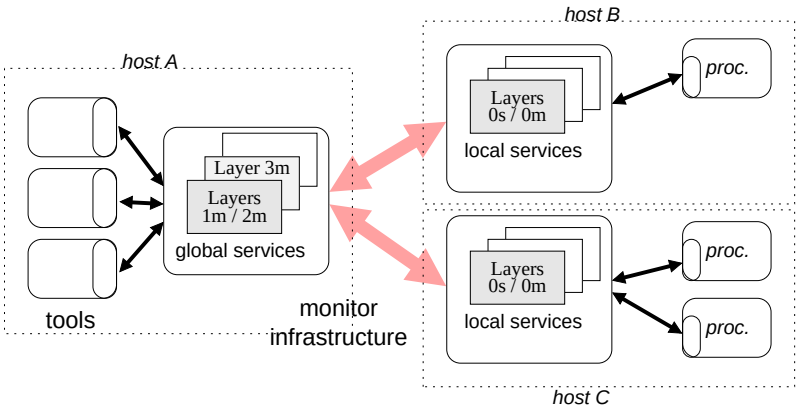

FIG. 11. *Fiddle as a DAMS service*

One of the goals of our work is to evaluate how DAMS can be used to support the inclusion of Fiddle functionalities as DAMS services. This promotes the coexistence of Fiddle with other services which may support complementary development tools, other than debugging. It also exploits the DAMS mechanisms to ease the implementation of some of Fiddle concepts and help the managing of the interactions established among tools. In this section we show how Fiddle architecture can be mapped onto the DAMS framework.

As shown in Fig. 11 both Layer 0_s and Layer 0_m are mapped onto a DAMS *node debugging service*. As DAMS by default accepts many clients for each service supported, Layer 1_m and Layer 2_m are collapsed into just one Service Module which becomes a *global distributed debugging service*. Layer 3_m is also mapped to a service whose implementation is strongly simplified, as it can make extensive use of DAMS event dissemination capabilities, needing only to concentrate in their application to the distributed debugging service.

5.3. Case Studies. Fiddle–DAMS is being experimentally evaluated through three case studies, concerning distinct scenarios of the debugging activity in a parallel software engineering environment.

1. Integration of Graphical Application Design and Debugging;

2. Integration of Testing and Debugging;
3. Integration of Execution Visualization and Debugging.

Although each case study was developed separately from the others, they all aim at improving the support for developing correct parallel and distributed applications, as illustrated by the global picture in Fig. 12.

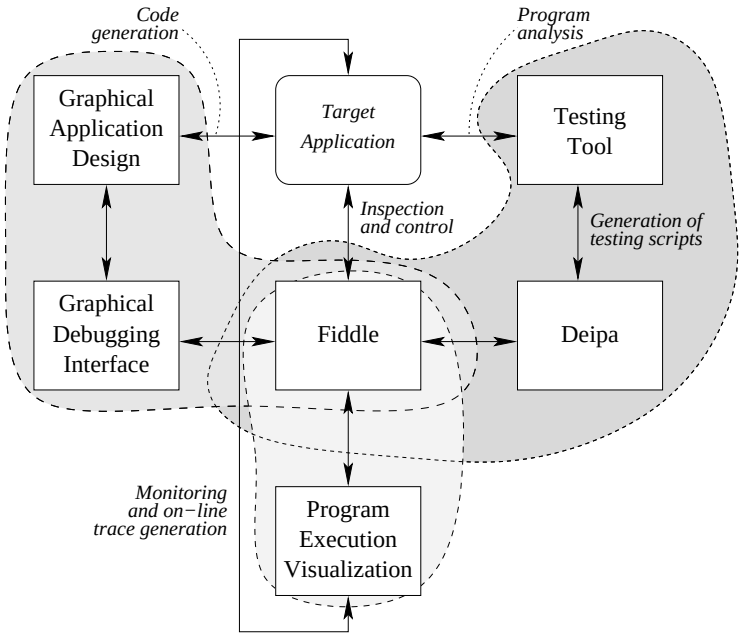

FIG. 12. *Case studies*

Previously, we have developed prototypes for the first two scenarios, based on the early prototypes of the DDBG/PDBG debuggers [8, 13]. This past experimentation has shown the benefits of tool composition [6], but it also shown the need of developing more robust and flexible prototypes.

