Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

A Debugging Engine for Parallel and Distributed Programs

João Manuel dos Santos Lourenço

Dissertação apresentada para a obtenção do Grau de Doutor em Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa
(2003)

This dissertation was prepared under the supervision of
Professor José Cardoso e Cunha,
of the Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa.

To my wife, Teresa
my son, Miguel
and my daughter, Rita

[This page was intentionally left blank]

Acknowledgements

I would like to express my gratitude to all those that, directly or indirectly, have contributed to make this thesis possible.

First and foremost, to my supervisor, José Cardoso e Cunha, to whom I'm in debt for the necessary guidance on my research work and, at the same time, for providing me with the desired freedom to pursue my own path. He was also the local task leader for the research projects which motivated this research work, and I owe him for including me in his working team. The chance I was given to participate in the project meetings and to know and discuss my work with other researchers were invaluable contributions to the maturity of my research.

To Vítor Duarte, who was my closest research companion, and has always been a kind friend, ready to attend me in is office room, no matter the reason was research, teaching, systems administration or a light chat.

To Pedro Medeiros and Paulo Lopes, who have volunteered to fulfill some of my teaching duties, providing me with some extra time to write this dissertation. Pedro also shares the office room with me, and he contributes to our informal, friendly and respectful working environment.

to Cecília Gomes, who kindly made a review of a draft of this dissertation; and to Jorge Custódio for the friendly companionship.

To all my colleagues at Departamento de Informática of FCT/UNL, for their contribution to make every working day a pleasant journey.

To those that worked with me in the development and validation of Fiddle, namely Ricardo Anastácio, Pedro Augusto and Vítor Moreira, when students at Universidade Nova de Lisboa; and Denise Stringini and Mairo Pedrini, from Universidade Federal do Rio Grande do Sul, Brazil. Mairo deserves a special reference, for his courage in "digging" into the source code of Fiddle and his wiseness in correcting some of the remaining known bugs.

Finally, my very special thanks to my family. To my wife, Teresa, for her love, support and permanent understanding; to my son, Miguel, for his unconditional love, joy and tenderness; and to my newborn daughter, Rita, for bringing another lighting star to my life. I love you all.

I also would like to acknowledge the following institutions for their financial support: Departamento de Informática and Faculdade de Ciências e Tecnologia of the Universidade Nova de Lisboa; Centro de Informática e Tecnologias da Informação of the FCT/UNL; Reitoria da Universidade Nova de Lisboa; Fundação Calouste Gulbenkian; Fundação Luso-Americana para o Desenvolvimento; Fundação para a Ciência e Tecnologia through CIÊNCIA and PRAXIS-XXI Programmes, projects PROLOPPE (Contract 3/3.1/TIT/24/94) and SETNA-ParComp (Contract 2/2.1/TIT/1557/95); Instituto de Cooperação Científica e Tecnológica Internacional; French Embassy in Portugal; European Union Commission through the Copernicus Programme, projects SEPP (Contract CIPA-C193-0251) and HPCTI (Contract CP-93-5383); and to Digital Equipment Coorporation through the European External Research Programme, project PADIPRO (Contract nº P-005).

Summary

In the last decade a considerable amount of research work has focused on distributed debugging, one of the crucial fields in the parallel software development cycle. The productivity of the software development process strongly depends on the adequate definition of what debugging tools should be provided, and what debugging methodologies and functionalities should these tools support.

The work described in this dissertation was initiated in 1995, in the context of two research projects, the SEPP (Software Engineering for Parallel Processing) and HPCTI (High-Performance Computing Tools for Industry), both sponsored by the European Union in the Copernicus programme, which aimed at the design and implementation of an integrated parallel software development environment. In the context of these projects, two independent toolsets have been developed, the GRADE and EDPEPPS parallel software development environments.

Our contribution to these projects was in the debugging support. We have designed a debugging engine and developed a prototype, which was integrated the both toolsets (it was the only tool developed in the context of the SEPP and HPCTI projects which achieved such a result). Even after the closing of those research projects, further research work on distributed debugger has been carried on, which conducted to the re-design and re-implementation of the debugging engine.

This dissertation describes the debugging engine according to its most up-to-date design and implementation stages. It also reposts some of the experimental work made with both the initial and the current implementations, and how it contributed to validate the design and implementations of the debugging engine.

[This page was intentionally left blank]

Sumário

Na última década uma quantidade considerável de trabalhos de investigação focaram a sua atenção na depuração distribuída, um dos tópicos cruciais no ciclo de desenvolvimento de programas paralelos. A produtividade do processo de desenvolvimento de software depende fortemente da definição adequada das ferramentas de depuração que deverão ser disponibilizadas, e de quais as funcionalidades e metodologias de depuração que deverão ser suportadas por essas ferramentas.

O trabalho descrito nesta dissertação for iniciado em 1995, no contexto de dois projectos de investigação, SEPP (Software Engineering for Parallel Processing) e HPCTI (High-Performance Computing Tools for Industry), ambos patrocinados pela União Europeia no contexto do programa Copernicus, e que visavam o desenvolvimento de um ambiente integrado de desenvolvimento de aplicações paralelas. No contexto destes projectos, foram desenvolvidos dois ambientes disjuntos de desenvolvimento de aplicações paralelas e distribuídas, o GRADE e o EDPEPPS.

A nossa contribuição para estes projectos concentrou-se no suporte à depuração. Desenhámos e implementámos um protótipo de um depurador paralelo, que foi integrado em ambos os ambientes de desenvolvimento de aplicações paralelas (foi a única ferramenta desenvolvida no contexto daqueles projectos a fazê-lo). Mesmo depois do término daqueles projectos, a investigação em depuração distribuída continuou, conduzindo ao redesenho e re-implementação do depurador distribuído.

Esta dissertação descreve o depurador distribuído na seu estágio mais actual. Também reporta algum do trabalho experimental levado a cabo com ambas as implementações, e como ele contribuiu para a validação do desenho e implementação do depurador distribuído.

[This page was intentionally left blank]

Sommaire

Dans la dernière décade, une quantité considérable de travaux de recherche se sont focalisés sur le débogage distribué, un des principaux aspects du cycle de développement de programmes parallèles. La produtivité du procès de développement de logiciel dépend beaucoup de la définition correcte des outils de débogage qui devront être disponibles. Il faut aussi définir les capacités et les méthodologies qui devront être soutenues par ces outils.

Le travail décrit dans cette thèse a été commencé en 1995, dans le contexte des deux projets de recherche, SEPP (Software Engineering for Parallel Processing) et HPCTI (High-Performance Computing Tools for Industry), avec l'appui de l'Union Européenne, dans le contexte du programme Copernicus, et qui cherchaient à développer un environnement intégré de développement d'applications parallèles. Dans le contexte de ces projets-là, on a développé deux différents environnements de développement d'applications paralèlles distribuées, le GRADE et le EDPEPPS.

Notre apport pour ces projets s'est centré sur le support du débogage. On a dessiné et on a mis en oeuvre un prototype d'un débogueur parallèle, qui a été intégré dans les deux environnements de développement d'applications parallèles (c'était le seul outil développé dans le contexte de ceux projets-là à faire ça). Même après la fin des projets, la recherche sur le débogage distribué ne s'est pas arrêtée, conduisant à une nouvelle mise en oeuvre de l'épurateur distribué.

Cette thèse décrit le débogueur distribué dans son état le plus actuel. Elle s'en occupe aussi d'une partie du travail expérimental réalisé dans les deux mises en oeuvre du déboguer et de la façon dont il a contribué pour la validation du dessin et de la mise en oeuvre du débogueur distribué.

[This page was intentionally left blank]

Contents

[This page was intentionally left blank]

List of Figures

List of Tables

[This page was intentionally left blank]

1

Introduction

Contents

This Chapter introduces the motivation to the debugging activity and its role as one important task in the software development process, enumerates the main contributions of this thesis and presents an outline of the dissertation, with a brief summary of each of the remaining Chapters.

1.1 Introduction

There is a long and hard way to go, since someone realises that there is a problem which could be solved with the aid of a computer, until the moment the computer is contributing towards such goal. One of the most important steps in such a path is the design and development of a computer program which will *correctly* meet the requirements of the problem to be solved.

Developing a computer program is, indubitably, a complex task. There is the need to analyse the application requirements and to produce a valid model which describes a solution. Such model will then be carefully specified in a programming language and refined successively until its description fits the machine language of a specific computer architecture. Once such stage is reached, the specification (computer program) will be executed by the computer.

If there were no mistakes in any of the software development stages, the computer program will, supposedly, be correct and will implement a solution for the initial problem. However, experience shows that, frequently, there were mistakes or misconceptions at some of the stages. Such mistakes or misconceptions will not only compromise some of the activities at that specific stage but also in the following ones. The final program will then be an inadequate, incomplete, and/or erroneous solution to the initial problem.

Defining program correctness isn't an easy task. Like beauty, program correctness strongly depends on the eye of the beholder. A program may be correct from the point of view of the implementer if it satisfies all the previously defined requirements, but may be incorrect from the client perspective if the requirements were incomplete or incorrectly defined. To minimize the risk of building useless computer programs, a software development methodology should be followed [Roy70]. Even when such a software development methodology is carefully followed, there may be discrepancies between the theoretically correct value or behavior and the computed or observed behavior. In such cases the program is said to contain an error (or a set of errors).

The final aim is, ideally, to generate bug free programs at the first try. However, often this is not the case and, thus, a careful testing of the individual program components (unit testing) and of the full program (program testing) is mandatory. Whenever an unexpected behavior is observed, additional verifications are required to determine if, although unexpected, the observed behavior is or isn't acceptable. In the latter, a debugging methodology should then be followed to diagnose and correct the undesired behavior.

A computer program may present different kinds of malfunctions, or even not to operate at all, due to program errors. According to their nature, such errors may be classified in the following categories:

a) *Specification errors* result from an inadequate characterization of the problem or from

an ambiguous or incomplete definition of the requirements of the proposed solution;

b) *Algorithmic errors* result from the application of an inadequate algorithm to implement the devised specification;

c) *Logical errors* result from the inadequate comprehension of the algorithm and lead to a faulty implementation, which may cause the program to fail intermittently due to a conjunction of factors, or to fail permanently, by not implementing the algorithm at all;

d) *Coding errors* result from an inadequate understanding of the programming language being used or from mistakes when writing the source code. Unpredictable behavior due to unpredicted input, wrong array indexing and use of uninitialized variables are examples of such errors;

e) *Architectural errors* are those due to the underlying system layers, such as the operating system or message passing libraries, typically out of the control of the application software developers.

Many of the above errors could be avoided or, at least, minimized, if a formal specification language could be used to specify the desired program behavior and, later, have its behavior observed and automatically matched against that specification. Research has been conducted towards such goals [Jac02, Abr96, Spi95, Cho78] but is usually limited to small examples, and some authors argue that writing such formal descriptions for larger programs is impracticable [Fet88].

Some coding errors, such as lexical and syntactical errors, may be efficiently detected by a compiler-based static analysis of the source code and reported to the software developer as a compiler fatal-error, forcing its immediate elimination. Even other non-trivial errors, such as the potential usage of uninitialized variables may be detected by static analysis and reported to the software developer as a compiler warning (non-fatal error).

Just a subset of the coding errors are commonly detected at compile-time. All the others rely on the software developer perception to detect any misbehaviors, to analyse the program source code and to devise a possible correction.

The term debugging is usually associated with the process of locating, diagnosing and correcting errors of the logical and coding classes which are only detectable during program execution.

Non-intrusive debugging is based on the static analysis of the source code and on a symbolic execution. Such symbolic execution may be performed by the software developer, which reads the source code and mentally simulates its execution, or by some tool, which also analyses the program source code without running it [Fau03]. Based on the input and output data, the symbolic execution will allow to hypothesize

about the error and to devise a correction. This approach, however, strongly depends on the cause of the error and on the software developer skills for such analysis and, frequently, the nature of the error can't be found this way.

Intrusive debugging modifies the program behavior by changing the program itself or, at least, by controlling its execution, and can only be used in reproducible errors, when subsequent runs of the program behave identically wrong. Changing the program itself, by adding a few memory dump commands (e.g., printing variable values) is, by far, the most common approach for program debugging. Even if acceptable for very simple programs, such an approach is of very limited effectiveness for larger and more complex programs. The most effective approach to intrusive debugging is based in the usage of *debuggers*.

> ***Debugger*** *(n.)*
>
> *A program for locating operational errors in another program. The debugger usually enables the developer to step through the malfunctioning portion of the program to examine data and check operational conditions.*
>
> *In http://docs.sun.com/db/doc/805-4368/6j450e60d*

Debuggers depend on the computer hardware, operating system and also on the programming language. This last dependency is more relevant if the debugger supports *source-level debugging*, i.e., allow the software developer to use the source code as the basic reference for debugging instead of the target machine code.

For a given set of input data, sequential programs are typically deterministic and their errors are reproducible. Such characteristics make them the perfect targets for controlled execution by a debugger, single stepping over the sequence of machine instructions (or source-lines, in case of a source-level debugger) and examining process core memory (including variable values) whenever needed. Although there is always space for innovation, the techniques and technologies behind the debugging of sequential programs are quite stable and of widespread use nowadays.

Concurrent programs are intrinsically non-deterministic and repeated executions of the same program with the same input data may originate different behaviors. As such, their controlled execution by a debugger may pose an unacceptable degree of intrusion and hide/mask an error. The simple act of observing the execution of a concurrent program is another source of intrusion, which may have serious implications on a non-deterministic program behavior.

Distributed programs bring additional difficulties to the debugging activity because they run on a distributed system architecture, which lacks a global clock, making it impossible to have instantaneous snapshot of all the processes and communication channels of the distributed program. As the observation of the distributed program state is non-atomic, in the time elapsed between observing one program component and another, the state of the first one may have changed. The resulting global state

obtained in such a way is outdated and may even be inconsistent, precluding reasoning about the program behavior. To avoid such situations, special concerns must be considered to ensure that all the reasoning about the program state is based on valid (consistent) global states.

Global states are built based on observations of the execution of program components. What should be observed and how, depends on the abstraction level and the programming model being used. Different abstraction levels may be considered, and the debugging activity may focus in individual processes or their interactions. For example, one may be interested in knowing the state of the communication channels at a certain point or in a temporal perspective reporting the processes and messages associated with those channels in specific time intervals.

Typically, although not always, one would like to observe the program state at the same abstraction level as the one used for the program development. For example, if the used programming language allowed the software developer to ignore the processor registers level, normally it should also be ignored during debugging. However this is not always true, and the debugger should provide the means to deal with the multiple abstraction-levels associated with the program and its execution environment, giving the software developer the freedom to chose, at any point, which ones should be considered and which ones should be left out.

A distributed program is a collection of sequential processes which interact among themselves. As such, debugging a distributed programs encompasses all the difficulties of debugging sequential programs and, additionally, many new ones, such as the need to deal with multiple flows of control (multiple threads and/or processes), process interactions, non-determinism, additional failure sources, multiple programming models and abstractions, and the lack of production quality software development environments with specific support for concurrency, parallelism and/or distribution.

Although there are a few commercial distributed debuggers [Etn00, Mos88], distributed debugging is still a fruitful research topic with much ongoing work in different points of the globe. The international conferences specifically dedicated to the topic of (distributed) debugging, such as the past ACM/ONR Workshops on Parallel and Distributed Debugging (1989 [acm89], 1991 [acm91] and 1993 [acm93]) and the International Workshops on Automated Debugging (1993 [Fri93], 1995 [Duc95], 1997 [Duc97], 2000 [Duc01] and 2003 [RB03]), are rich information resources about the ongoing research on this field and confirm its relevance as a specialized research topic.

1.2 Motivation

The usage of symbolic debuggers to help in the location and identification of program errors is a major step over more *ad-hoc* methodologies, such as inserting variable printing statements into the source code, recompiling and rerunning the program, and then

browsing the (potentially) large amount of output produced by those print statements. Symbolic debuggers for sequential programs are, essentially, state based. This means they support a debugging methodology based on stopping the program execution at specific points and examining its computation state (variables, stack, registers, etc).

Program development for distributed systems has motivated the redesign of some programming languages and models, and the development of new ones. Understanding distributed computations in both state and temporal perspectives (see Figure 1.1) involves a set of new difficulties, such as non-determinism, lack of global components (memory, clock, etc.), multiple execution flows, and variable communication delays. As debuggers are expected to help the software developer understanding the program behavior, distributed debuggers should help the developer to cope with such new difficulties.

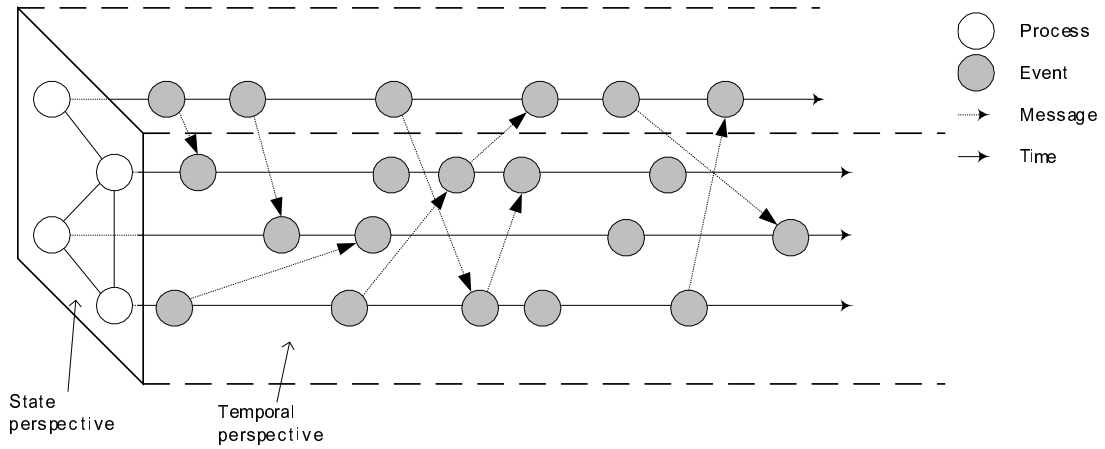

Figure 1.1: State vs. temporal perspective of a distributed program

The first and more natural approach to distributed debugging is to extend a sequential debugger to interact with more than one process, providing the software developer with a single debugging interface to access all the processes of the distributed program. However, the considerable number of adaptations needed to allow a set of sequential debuggers to operate upon distributed programs, and the even larger number of new features that should be addressed and supported, would have strong implications upon the size and complexity of the debugger program itself, with the consequent difficulties in its maintenance.

It is common for a sequential debugger to be unable to interact with the software developer while the controlled process is running, the interaction being resumed as soon as the process stops. Distributed debuggers, however, can't impose such restriction on the user interface, as while some of the processes are running the developer may have a significant activity to perform upon the remaining ones. Such requirement suggests that the user interface should operate asynchronously regarding the distributed debugger. One of the easiest ways to support such asynchronous operation is to make the debugging interface multi-threaded, having a set of threads to control the target processes and another set to control the user interface.

Another basic requirement for distributed debuggers is to provide transparent access to remote processes, using a global naming scheme independent of process localization and freeing the software developer from the burden of knowing which process is running where.

The complexity of a distributed debugging tool will increase as the number of required debugging functionalities grows or changes over time. An approach to reduce considerably the overall complexity of the distributed debugger, is to precisely defining the core functionalities as a minimal set of services, and support an extension mechanism. Additional services, developed as external modules, can then be incorporated into the distributed debugger as extensions.

One basic requirement to support extensions, is to decouple the debugging engine from the debugging user interface. In such a way, the functionalities provided by the debugging engine may grow incrementally and independently from the user interface; and multiple independent user interfaces can be allowed to operate concurrently upon the same target processes, exploring the basic functionalities provided by the debugging engine and some of the functionalities provided by one or more extensions.

For a better understanding of the distributed computation, it is desirable for the software developer to have multiple perspectives of the target program, probably provided by different tools. For example, having a graphical editor of a visual parallel programming language providing an high-level view of the source code, a distributed debugger providing a state-based view, and a computation visualizer providing a time-based view. Different tools have different coordination requirements and operate accordingly to different coordination models, from loosely-coupled cooperations with simple interactions, to tightly-coupled integrations with complex data and control interactions.

The debugging engine proposed in this thesis follows the line of thinking presented above, aiming at the provision of basic debugging services for distributed programs and the support for interoperability and integration with other software development tools. It provides a complete set of process-level services, such as breakpointing and single-stepping, which was extended to include additional distributed debugging services, such as monitoring and replaying, the possibility to cooperate with other tools, such as computation visualizers, and the ability to be integrated in *parallel software development environments*.

1.3 Contributions of this Thesis

We may summarize the research work discussed in this thesis as:

> *Studied the main requirements for the debugging of distributed programs, defined a debugging engine which tries to fulfill those requirements, designed a software architecture which supports the defined debugging engine, implemented this soft-*

ware architecture in Linux based machines, and evaluated the debugging engine by designing, implementing and evaluating a set of experiments which explore its functionalities.