In the following, we briefly summarize how Fiddle and DAMS may improve tool interoperability support for each case study, when compared to previous work.

5.3.1. Visual Programming and Graphical Debugging. Fiddle-DAMS functionalities can help developing a better tool integration than the previously developed prototype based on DDBG/PDBG debugger:

- Thread support in the client tools eases the control of the asynchronous interactions with the graphical user interface;
- Support for tool management allows a clear control of the concurrent actions of the graphical user editor and debugging interfaces, and a text-oriented console, when it is necessary to inspect the application behavior at the level of the generated source code;
- Tool synchronization events ease the management of consistent views and their updates at the graphical user interface level, according to the debugger actions and the application execution state transitions.

5.3.2. Integrating Testing and Debugging. There is ongoing work in improving this prototype by using Fiddle-DAMS instead of the DDBG. The new prototype facilitates the inte-

gration of Deipa as a new DAMS service which interacts with STEPS and Fiddle. It also eases the coordination between the global views and the actions performed by Deipa and Fiddle.

5.3.3. Integrating Execution Visualization and Debugging. Program visualizers are commonly used to help understanding the behavior of a distributed application. Most of these visualizers display (at least) the status and communication events of the application processes. Pajé [14], developed at ID-IMAG, France, is a thread visualizer for the Athapascan [4] programming language. Athapascan is based on a distributed memory model consisting of multiple program nodes which communicate using message-passing, each node consisting of multiple threads.

Research is under way to coordinate the on-line observation of a distributed application, as provided by Pajé, and the debugging actions of Fiddle. Both tools were previously independently developed so they must now be adapted as DAMS services and their user interfaces must become DAMS clients. This will enable to exploit the DAMS functionalities and services (see Sec. 3).

Tool coordination will be based on the *Target-process / Tool* and *Tool / Tool* interactions, so that both tools may be aware of the target application state and of each other actions. Also, coordination of their individual actions must be ensured as they refer to the same shared distributed application. For example, whenever a process under debugging reaches a breakpoint and stops, then such a state transition must be accordingly updated by Pajé and reflected on its on-line visualization. On the other hand, if Pajé shows that a given process is blocked waiting for a message, we may be interested in selecting the process and having Fiddle automatically stopping the process, and selecting the source line containing the message reception code.

The main dimensions of this ongoing project are as follows:

- The Pajé tool is being adapted to support on-line monitoring and visualization of an Athapascan program;
- Due to its multi-thread and multi-process support, Fiddle can be used as a debugger for the Athapascan multi-threaded distributed model;
- Tool synchronization requires consistent updates of Pajé displays with respect to the Fiddle actions and the execution events.

5.3.4. Conclusion. Overall, these three projects are illustrating the feasibility of developing complex functionalities for program development, through the composition of separate tools. They are also showing the flexibility of a monitoring infrastructure like the DAMS, for supporting distinct types of services and allowing the coordination of multiple concurrent cooperating tools.

6. Implementation Status. The Fiddle development followed two distinct approaches. First, a Fiddle prototype was developed as a stand-alone tool, i.e., without relying on any independent monitoring and control layer. This allowed the independent testing of all Fiddle layers from 0 to 2, which are now fully operational in the current prototype. Event notification, provided by Layer 3_m, is currently under development.

The second approach, is based on implementing Fiddle as a set of services over a new prototype of DAMS, currently under development. This new DAMS prototype provides a clean definition and support for Service Modules on any host from the DAMS environment, and also a fully working event dissemination mechanism.

7. Conclusions and Ongoing Work. We discussed solutions for the development of flexible tools for on-line observation and control of parallel and distributed applications. This work is a continuation of our previous work on the development of the DAMS monitoring architecture and the DDBG/PDBG debuggers. As discussed in the paper, our current focus is on the improvement of DAMS for supporting tool interoperability and coordination services.

This work has been guided by experimentation with case studies involving real software development tools. We have shown how DAMS can include a new complex service, such as the multiple Fiddle layers. The Fiddle tool addresses several requirements for the debugging of multi-thread/multi-process distributed applications. Its suitability is evaluated through the development of new prototypes for the described tool case studies.

The tool integrating scenarios are allowing us to assess tool interoperability when using the Fiddle services and this also helps us improving the DAMS event notification mechanisms and the definition of new services for tool coordination. Further work is required to address still open issues concerning low-level interferences between concurrent tools which access the same application state.

Current and future work focus on the improvement of the DAMS mechanisms, the complete implementation of the Fiddle version as a DAMS service, and the full implementation of the three tool integration prototypes, based on Fiddle and DAMS. In each tool integration case study there are interesting open issues. Namely, testing and debugging can evolve to provide some automated support in verifying if the behavior of an application is consistent with a behavior specification file, and then in activating a set of Fiddle clients to help the user analyze the erroneous situation.

Acknowledgments. Thanks to the anonymous reviewers for their comments and suggestions to improve the paper. The work reported in this paper was partially supported by the PRAXIS XXI Programme (SETNA Project), by the CITI (Centre for Informatics and Information Technology of FCT/UNL), and by the 1999/2000 cooperation protocol ICCTI/French Embassy in Portugal. Previous work was partially funded by the SEPP/ HPCTI European Copernicus/KIT projects.

REFERENCES

[1] R. A. AYDT, *SDDF: The Pablo Self-Describing Data Format*, tech. rep., Department of Computer Science, University of Illinois, Apr. 1994.

[2] S. BROWNE, J. DONGARRA, N. GARNER, G. HO, AND P. MUCCI, *A portable programming interface for performance evaluation on modern processors*, The International Journal of High Performance Computing Applications, (2000), pp. 189–204.

[3] B. BUCK AND J. K. HOLLINGSWORTH, *An API for runtime code patching*, The International Journal of High Performance Computing Applications, 14 (2000), pp. 317–329.

[4] A. CARISSIMI AND M. PASIN, *Athapascan: An experience on mixing MPI commmunications and threads*, in 5th Euro PVM/MPI, vol. 1497 of LNCS, Springer, 1998, p. 137.

[5] J. C. CUNHA AND V. DUARTE, *Monitoring PVM programs using the DAMS approach*, in 5th Euro PVM/MPI, vol. 1497 of LNCS, Springer, 1998, pp. 273–280.

[6] J. C. CUNHA, P. KACSUK, AND S. WINTER, eds., *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environment*, Nova Science Publishers, Inc., 2000.

[7] J. C. CUNHA, J. LOURENÇO, AND T. ANTÃO, *An experiment in tool integration: the DDBG parallel and distributed debugger*, Journal of Systems Architecture, 45 (1999), pp. 897–907. Elsevier Science Press.

[8] J. C. CUNHA, J. LOURENÇO, J. VIEIRA, B. MOSCÃO, AND D. PEREIRA, *A framework to support parallel and distributed debugging*, in Proc. of the International Conference on High-Performance Computing and Networking (HPCN'98), vol. 1401 of LNCS, Amsterdam, The Netherlands, Apr. 1998, pp. 708–717.

[9] G. A. GEIST, M. T. HEATH, B. W. PEYTON, AND P. H. WORLEY, *A User's guide to PICL, a Portable Instrumented Communication Library*, Oak Ridge National Lab., Tennessee, 1990.

[10] O. M. GROUP, *The Common Object Request Broker: Architecture and Specification (v2.4)*, OMG, Inc., 2000.

[11] M. T. HEATH AND J. A. ETHERIDGE, *ParaGraph: A tool for visualizing performance of parallel programs*. Univ. of Illinois and Oak Ridge National Lab., 1992.