The above contributions can be further detailed as:

a) *Requirements for the debugging of distributed programs.* We have studied how the software development and execution environments influence the functionalities required from a distributed debugger, and how the testing methodology and user's (software developer's) perspective influences how those functionalities may be explored;

b) *Definition of the debugging engine.* We have focused in the definition of a debugging engine which would satisfy three main requisites:

   b.1) *Minimalism.* To include a set of core basic services which are essential to the debugging of distributed programs and to the support of the other requisites;

   b.2) *Extensibility.* To allow the evolution of the distributed debugger, the support of more complex functionalities, and the adaptation of the debugger to specific needs;

   b.3) *Interoperability* and *Integrability.* To support the exchange of data and control information with other software development tools. We define interoperability as a loosely-coupled cooperation and integrability as a tightly-coupled cooperation between two tools;

c) *Design of a software architecture for the debugging engine.* The defined debugging engine was structured in (five) functional layers, each new layer based upon the previous one and incrementally providing a new set of services;

d) *Implementation of the debugging engine.* We have made two major implementations of the debugging engine: DDBG, which implemented an initial specification of the debugging engine; and Fiddle, which implemented the specification and used the software architecture that is described in this dissertation;

e) *Design and implementation of extensions to the debugging engine.* The debugging engine was defined to incorporate a minimal core set of services, and to provide the means for other services to be incorporated as extensions. In this context, a set of extensions providing complementary services were designed, implemented and incorporated into the debugging engine;

f) *Evaluation of the debugging engine.* A considerable number of experiments using both prototypes, DDBG and Fiddle, have been performed both locally (at UNL) and by other external research groups. Some of these experiments explored the extensibility of the debugging engine to incorporate new services, while some others

involved the full development of client tools which explored the available functionalities. In both cases, they allowed to validate the design and the implementation of the debugging engine.

1.4 Outline of the Dissertation

This dissertation contains seven chapters, whose contents are summarized below:

Chapter 1. This Chapter introduces the motivation to the debugging activity and its role as one important task in the software development process, enumerates the main contributions of this thesis and presents an outline of the dissertation, with a brief summary of each of the remaining Chapters;

Chapter 2. In this Chapter a brief overview is presented of the main dimensions involved in the debugging of parallel and distributed programs;

Chapter 3. This Chapter introduces the main requirements for distributed debugging and how traditional debugging services fulfill some of those requirements, followed by presentation of the software architecture of a debugging engine which fulfills some of those requirements, and how this debugging engine may be extended with complementary functionalities which may cover the remaining requirements;

Chapter 4. This Chapter illustrates how the debugging engine described in the previous Chapter has been instantiated in two prototypes: the DDBG (Distributed DeBuGger) and Fiddle (Flexible Interface for Distributed Debugging: Library and Engine);

Chapter 5. This Chapter presents a set of case studies, where one of the debugging engine implementations (DDBG or Fiddle) have been used, and how they contributed to the operational and functional validations of the debugging engine and its implementations; and

Chapter 6. This Chapter summarizes the achievements of research work described in this thesis, and lists some still open issues, which should and will ground our future research work.

[This page was intentionally left blank]

2

Debugging of Parallel and Distributed Programs

Contents

Program debugging is one of the fundamental activities in the software development process. In the past two decades there were continuous efforts towards improving the debugging of concurrent, parallel and distributed programs. In this Chapter, a brief overview is presented of the main dimensions involved in the debugging of parallel and distributed programs.

2.1 Basic Concepts

A computer program is defined in the Lectric Law Library's Lexicon [Lex] as,

> ***Computer Program*** *— A set of statements or instructions to be used directly or indirectly in a computer in order to bring about a certain result.*

According to the Hyper Dictionary [Dica], computer programs may be split in two groups, system software and applications, defined as

> ***System Software*** *— System software is any software required to support the production or execution of application programs but which is not specific to any particular application. Examples of system software would include the operating system, compilers, editors and sorting programs;*
>
> ***Applications*** *— A complete, self-contained program that performs a specific function directly for the user. Examples of application programs would include an accounts package or a CAD program.*

Relying on the above definition of computer program, a process can be defined as [Dica]

> ***Process*** *— The sequence of states of an executing program. A process consists of the program code (which may be shared with other processes which are executing the same program), private data, and the state of the processor, particularly the values in its registers. It may have other associated resources such as a process identifier, open files, CPU time limits, shared memory, child processes, and signal handlers.*

Associated to the execution of a process is the concept of *current state*, which implies the knowledge of what has already been done, what is currently being done, and what still remains to be done.

The same computer program may be executed again and again, each time in a new process, so that each new process provides a new execution context. It is also possible to have multiple instances of the same or different computer programs executing concurrently in the same computing node, in a *multitasking* system.

> ***Multitasking*** *— A technique used in an operating system for sharing a single processor between several independent jobs. [. . .] A multitasking operating system should provide some degree of protection of one task from another to prevent tasks from interacting in unexpected ways such as accidentally modifying the contents of each other's memory areas.*

Such multitasking systems are, in general, capable of isolating and hiding each process from the others, providing an execution environment which simulates exclusiveness on the access to the computing and computer resources.

In [Dica], a thread is defined as

Thread *— A control (execution) flow in a process.*

When a process contains a single control flow, i.e., a single thread, it is common to associate the single control flow to the process itself and omit the references to the thread. However, some programs may use multiple control flows evolving concurrently "inside" the execution environment provided by the process. Such programs are said to be multi-threaded.

In [Dica] multithreading is defined as

> **Multithreading** — Differs from multitasking in that threads share more of their environment with each other than do processes under multitasking. Threads may be distinguished only by the value of their program counters and stack pointers while sharing a single address space and set of global variables. There is thus very little protection of one thread from another, in contrast to processes in multitasking.

Summarizing, one can say that programs are a passive entity and contain a set of instructions to be executed by the computer. Processes are active entities, resulting from particular instantiations of programs being executed. The programs directed towards the end-user are called applications, while those associated with the management of system (computer) resources are called system programs. Some processes contain a single control flow while some others do contain multiple control flows, and are said to be single- or multi-threaded processes respectively. Systems that allow the time sharing of the CPU between multiple processes (and their control flows) are said to support multitasking.

Due to the isolation factors, usually it makes no difference whether a program is being executed in a single or in a multitasking environment. In what concerns to multithreading, the situation is quite different, and the program must be aware of the multiple control flows and use them explicitly.

2.1.1 The Program Specification and Behavior

A computer program has, necessarily, a goal, which depends on the accomplishment of a set of (intermediate) objectives. Such set of objectives informally define the intended program behavior.

Frequently, such behavior model exists uniquely in the mind of the developer, being constructed, adapted, extended and corrected as the need arises. Even when there is an initial written specification of such intended behavior, it is frequently done in a very high-level description language with no formal grounding, such as natural language. This results in incomplete, ambiguous or even inconsistent behavior descriptions, with negative implications to the program development process and its assessment.

Ideally, the programming language would be able to fully capture and express the intended semantics for the program being developed and, therefore, its intended be-

havior. Unfortunately this is not the usual case and, to be able to express the intended program behavior, the developer has to perform abstraction and simplification efforts, recurring to a limited number of concepts and under the syntactic and semantics restrictions and limitations of the programming language.

Programming language is defined in [Dica] as

> ***Programming Language*** *— A formal language in which computer programs are written. The definition of a particular language consists of both syntax (how the various symbols of the language may be combined) and semantics (the meaning of the language constructs).*
>
> *Languages are classified as low level if they are close to machine code and high level if each language statement corresponds to many machine code instructions.*

Programs are converted to machine code (CPU instructions) by compilers or interpreters, defined in [Dica] as

> ***Compiler*** *— A program that converts another program from some source language (or programming language) to machine language (object code) which is output to a file for later execution. Some compilers output assembly language which is then converted to machine language by a separate assembler.*
>
> *A compiler is distinguished from an assembler by the fact that each input statement does not, in general, correspond to a single machine instruction or fixed sequence of instructions.*
>
> ***Interpreter*** *— A program which executes other programs. [. . .]]It may be possible to execute the same source code either directly by an interpreter or by compiling it and then executing the machine code produced.*

The compilers and interpreters verify that the program strictly complies to the syntactic rules of the programming language and also do some simple semantic verifications, such as detecting that a variable is used before being initialized. However, such semantic verifications are quite far from the intended program behavior in the mind of the programmer.

Due to the limitations of programming languages in the expressiveness of the intended program behavior, and of compilers/interpreters in its verification, one can (and should) also verify the program behavior during execution, the *observed behavior*, against the intended behavior specification. The success in such verification simply allows the developer to have "some confidence" that its specification (program) was correct, but does not constitute a formal proof of program correctness.

2.1.2 Program Correctness

The complex nature of the problem, the inability of the programmer to conceive a valid solution, the adequacy of the programming language to express such solution, and the

software development tools available, are some examples of the many factors that may influence the correction of a computer program.

Program errors result from a mental mistake made by the programmer, and are defined in [Dica] as,

> ***Error*** *— A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.*

Incorrect steps, processes and data definitions are examples of errors. The execution of a program containing errors may originate faults, defined in [Dica] as,

> ***Fault*** *— A manifestation of an error in software.*

Sometimes programs are able to handle some predicted faults. In these cases, although they are still manifestations of program errors, these faults are benign, as they allow the program execution to proceed. Serious or unpredicted faults may be the origin of process (or even system) failures. A failure is defined in [Dica] as,

> ***Failure*** *— The inability of a system or system component to perform a required function within specified limits.*

Errors in software are generically called *bugs*, and the process of locating, diagnosing and correcting software errors called *debugging*.

In [Dicb] there is a definition of bug which includes some interesting historical references. Such definition is duplicated in Figure 2.1 on the next page.

Bugs can also be classified according to the way they behave or manifest themselves. The following definitions are also from [Dicb].

i) ***Bohr bug*** *(n.) [from quantum physics]*
A repeatable bug; one that manifests reliably under a possibly unknown but well-defined set of conditions. Antonym of heisenbug.
In *http://info.astrian.net/jargon/terms/b/Bohr_bug.html*

ii) ***Mandelbug*** *(n.) [from the Mandelbrot set]*
A bug whose underlying causes are so complex and obscure as to make its behavior appear chaotic or even non-deterministic. This term implies that the speaker thinks it is a Bohr bug, rather than a heisenbug.
In *http://info.astrian.net/jargon/terms/m/mandelbug.html*

iii) ***Heisenbug*** *(n.) [from Heisenberg's Uncertainty Principle in quantum physics]*
A bug that disappears or alters its behavior when one attempts to probe or isolate it. (This usage is not even particularly fanciful; the use of a debugger sometimes alters a program's operating environment significantly enough that buggy code, such as that which relies on the values of uninitialized memory, behaves quite differently.) Antonym of Bohr bug; see also mandelbug,

The Jargon Dictionary – http://info.astrian.net/jargon/terms/b/bug.html

*The Jargon Dictionary : Terms : The **B** Terms : bug*

bug

bug *n.* An unwanted and unintended property of a program or piece of hardware, esp. one that causes it to malfunction. Antonym of feature. Examples: "There's a bug in the editor: it writes things out backwards." "The system crashed because of a hardware bug." "Fred is a winner, but he has a few bugs" (i.e., Fred is a good guy, but he has a few personality problems).

Historical note: Admiral Grace Hopper (an early computing pioneer better known for inventing COBOL) liked to tell a story in which a technician solved a glitch in the Harvard Mark II machine by pulling an actual insect out from between the contacts of one of its relays, and she subsequently promulgated bug in its hackish sense as a joke about the incident (though, as she was careful to admit, she was not there when it happened). For many years the logbook associated with the incident and the actual bug in question (a moth) sat in a display case at the Naval Surface Warfare Center (NSWC). The entire story, with a picture of the logbook and the moth taped into it, is recorded in the "Annals of the History of Computing", Vol. 3, No. 3 (July 1981), pp. 285–286.

The text of the log entry (from September 9, 1947), reads "1545 Relay #70 Panel F (moth) in relay. First actual case of bug being found". This wording establishes that the term was already in use at the time in its current specific sense –– and Hopper herself reports that the term 'bug' was regularly applied to problems in radar electronics during WWII.

Indeed, the use of 'bug' to mean an industrial defect was already established in Thomas Edison's time, and a more specific and rather modern use can be found in an electrical handbook from 1896 ("Hawkin's New Catechism of Electricity", Theo. Audel & Co.) which says: "The term 'bug' is used to a limited extent to designate any fault or trouble in the connections or working of electric apparatus." It further notes that the term is "said to have originated in quadruplex telegraphy and have been transferred to all electric apparatus."

The latter observation may explain a common folk etymology of the term; that it came from telephone company usage, in which "bugs in a telephone cable" were blamed for noisy lines. Though this derivation seems to be mistaken, it may well be a distorted memory of a joke first current among *telegraph* operators more than a century ago!

Or perhaps not a joke. Historians of the field inform us that the term "bug" was regularly used in the early days of telegraphy to refer to a variety of semi–automatic telegraphy keyers that would send a string of dots if you held them down. In fact, the Vibroplex keyers (which were among the most common of this type) even had a graphic of a beetle on them (and still do)! While the ability to send repeated dots automatically was very useful for professional morse code operators, these were also significantly trickier to use than the older manual keyers, and it could take some practice to ensure one didn't introduce extraneous dots into the code by holding the key down a fraction too long. In the hands of an inexperienced operator, a Vibroplex "bug" on the line could mean that a lot of garbled Morse would soon be coming your way.

Further, the term "bug" has long been used among radio technicians to describe a device that converts electromagnetic field variations into acoustic signals. It is used to trace radio interference and look for dangerous radio emissions. Radio community usage derives from the roach–like shape of the first versions used by 19th century physicists. The first versions consisted of a coil of wire (roach body), with the two wire ends sticking out and bent back to nearly touch forming a spark gap (roach antennae). The bug is to the radio technician what the stethoscope is to the stereotype medical doctor. This sense is almost certainly ancestral to modern use of "bug" for a covert monitoring device, but may also have contributed to the use of "bug" for the effects of radio interference itself.

Actually, use of 'bug' in the general sense of a disruptive event goes back to Shakespeare! (Henry VI, part III – Act V, Scene II: King Edward: "So, lie thou there. Die thou; and die our fear; For Warwick was a bug that fear'd us all.") In the first edition of Samuel Johnson's dictionary one meaning of 'bug' is "A frightful object; a walking spectre"; this is traced to 'bugbear', a Welsh term for a variety of mythological monster which (to complete the circle) has recently been reintroduced into the popular lexicon through fantasy role–playing games.

In any case, in jargon the word almost never refers to insects. Here is a plausible conversation that never actually happened:

"There is a bug in this ant farm!"

"What do you mean? I don't see any ants in it."

"That's the bug."

A careful discussion of the etymological issues can be found in a paper by Fred R. Shapiro, 1987, "Entomology of the Computer Bug: History and Folklore", American Speech 62(4):376–378.

[There has been a widespread myth that the original bug was moved to the Smithsonian, and an earlier version of this entry so asserted. A correspondent who thought to check discovered that the bug was not there. While investigating this in late 1990, your editor discovered that the NSWC still had the bug, but had unsuccessfully tried to get the Smithsonian to accept it –– and that the present curator of their History of American Technology Museum didn't know this and agreed that it would make a worthwhile exhibit. It was moved to the Smithsonian in mid–1991, but due to space and money constraints was not actually exhibited years afterwards. Thus, the process of investigating the original–computer–bug bug fixed it in an entirely unexpected way, by making the myth true! ––ESR]

Figure 2.1: Definition of "bug"

> *schroedinbug. In C, nine out of ten heisenbugs result from uninitialized auto variables, fandango on core phenomena (esp. lossage related to corruption of the malloc arena) or errors that smash the stack.*
>
> In *http://info.astrian.net/jargon/terms/h/heisenbug.html*

iv) ***Schroedinbug*** *(n.) [MIT: from the Schroedinger's Cat thought-experiment in quantum physics]*
A design or implementation bug in a program that doesn't manifest until someone reading source or using the program in an unusual way notices that it never should have worked, at which point the program promptly stops working for everybody until fixed. Though (like bit rot) this sounds impossible, it happens; some programs have harbored latent schroedinbugs for years.

In *http://info.astrian.net/jargon/terms/s/schroedinbug.html*

The correctness of a program is related to some specification of its intended behavior. Ideally, in order to ensure program correctness, we would like to have a well-defined formal notation to describe application behaviour rigorously and without ambiguity. Such an approach would allow the automatic generation of correct program code. In order to achieve a reasonable level of efficiency, such an approach usually relies upon a series of program transformations, from the high-level specification down to the executable code, with the guarantee of always generating equivalent program representations. In such an approach, bugs can only appear at the level of the application specification, in relation to its intended behavior: *specification bugs*.

However, such an approach cannot be applied in general, so a programmer becomes responsible for the mappings from some expression (formal or informal) of the intended behaviour, that is converted to a program code. Depending on the expressiveness of the programming model and language used, such task can be greatly facilitated. However, such an activity gives the opportunity to introduce another kind of bugs, *programming bugs*. Figure 2.2 illustrates such concepts.

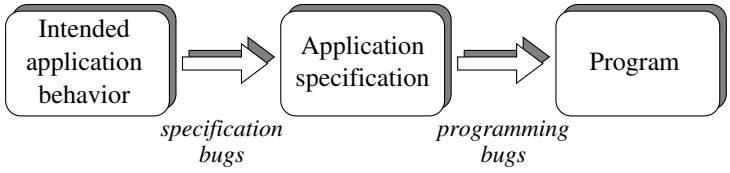

Figure 2.2: Specification and programming bugs

The lack of a formal specification of program behaviour makes the debugging activity extremely complex, as specification and programming bugs both tend to appear mixed at the program code level.

The debugging task becomes more difficult also due to the multiple internal software layers of a computing system. Namely, operating system and machine code levels can also contribute to the appearance of misbehaviors that are usually beyond the programmer's control.

In the past fifty years, there was a huge amount of work concerning the debugging of sequential applications. Several significant debugging techniques were developed, addressing both specification and programming bugs, depending on the kind of programming models and languages (e.g., imperative or declarative). In order to analyse the behaviour of a sequential program, a state-based approach is appropriate, supported by an interactive debugger. This allows the inspection of the succession of computation states (steps), also aided by placing breakpoints at desired conditions or regions of code. Due to its deterministic behavior, it is easy to re-execute the program under a given set of input conditions in order to repeatedly examine its behavior in detail. Sequential debugging is also made simpler because the program execution follows only one thread of control.

The observation of the program execution during debugging does not change the original program behaviour, except for real-time applications.

Parallel and distributed applications introduce several distinct aspects that make them much more difficult to debug.

A *distributed program* consists of a collection of sequential processes which cooperate by using some communication model. This definition also includes the concept of a *parallel program*, although the latter term is more usually applied when there is a need to meet the application performance requirements by exploiting simultaneous execution of program units in distinct physical processors. In this text, the term *parallel and distributed* is often used, in order to highlight the use of multiple processors, on one hand, and in order to focus our attention on distributed architectures without global clock, no global shared memory, and no bounds on message transmission times, on the other hand.

The following aspects make distributed debugging much more difficult than sequential debugging:

i) The large number of concurrent and interacting entities;

ii) The intrinsic non-deterministic behavior of a distributed program;

iii) The difficulties of constructing accurate, up-to-date, and consistent observations of the global states of a distributed computation;

iv) The perturbation due to the observation and control mechanisms.

The concept of a *distributed computation* represents possible behaviors which result from executing a distributed program in a distributed systems (that is, supported by the operating system plus the hardware layers).

In order to analyse the correctness of a distributed program, a possible strategy would be to observe all distributed computations which are generated when running the program. In such a way, a set of correctness predicates can be evaluated in meaningful computation states, to give us confidence about program correctness.

In the following section of Chapter, a brief survey is presented of the theory of distributed computations in order to explain the reasons why it is so difficult to debug distributed programs. In the remaining sections of the Chapter, an overview of the main distributed debugging approaches is presented.

2.2 Distributed Computations

Depending on the programming language used, the operational semantics of a distributed program can be defined in terms of events that correspond to process control and communication actions. Such active computational entities (e.g., processes) and their state transitions, described by events, are mapped into the lower level primitive events defined by the underlying architecture of the distributed system.

Usually, for the study of distributed computations, a distributed program (system) is defined as a collection of processes that communicate using a basic message-passing model with the classical *send* and *receive* primitives. Such a system has asynchronous characteristics, with arbitrary process speeds and message transmission delays, and lacks a global physical time reference.

Such nondeterminism makes it very difficult to evaluate correctness properties that should hold for all possible executions of a distributed program, and not only for one observed execution. Also the generated computation usually follows distinct execution paths when repeatedly running the same distributed program, with a given set of input conditions.

There are two main concepts for helping us to describe all possible execution runs of a distributed program. One is the concept of local history of each sequential process that is involved in the execution of the distributed program. The other concept is the causal precedence ordering of events, defined by the sequential process ordering and the event dependences originated in process interactions.

A *process* P_i is defined as a sequence of *events*, which defines its local history h_i. Two main types of events are considered: *internal* events represent local state transitions made by Pi alone, not involving any other processes; *interaction* events represent process communications corresponding to message send and receive actions. The totally ordered events in P_i's local history represent the evolution of the values of all the P_i's variables and of the interactions involving P_i in a distributed execution.

$h_i = \{e_i^0, e_i^1, ..., e_i^f\}$

A process starts with its event e_i^0, that is the initialization event of P_i. It defines the process initial state, denoted by s_i^0. In general, the k^{th} event in the process history, denoted by e_i^k, produces the local state s_i^k, as the state immediately right after e_i^k occurrence. One can assume e_i^f is the termination event of P_i, and s_i^f is P_i's final state.

A prefix of h_i, for example up to and including the k^{th} event, is denoted by h_i^k and it represents the partial history of P_i, up to a certain point in P_i's computation.

A *global history* (H) is defined by the union of all local histories.