[12] R. HOOD, *The p2d2 project: Building a portable distributed debugger*, in Proceedings of the 2nd Symposium on Parallel and Distributed Tools (SPDT'96), Philadelphia PA, USA, 1996, ACM.

[13] P. KACSUK, J. C. CUNHA, G. DÓZSA, J. LOURENÇO, T. FADGYAS, AND T. ANTÃO, *A graphical development and debugging environment for parallel programs*, Parallel Computing, 22 (1997), pp. 1747–1770.

[14] J. C. KERGOMMEAUX AND B. O. STEIN, *Pajé: An extensible environment for visualizing multi-threaded programs executions*, in Proc. Euro-Par 2000, vol. 1900 of LNCS, Springer, 2000, pp. 133–140.

[15] H. KRAWCZYK AND B. WISZNIEWSKI, *Interactive testing tool for parallel programs*, in Software Engineering for Parallel and Distributed Systems, I. Jelly, I. Gorton, and P. Crolll, eds., London, UK, 1996, Chapman & Hal, pp. 98–109.

[16] J. LOURENÇO AND J. C. CUNHA, *Flexible Interface for Distributed Debugging (Library and Engine): Reference Manual (V 0.3.1)*, Departamento de Informática da Universidade Nova de Lisboa, Portugal, Dec. 2000. Under development.

[17] J. LOURENÇO, J. C. CUNHA, H. KRAWCZYK, P. KUZORA, M. NEYMAN, AND B. WISZNIEWSK, *An integrated testing and debugging environment for parallel and distributed programs*, in Proceedings of the 23rd Euromicro Conference (EUROMICRO'97), Budapeste, Hungary, Sept. 1997, IEEE Computer Society Press, pp. 291–298.

[18] T. LUDWIG, R. WISMÜLLER, AND A. BODE, *Interoperable tools based on OMIS*, in Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98), ACM Press, 1998, pp. 155–155.

[19] T. LUDWING, R. WISMÜLLER, V. SUNDERAM, AND A. BODE, *OMIS – On-line monitoring interface specification*, tech. report, LRR-Technish Universiät München and MCS-Emory University, 1997.

[20] B. P. MILLER, J. K. HOLLINGSWORTH, AND M. D. CALLAGHAN, *The Paradyn parallel performance measurement tools*, IEEE Computer, (1995), pp. 37–46. Special issue on performance evaluation tools for parallel and distributed computer systems.

[21] MPI FORUM, *MPI-2: Extensions to the Message-Passing Interface*, Univ. of Tennessee, 1997.

[22] D. M. PASE, *Dynamic Probe Class Library (DPCL): Tutorial and reference guide*, tech. report, IBM, 1998.

[23] R. PRODAN AND J. M. KEWLEY, *A framework for an interoperable tool environment*, in Proc. Euro-Par 2000, vol. 1900 of LNCS, Springer, 2000, pp. 65–69.

[24] *PTR Working Group Home Page*. http://www.ptools.org/projects/ptr/.

[25] D. A. REED, R. A. AYDT, R. J. NOE, P. C. ROTH, K. A. SHIELDS, B. SCHWARTZ, AND L. F. TAVERA, *Scalable performance analysis: The Pablo performance analysis environment*, in Proc. of the Scalable Parallel Libraries Conference, IEEE Computer Society, 1993, pp. 104–113.

[26] D. TOOLWORKS, *TotalView*, Dolphin Interconnect Solutions, Inc., Framingham, Massachusetts, USA.

[27] J. TRINITIS, V. SUNDERAM, T. LUDWIG, AND R. WISMÜLLER, *Interoperability support in distributed online monitoring systems*, in Proc. of the International Conference on High-Performance Computing and Networking (HPCN'2000), vol. 1823, Amsterdam, The Netherlands, 2000.

[28] P. H. WORLEY, *A new PICL trace file format*, Tech. Report ORNL/TM-12125, Oak Ridge National Laboratory, Tennessee, 1992.