A fixed number (n) of processes is usually assumed without loss of generality. Among all the event orderings represented by H, only some of them can possibly occur that are compatible with the causal precedence relationship ($\mapsto$) as defined by Lamport [Lam78]. Event $e \mapsto e'$ iff e causally precedes e'. Event $e \,||\, e'$ iff neither $e \mapsto e'$ nor $e' \mapsto e$.

A distributed computation is formally defined as a partially ordered set (poset) defined by the $(H, \mapsto)$ pair. Intuitively, this reflects all physically feasible event combinations that must be obeyed by all possible executions of a distributed program by a distributed system. Distributed computations may be represented by a process-time diagram where the event causality chains replace the classical notion of instant physical time in a centralized system with a global clock.

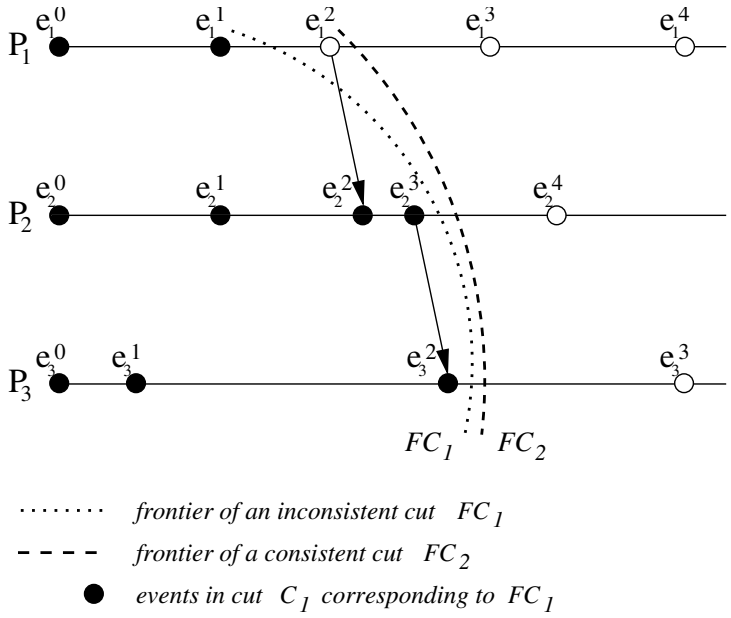

Figure 2.3: Process-time diagram with consistent and inconsistent cuts

Distributed debugging relies upon the observation of the global states of a distributed computation. A *global state* is a n-tuple of local states of all involved processes.

$S = s_1 \cup s_2 \cup \ldots \cup s_n$

where s_i is the local state of P_i ($1 \leq i \leq n$) corresponding to some prefix of P_i's local history. The initial global state (denoted by S^0) of a distributed computation is defined by the initial local states of all processes i.e. s_i^0 for $1 \leq i \leq n$. The final global state of a distributed computation (denoted by S^f) is defined by the final local states of all processes i.e. s_i^f for $1 \leq i \leq n$. The difficulty with the *intermediate global states* is that all combinations of local state tuples cannot occur in real executions of a distributed program.

In relation to a process-time diagram like in Figure 2.3, the concept of a *cut* is defined as a subset of the global history, that represents a partial global history. The *frontier of a cut* is the n-tuple of the last events in each prefix of h_i for all $1 \leq i \leq n$. The

frontier of a cut intuitively represents a view of the global progress up to a certain point in the execution in terms of the last occurred events. For example, in Figure 2.3 on the facing page there is a well-defined unique global state corresponding to each frontier of a cut, that gives the last occurred local states for each process.

However, only *consistent cuts* are significant for the purpose of meaningful observations. A consistent cut is *left closed* under the $\mapsto$ relationship, i.e.

$\forall e, e' \in H : e \in C_c \wedge e' \mapsto e \Rightarrow e' \in C_c$

Intuitively, a consistent cut incorporates all the past of its own events. A cut that would include some event e and not all events causally preceding e, cannot correspond to a possible view of a distributed execution.

A *consistent global state* is the global state defined by the *frontier of a consistent cut*. A consistent global state represents a global state that can possibly occur during a distributed program execution because it represents a view of the global state that respects the causal precedence among events. In Figure 2.3 on the preceding page FC_2 is a consistent global state and FC_1 is not.

The consistent cut and consistent global state concepts can be used as a basis to define observation models for distributed computation that can be used for distributed debugging purposes. An intuitive notion of the *current state* of a distributed computation can be visually caught by considering the events (and states) to the *left* of a consistent cut, as equivalent to a past history, and the events to the *right* of a consistent cut, as the ones in the future. This suggests one could consider an incremental progression of the distributed computation, followed by the user under the control of a distributed debugger, where *successive* consistent global states would be examined for evaluation of correctness predicates. Indeed this is an important research direction in distributed debugging, but it has several inherent difficulties that will be discussed in the following.

In order to understand the behavior of a distributed program one has to consider all intermediate consistent global states that can possibly occur starting by the initial state S^0 until the final state S^f. For each execution of a distributed program, a distinct set of consistent global states may be followed so each execution generates a distinct sequences of states, due to the nondeterminism of a distributed system. However, to ensure correctness, one needs to reason in terms of all such possible sequences of consistent global states.

The concept of *consistent run* represents a possible observation of a distributed computation where all the events appear in a total ordering that extends (i.e. is compatible to) the partial ordering defined by Lamport's causal precedence relation.

The arbitrary event ordering in a consistent run is due to the nondeterminism. In order to generate all possible sequences of consistent global states, one has to consider the set of all possible consistent runs, that is the set of all paths from S^0 to S^f [BM93]. An exhaustive traversal of such paths would be necessary to verify or detect correctness

properties of a distributed program. This approach is in general infeasible due to the large combinatory of global states that would have to be examined. Moreover, the problem of constructing individual global states poses additional difficulties.

More complete presentations of these concepts may be found in [BR94,CL85,Mat89]

2.2.1 Observation of Global States

The intuitive notion of global state of a distributed computation corresponds to a collection of local states that could be viewed by some ideal external observer. In a distributed system, an external observer can only build such a view through message exchange with each remote individual process. The following aspects are related to this observation problem:

i) The global state can be *obsolete* at the time the global view is actually constructed by the external observer. This occurs in case the observation is performed online, during actual execution. If the observation is performed offline, in a *postmortem* analysis of the global histories, this problem does not arise.

ii) The observed global state must be a consistent cut of the distributed computation. Observation of inconsistent cuts may occur due to the unpredictable message delivery orderings in a distributed system. An inconsistent sequence of events may be built by the observer that does not preserve the causal precedence relationship. Algorithms to build consistent cuts are thus required [BFR95, CL85].

iii) Multiple independent observers may build distinct views of the same distributed computation. The presentation of uniform views of a distributed computation to multiple concurrent and independent observers requires an adequate coordination between them.This is an issue that has not been considered in most of existing distributed debugging tools. However, it has high relevance due to the emergence of integrated development environments where several concurrent tools act as observers (and sometimes controllers) of an ongoing distributed computation.

The difficulties of the observation depend on the adopted distributed debugging approach:

i) *Off-line*. In this approach, it is possible to analyse global histories that were generated by a previous execution or by a simulation of the program model. These methods always deal with complete histories.

ii) *Online*. In this approach, it is necessary to develop algorithms to construct global states or consistent runs during an actual execution. These methods deal with partial histories.

The main approaches to construct observations of a distributed computation use an online external observer or monitor process. All existing approaches make specific assumptions on the message delivery rules that should be enforced by the distributed system, ranging from FIFO ordering between pairs of processes to causal delivery of messages. A discussion of the implementation of such delivery rules is beyond the scope of our work. A complete survey may be found in [BM93, Clá03].

2.2.2 Detection of Global Predicates

A general method underlies the work by several authors to support the distributed debugging activity, according to the three following steps.

i) Global predicate specification. This step starts by the identification of desired or undesired program properties corresponding to a set of correctness criteria. These properties are then expressed as global predicates which are boolean expressions involving conditions on the local variables of multiple processes or on the states of communication channels.

ii) Evaluation of global predicates. This step is responsible for the detection of global predicates using off-line or online approaches. The problem of evaluating general forms of global predicates has been studied and found NP-hard, so several authors have focused on the evaluation of restricted forms of global predicates, such as conjunctive and disjunctive. Although restricted, such global predicates are still useful in distributed debugging. An important distinction is established among *stable properties*, such as deadlock and termination, and *unstable properties* of a distributed program, which may dynamically change their truth values during the computation. The detection of unstable properties is obviously more difficult. It cannot be ensured by online observations based on the global snapshot approach, as the constructed state may miss the point of the computation where that property holds. Concerning the online construction of consistent run, even if the property holds for a certain consistent global state in that constructed run, this does not gives information about how it behaves in other possible runs. Extended forms of global properties have been proposed by several authors that try to express the program behavior in terms of the entire distributed computation, instead of related to a single global state.Several authors have exploited approaches for building and traversal the entire space of consistent global states, which are adequate for evaluation of both stable and unstable predicates. Other authors have tried to exploit specific and simplified forms of global predicates, e.g., consistent global predicates, in order to avoid an exhaustive search of that space. These approaches are further discussed in [CG98, CM91, HPR93, BM93, GCMK96, TG93].

iii) Reaction on detection of a global predicate. Depending on the user interpretation of the logical condition that was evaluated, a particular action may be necessary.

For example, if the detected global predicate corresponds to a bug situation, a distributed debugger should be able to stop the execution and restore the local states of all processes in a meaningful consistent global states that satisfy the detected global predicate.

2.3 Distributed Debugging Methodologies

Distributed debugging methodologies can be classified according to the level of support they provide to the user concerning the activities of global predication specification and detection, and the search for the causes of the bugs.

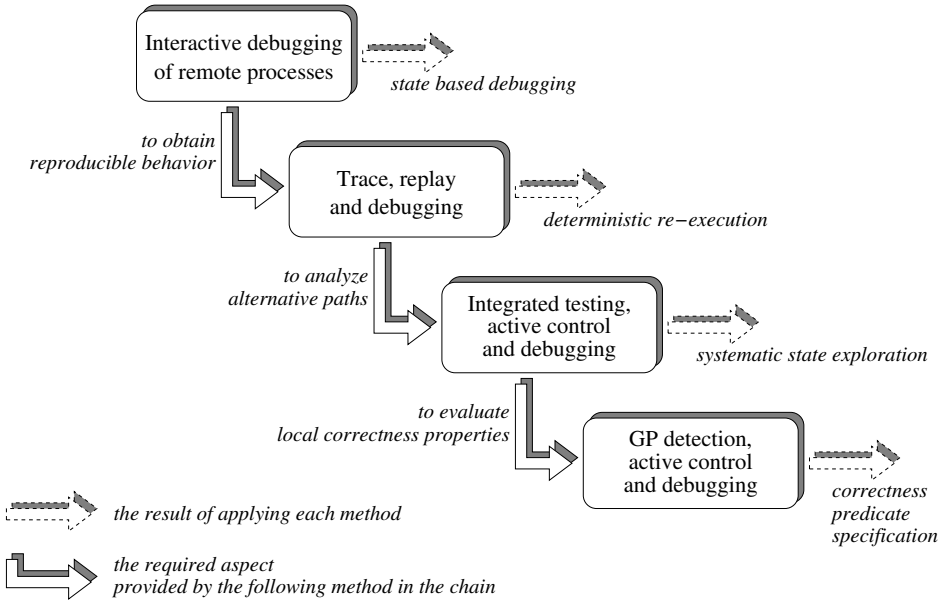

Figure 2.4: distributed debugging methodologies

In the following, these approaches are successively discussed, starting from the simpler approaches to the more complex ones. These approaches are complementary to each other, in the sense that each approach tries to overcome a limitation of the previous approach in the sequence.

2.3.1 Interactive Debugging of Remote Processes

Conventional sequential debugging commands can be extended to allow individual online observation and control of the execution of remote processes. This is a limited approach that only allows to examine local histories of individual processes of a distributed program. As each local history only describes the evolution of each process in terms of its internal and interaction events, it is the programmer's task to build the global picture of the corresponding distributed computation. However, as such basic remote debugging mechanisms are required to enable more sophisticated approaches, they are supported by almost all existing commercial or academic debuggers. The

main distinction between existing distributed debuggers of this kind is related to the functionalities and design of their architectures.

This approach is important as a first step. However, it does not handle the nondeterminism behavior of a distributed program.

2.3.2 Trace, Replay and Debugging

In order to address the nondeterminism, this approach is based on collecting a trace of the relevant events generated by a distributed computation, during a first program run. The trace describes a computation path (a consistent run) that can be analysed at a *postmortem* stage. If erroneous situations are found, the program can be re-executed under the control of a supervisory mechanism. This mechanism uses the traced sequence of events to force the execution to follow the same path as the ones in the previous run. This allows the user to examine the behavior of that path within a cyclic interactive debugging session, in a reproducible way. In such a session, the user may use the observation and control functionalities provided by the previous approach. The trace and replay technique has been the focus of intensive research in the past decade, mostly concerning the reduction of the probe effect and of the volume of the traced information [Net94, Net93, FCdK95, LMC87, RK98, Wit88, RBC+03]. However, not all commercial debuggers include such a facility.

From the view point of distributed debugging, there is a limitation in this approach if it gives no support to analyse other computation paths besides the traced one. If the first run which is used to collect the trace is a 'free' run i.e. under the control of no supervisory mechanism, the resulting trace describes only a randomly occurring path from the large set of possible paths. This gives no guarantee that such is an (the) interesting path to consider for analysis. Indeed, it is highly unlikely this will be the case.

Although this approach improves on the first one, it still needs to be complemented by the following approach.

2.3.3 Integrated Testing, Active Control and Debugging

This approach tries to overcome the above mentioned limitation of a simple passive trace and replay approach. Multiple authors have proposed approaches for the active control of distributed program execution for debugging purposes. They try to provide a facility to enforce the execution of specific runs of a distributed computation in order to ease the location of erroneous situations. They differ in the way they generate and specify the desired consistent run that a controlled execution should follow. In the following, one of these approaches is briefly described for illustrative purposes.

The approach considers two separate phases in the distributed debugging activity. It is based on the integration of a static analysis and testing phase and a dynamic

analysis and debugging stage. The goal of the testing phase is to assist the user in the generation of interesting runs that may exhibit violations of correctness properties. In general it is not feasible (or even possible) to provide a completely automated testing phase. An interactive testing tool is useful to cooperate with the user to specify and refine the conditions and regions of program code that should be considered for analysis. The testing phase is then used to generate a sequence of commands that will be used to drive a program run, in order to exercise the paths defined by the above testing scenarios. Such a run can then be the subject of a trace and replay approach, and integrated in a cyclic debugging session.

The main advantage of this methodology is that it allows the user to interactively 'walk' through the testing and debugging phases, until one is convinced about the satisfaction of the correctness properties that are being investigated. Another advantage of this approach is that it combines the advantages of static and dynamic analysis in order to help the user to understand program behavior (cf below).

The main problem with this approach is that it basically relies upon the user conviction that all relevant scenarios were specified and generated, tested and analysed, so that one gets confidence on distributed program correctness. There is no full guarantee that no important situations went unnoticed. Still, this approach has been the basis of intensive research and has produced interesting results [LCK+97].

2.3.4 Automated Detection of Global Predicates

This approach is an attempt to help the user increasing the confidence on the results of the previous approach, by allowing the specification of the correctness criteria in terms of global predicates. Such global predicates are then automatically evaluated by detection algorithms, working off-line or on-line distributed debugging [Bat95, Bat88, CG98, CM91, Clá03]. As the efficient evaluation of global predicates is limited to restricted classes of global predicates, this approach may be seen as complementary to the testing and debugging approach. Their integration seems a promising research direction to improve.

2.3.5 Distributed Debugging Based on Static Analysis

This approach uses the program code as a basis and it does not require actual program execution. It relies on formal models of program behavior that can be used to check certain kinds of properties, usually expressed as temporal logic formulas. However, model checking techniques can only be used to analyse certain properties and do not give information on dynamic properties that depend on actual runtime program behavior, e.g., termination. Also, they usually incur great computational costs in their search for all allowable state transitions in the modelled computation space.

Still, static analysis of the program source code is one approach that can reveal

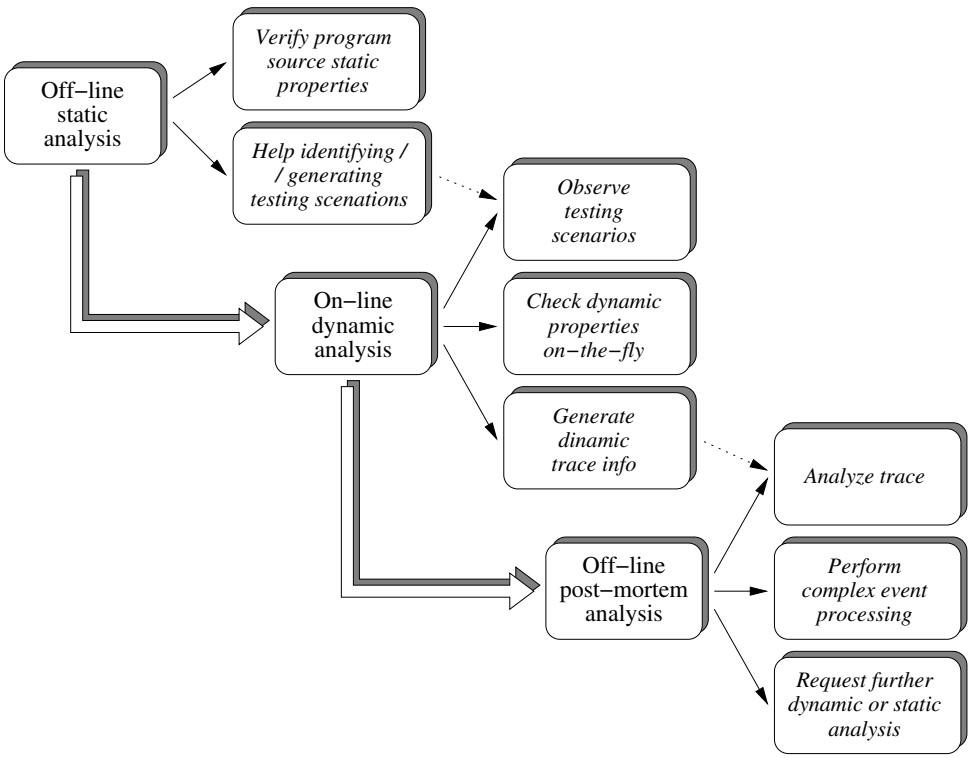

Figure 2.5: distributed debugging methodologies

itself as of great importance for distributed debugging, if adequately combined with complementary approaches.

2.3.6 Distributed Debugging Based on Dynamic Analysis

Due to the mentioned limitation of static analysis, one needs to use online approaches that help evaluating the actual program behavior on-the-fly. Such approaches rely upon online observation techniques so they must deal with the difficulties of accurate construction of consistent global states. Once a specific program behavior pattern was detected, these approaches also require adequate control mechanisms to help the user inspecting the individual computation states of interest. This approach must deal with the probe effect, in order to ensure that the observed computation path exhibits the same logical behavior as the original computation would, when running with no observation mechanisms.

Dynamic and static analysis approaches can be combined in order to provide the distributed debugger with functionalities as the ones required by the mentioned integrated testing, active control and debugging approach.

2.3.7 Distributed Debugging Based on *Postmortem* Analysis

Postmortem analysis approaches provide an effective way to analyse program behavior because they rely upon previously collected traces of the processes' local histories. On one hand, it becomes easier to construct a consistent global state, out of these local

histories, by regenerating the causal precedence chains. This reduces the runtime overhead incurred by online approaches. It also enables facilities for analysis of complete computation histories, with the help of a diversity of event analysis and visualization tools. On the other hand, *postmortem* techniques can be integrated with online techniques, in order to exploit tracing, replay and debugging methods, to address the non reproducibility issue. Incremental methods consisting of online and *postmortem* stages also allow to handle the potentially large volume of traced information. A first run is used to collect only the minimum amount of information to ensure reproducible re-execution, and further *postmortem* analysis can determine the need to collect further information on successive runs.

The summary of how such approaches are complementary to each other:

i) *Off-line.* Verify certain properties using static analysis and help identifying relevant scenarios for testing

ii) *Online*. Check dynamic properties on-the-fly, and observe testing scenarios, under an actively controlled execution

iii) *Postmortem*. Analyse traces of complete global histories, perform more complex event processing (e.g., high level event abstractions) and visualization. Use the results of such analysis to determine further runs and dynamic analysis.

Further discussion on the classification of distributed debugging approaches can be found on [CLD01a].

3

Fiddle: a Distributed Debugging Engine

Contents

This Chapter introduces the main requirements for distributed debugging and how traditional debugging services fulfill some of those requirements, followed by presentation of the software architecture of a debugging engine which fulfills some of those requirements, and how this debugging engine may be extended with complementary functionalities which may cover the remaining requirements

3.1 Introduction

There is a large diversity of approaches commonly used to locate bugs in programs, depending on the hardware, the operating system, the programming model, the programming language and, most of all, on the program developer.

Traditionally, the software developer has the possibility to execute a program under control of a debugger. This allows to inspect the process core image, e.g., the memory contents, and to have a detailed control over program execution by using techniques such as breakpointing and single-stepping. However, they are less effective for distributed programs, composed by multiple processes executing on multiple computing nodes, due to the intrinsic non-determinism of the corresponding computations.

3.2 Techniques for Distributed Debugging

Errors in distributed programs may, naturally, be due to unplanned process interactions as well as errors in sequential code in any of the processes. Distributed debuggers should not neglect the latest kind of errors as, although easier to locate and correct, their frequency in distributed programs is much higher then those resulting from unwanted process interactions.

Different classes of techniques may be used to debug distributed programs:

i) *Sequential debugging techniques*. Distributed programs also include sequential sections of code, and such sections are not, necessarily, bug free. Sequential debugging services, such as breakpointing, single-stepping and inspecting/changing process variables, are also a basic requirement for distributed debugging;

ii) *Distributed debugging techniques*. Distributed programs are composed of interacting processes, whose behaviors reflect mutual interdependencies. The need to understand these interdependencies puts specific requirements upon distributed debuggers;

iii) *Information sharing and tool cooperation*. Distributed programs are considerably more complex than their sequential counterparts, and the software developer frequently relies on different tools to analyse the distributed program behavior. Using such tools separately forces the software developer to a permanent "context switch" in the working environment every time a different tool is used. Most important, this puts a strong demand upon the software developer to analyse and correlate the data gathered from multiple tools.

   This motivates the requirement to support a set of software tools which are able to cooperate with each other, by exchanging data and control information and, if possible, to coexist and be accessed through uniform and consistent user interfaces.

The above identified techniques to support distributed debugging are discussed in the following sections.

3.2.1 Sequential Debugging Techniques

Sequential debugging techniques can be applied to the debugging of the sequential sections of code of a distributed program. These techniques can be organized into the following main classes of services.

i) *Breakpointing services*. To stop program execution at points corresponding to specific source code locations or on read/write accesses to specific memory locations;

ii) *Control services*. To control the program execution, through the classical *step*, *next* and *continue* operations;

iii) *Data services*. To examine and change the program core image, for example, to access local and global variables contents;

iv) *Stack services*. To inspect and manipulate the program execution stack;

v) *File services*. To inspect and control the source and executable files associated to a target program;

vi) *Management services*. To support the management of the debugging tool itself, such as initiating and terminating the distributed debugging tool.

These services allow the software developer to examine and debug the individual processes of a distributed program, but do not consider the additional needs resulting from distribution and process interaction, which are covered by the next class.

3.2.2 Distributed Debugging Techniques

The software development environments, and the distributed debugging system in particular, must address the difficulties introduced by distributed computations. Concerning the debugging activity, the following list of functionalities contribute to handle such difficulties.

i) *Behavior record and replay*. Due to the inherent non-determinism of distributed computations, the simple act of observing a distributed computation may modify its behavior, changing or even hiding a bug manifestation. This is frequently called the *probe effect* or the *Heisenbug effect*.

The undesired effects of such changes in the program behavior can be reduced by registering and recording the distributed program behavior in a first run. The same distributed program can then be replayed, i.e., re-executed and forced to follow the same steps which were registered in the first run. This allows to perform

the required inspection and even control operations during the replayed execution.

ii) *Controlled execution*. A specification of the intended program behavior can be generated from annotations introduced in the source code by the software developer, or by a testing tool which analyses the distributed program and its source files.

Inspecting a distributed computation generated by a distributed program under controlled execution will not influence its behavior, as it is being forced to follow well specified paths corresponding to a specification of such intended behavior;

iii) *Checkpointing*. For long time runs, it may be unacceptable to re-run the application from the very beginning once a program failure is detected. In such cases, it is common to periodically perform execution checkpoints, where the global state of the distributed computation is saved. Once a failure is detected, the computation may be resumed to a previously saved state, and proceed from that point on;

iv) *Inspection and control of system level entities*. A distributed program frequently depends on third party software packages, e.g., communication libraries, or runtime or operating system "objects", e.g., semaphores and mutexes. The ability to examine and change such external entities is necessary for a better understanding of the distributed program behavior and, therefore, for the identification and correction of its errors;

v) *Log analysis and error detection*. By analysing a log file where the behavior of a distributed program was registered, and correlating that information to the source code, many program errors can be detected, e.g., some race conditions, even before they actually generate a fault and a failure.

The distributed debugging techniques enumerated above, when adequately supported by a debugging environment, allow the software developer to reach a better understanding and control of the distributed program behavior, and of the interactions between its processes. However, the tools providing those services should be able to share information and cooperate with each other, as discussed in the following.

3.2.3 Tool Integration Issues

The development of distributed programs encompasses a hierarchy of abstraction levels, supported by a diversity of models and tools. Such models and tools may considerably help the software developer to understand the process interactions and the global application behavior, and may include:

i) *High(er)-level programming languages*. To allow problem specification and coding;

ii) *High(er)-level programming languages editors*. To support the effective development of programs on those languages;

iii) *Code generators and compilers*. To convert the high(er)-level programs into intermediate and executable code;

iv) *Simulators*. To help predicting and evaluating the program behavior without the need of a real execution;

v) *Mapping and load balancing tools*. To define the initial distribution of processes on the computing nodes, and to balance during execution;

vi) *Performance evaluators*. To analyse and optimise the overall program performance;

vii) *Computation visualizers*. To graphically represent the behavior of computation nodes and processes, and display their interactions;

viii) *Debuggers*. To help detecting, locating and correcting program errors, by providing the means to inspect and control the distributed program behavior.

The above tools are, often, developed by different tool-makers, thus having limited compatibility and interaction capabilities. The result is, typically, a complex developing environment, where the software developer has to constantly switch between different tools to edit, compile, test and debug the distributed program.

The difficulties of using such development environments, where multiple tools from different origins and vendors are used together, motivated the efforts towards the specification of standard services and interfaces [LWSB97], and increased the relevance of tool integration and interoperability issues.

Frequently, the interoperability of a distributed debugger with other tools in a software development environment, such as graphical program editors and computation visualizers, is a key requirement.

3.3 A Proposal for a Distributed Debugging System

Many of the distributed debugging tools available from both the research and the commercial communities have a stronger emphasis on the distributed debugging services, partially (or even completely) neglecting the other two classes, the sequential debugging services and the tool cooperation and integration services.

In this dissertation we propose a different approach. We start with a minimalist debugging engine, which supports some basic sequential debugging services for distributed programs. This debugging engine can thus be extended with additional services, that can be integrated in a debugging software architecture. New services can be defined by an adequate combination of those already available, or by implementing a new set of servers and libraries and linking them to the debugging engine.

We use the term "debugging engine" and not "distributed debugger" because its definition is restricted to the debugging services to be supported and not how they should and will be used. Consequently, the proposed engine does not include any specification of a built-in user interface, neither graphical nor text oriented.

By separating the definition of the user interface from the specification of the debugging engine, we allow the development of a diversity of user interfaces with different targets and goals. Other important characteristic of the proposed debugging engine is the tool interoperability and integration facilities.

In summary, the proposed debugging engine aims at:

i) *Allowing the access and control of multiple distributed processes*. Distributed programs are composed of a set of cooperating processes, and the debugging engine will be able to access and control all or some of those processes;

ii) *Allowing the access and control of multi-threaded processes*. In the last years there has been a trend towards thread-based programming models, and thread-based programming is an issue which is also addressed by the debugging engine;

iii) *Supporting symbolic process identifiers*. The debugging engine supports a symbolic process naming mechanism, allowing to abstract from the physical location of the processes under debugging;

iv) *Providing a minimalistic set of debugging functionalities*. There is such a diversity of distributed execution environments, regarding the hardware, the operating system, the communication libraries, and the distributed programimg models, that the range of supported execution environments by each debugging engine are considerably restricted. Attempts to design a debugging engine in order to accept a wider scope of execution environments may lead to a huge and unmaintainable tool. We opted to provide just a minimalistic set of debugging functionalities in the debugging engine core. Tis has the effect of limiting its size and easing its maintenance. It also allows to provide a common set of basic functionalities that may be useful in distinct environments;

v) *Being extensible*. The minimalist set of debugging functionalities may not be enough for a particular use of the debugging engine or for a particular execution environments. The debugging engine should be adaptable to particular execution environments or to support specific needs. This should be achieved by designing and implementing *debugging engine extensions* which are incorporated into the debugging engine core;

vi) *Supporting a diversity of user interfaces*. Typically, a distributed debugger includes a generic (sometimes graphical) user interface, which allows to generically debug different classes of programs, but lacks the ability to adapt to the specific needs imposed by each application or by the execution environment;

This desired adaptability and flexibility of the debugging engine can be achieved by decoupling its core from the user interface. In such a way, besides any generic debugging user interface available in the debugging engine distribution, both generic and customized debugging user interfaces can be developed and integrated into the debugging engine;

vii) *Supporting for multiple concurrent user interfaces*. Multiple debugging user interfaces to coexist and operate concurrently, being involved in the task of debugging the same distributed program. This is achieved by keeping each debugging user interface as a small and simple unit, and by building more complex user interfaces through the composition of the simpler units.

3.4 The Debugging System Components

Three main components are involved in the debugging of a distributed program: the client tools, the debugging engine and the target application processes. These components are discussed in the following.

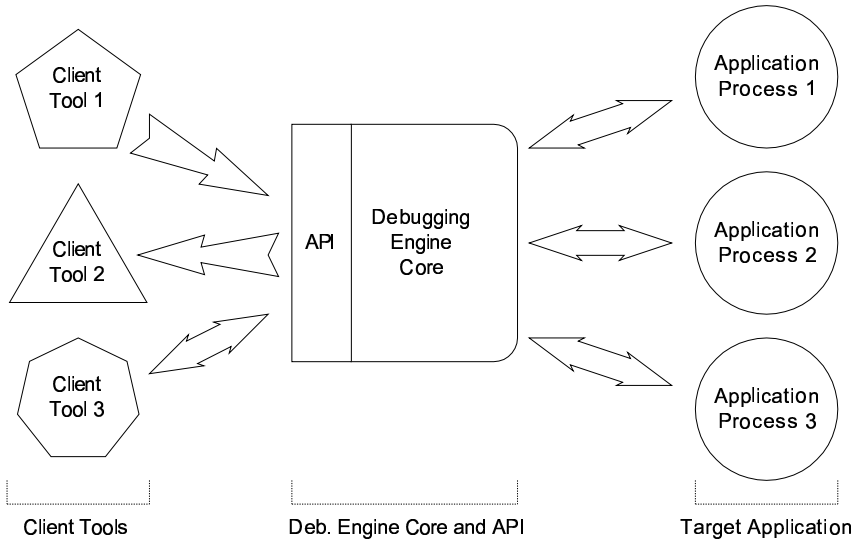

Figure 3.1: The debugging engine logical organization

i) *The target program.* The execution of a distributed program involves the execution of a set of interacting processes (possibly) running on different nodes. Debugging such a program involves observing and controlling not only some of its processes but also their interactions;

ii) *The client tools (debugging user interfaces).* Provide access to the debugging functionalities supported by the debugging engine;

iii) *The debugging engine.* Implements the basic debugging functionalities which can be accessed from the client tools. The debugging engine is defined by:

a) *The debugging engine core.* This is the center of all the debugging operations. It manages, among other things, the status of the target processes. It also interprets the debugging service requests, and handles their application to the target processes;

b) *The application programming interface (API).* The specification of how the client tools can access the services provided by the debugging engine core.

In the following, each of these components is discussed further.

3.4.1 The Target Program and Processes

When testing a distributed program, we may believe that a certain misbehavior is being originated in one (or in a subset of) the processes of that distributed program. In this case, we may be interested in debugging just this subset and not all the processes.

A program under debugging is called *the target program*. The subset of its processes under debugging are called *the target processes*. Target processes are said to be *local* or *remote*, depending on whether they are executing in the same physical node as the client tool (user debugging interface) or in some other (remote) node, respectively.

3.4.2 The Client Tools

The debugging engine provides a set of debugging services, each set being accessible through an API, but the engine does not provide the interface for the software developer to access those services. Each client tool must implement this bridge between the software developer and the debugging engine, providing a user interface to access a subset of the services available in the debugging engine.

The debugging engine does not specify the kind of functionalities to be provided by the client tools, neither how will they be accessed and presented to the software developer. The functional and operational specifications of the client tools are completely left open to their developers.

In Figure 3.1 on the preceding page, three different kinds of client tools are depicted:

i) *A controller* (client tool 1), whose main function is to act upon the target application, by changing its internal state or controlling its behavior;

ii) *An observer* (client tool 2), whose main function is to collect and display information about the application status and behavior; and

iii) *An interactive tool* (client tool 3), which acts simultaneously as both a controller and an observer.

The debugging engine services are made available to all the client tools through an API (described in Section 3.4.4). In our experimental work, such services were actually explored by a wide range of tools, from text oriented and graphical debugging

interfaces, such as Fiddle consoles (described in Section 5.3) and FGI (described in Section 5.4), to automated debugging tools, such as DEIPA (described in Section 5.5), and even to parallel software development environments, such as GRADE and EDPEPPS (both described in Section 5.6).

According to the above classification for the client tools, DEIPA is a possible example of a controller. Any program visualizer, such as Pajé (whose possible cooperation with our debugging engine is discussed in Section 5.7), may be classified as an observer. Finally, any debugging interface, such as Fiddle consoles and FGI, belongs to the interactive tool category.

3.4.3 The Debugging Engine Core

The debugging engine core is internally organized in multiple functional layers, as depicted in Figure 3.2. The set of services available successively extend the debugging functionalities as we move from the lower to the higher layers of the debugging engine.

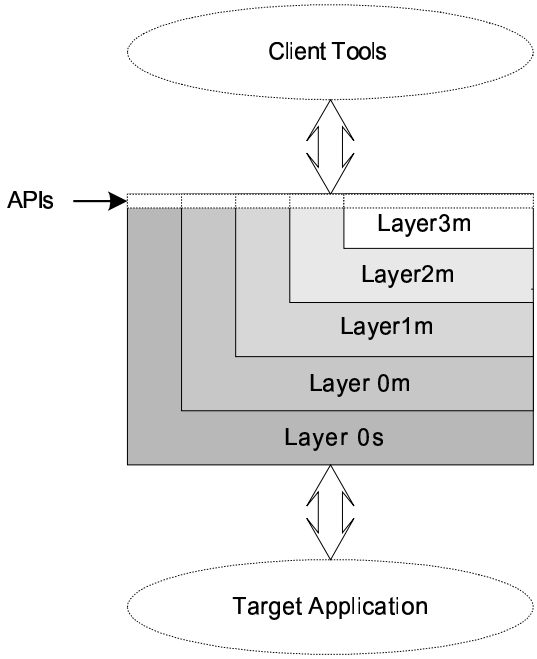

Figure 3.2: The debugging engine layered architecture

The functionalities provided by each layer have a well defined API that gives access to an associated *service library* (there is one service library for each layer). There are only minor differences between the APIs of all layers. The exception is Layer 3_m, which differs considerably from the others, both syntactically and semantically, due to the nature of the services it provides.

There is a minimum set of functionalities, common to all layers, which are always available: support for one or more client tools (depending on the layer being used), support for multiple target processes (both local and remote, also depending on the layer being used), where each of the target processes may be single- or multi-threaded.

Besides the above common features for all the layers, there are some others which are specific to each layer, as summarized in Table 3.1.

	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 client(s)	No	Yes	Yes	Yes	Yes
Remote debugging	No	No	Yes	Yes	Yes
Multiple concurrent clients	No	No	No	Yes	Yes
Events and call-back routines	No	No	No	No	Yes

Table 3.1: The debugging engine internal layers and their functionalities

Any layer may be directly used by a client tool, as long as the set of services provided by that layer are enough for the tool needs.

i) *Layer* 0_s. Provides debugging services at a single local node. By *local* we mean that both the target processes and the client tool must be executing in the same computing node. Only a single thread in the client tool may be issuing service requests to this layer. The debugging engine gives a symbolic identifier to each target process, and this identifier must be used by the client tools to identify each process that is targeted by their service requests;

ii) *Layer* 0_m. Extends the services provided by **Layer** 0_s to guarantee a thread-safe environment. When accessing this layer, a multi-threaded client tool may issue concurrent requests to the debugging engine, which will be executed concurrently if directed to different target processes, otherwise they will be applied sequentially;

iii) *Layer* 1_m. Extends the services provided by **Layer** 0_m to provide transparent debugging services to remote target processes. At this level all symbolic identifiers are made global in the distributed system and can, therefore, refer to any process independently from their physical location;

iv) *Layer* 2_m. Extends the services provided by **Layer** 1_m to allow multiple concurrent client tools. By using this layer, it is possible to have as many client tools as desired, all of them concurrently issuing debugging requests to the same set of target processes. Such multiple client tools may provide complementary views and debugging functionalities over the target program, by exploiting and manipulating different (but not exclusive) sets of debugging services;

v) *Layer* 3_m. Was designed to support a *common shared knowledge* to all client tools. This common shared knowledge would include the status of the target application and of the debugging engine itself. To accomplish this goal, an event notification mechanism was introduced by this layer, such that changes in the target

application data or execution state, or changes in the state of the debugging engine core, may originate events which will trigger the execution of event handlers in the client tools. Client tools may react to those events by changing their own state or by storing the event details in an internal database for later access.

The hierarchical structure of the debugging engine architecture implies that each layer $\mathcal{L}_{i\ (i>0)}$ is a direct client of layer $\mathcal{L}_{i-1}$ (see Figure 3.3). In this figure, Layer 3_m has two client tools ($CT_2^{3_m}$ and $CT_1^{3_m}$), Layer 2_m also has two client tools $CT_1^{2_m}$ and Layer 3_m), and each of the remaining layers has a single client tool (the immediately above layer).

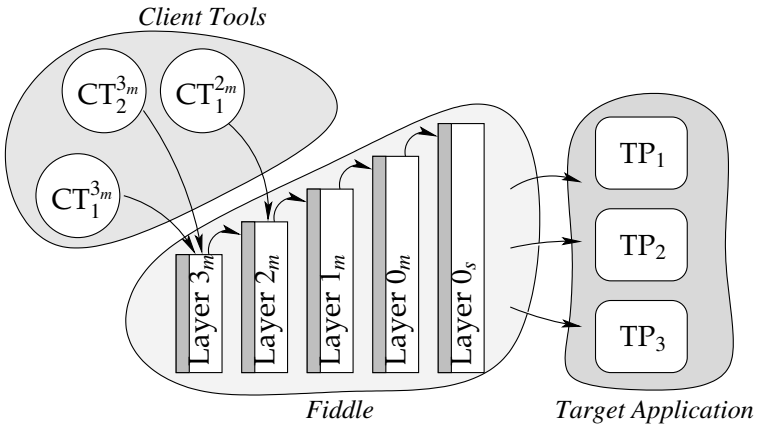

Figure 3.3: The debugging engine logical layers

When a service is requested by a client tool, it will be interpreted and passed to the successively underlying layers, until Layer 0_s is reached. At this point, the request is applied to the target process. The result of such an operation is also successively passed back to the upper layers until the client tool gets the reply. In the meanwhile, the invoking thread in the client tool is blocked waiting for the reply, except for a Layer 3_m client, which may define a reply handler for this service request and proceed with its computation.

3.4.4 The Debugging Engine API

All the services provided by the debugging engine core have well defined semantics and application programming interfaces. Such services provide a set of basic functionalities which operate upon individual processes and may be classified into the following main categories:

i) *Internal services*. Management services related to the debugging engine itself and not to the target program;

ii) *Breakpointing services*. To set different kind of breakpoints in the target processes, so that they will stop at specific code locations, unconditionally or when certain conditions are verified;

iii) *Running services*. To control the execution of target processes, such as single-stepping or proceed with program execution until a breakpoint is reached;

iv) *Data services*. To examine or change the target processes core state, such as local and global variables;

v) *Stack services*. To inspect and manipulate the target processes' stack contents;

vi) *File services*. To inspect and control the source files associated to the target processes;

vii) *Thread services*. To operate upon process threads;

viii) *Miscellaneous services*. For those services that do not fit in any of the previous classes.

Other classes of services, such as operations upon groups of processes, record and replay of distributed computations, checkpointing, etc. . . may be implemented as extensions to the debugging engine core and its API. The works on RPVM [LC98b] and PADI [SNC00] are examples of such extensions to the debugging engine. RPVM extends the debugging engine with the ability for deterministic replay of distributed programs. PADI extends the debugging engine with a set of services which operate upon process groups, and a graphical interface to access those services.

Similar services at different layers have only minor differences in their syntax and/or semantics. For example, services `f0m_step()` and `f1m_step()` only have a minor semantic difference, as the former is only applicable to local processes and the latter also applies to remote processes; and services `f0m_attach()` and `f1m_attach()` have minor differences in both syntax and semantics, as the latter receives one more argument than the former.

The API for all layers, except Layer 3_m, have an invocation semantics which follows a synchronous model. This means that, for the majority of the layers, when a thread in a client tool issues a debugging request by invoking a function from the API, it will remain blocked until the debugging engine handles the request and generates a reply. The handling of a service may require the arguments of the called service to be encoded, packed and transfered to a remote node, and the reply to follow a similar path, but in the opposite direction. Once received, the reply is unpacked, copied to the output arguments of the API function call, and passed back to the caller.

Table 3.2 on the facing page presents a list of the basic services provided by all layers. For each functional layer of the debugging engine there is a programming library which implements its API. For each library, the "X" in the service name prefix on Table 3.2 is changed to the layer's name, e.g., for Layer 1_m API, `fX_initialize()` is changed to `f1m_initialize()`. Full details of the syntax and semantics of these services are presented in [LC99].

Internal services	
`fX_initialize()`	Initialize the debugging engine
`fX_terminate()`	Terminate debugging engine
`fX_clients()`	List active clients of the debugging engine
`fX_tids()`	List the IDs of currently available target processes
Breakpointing services	
`fX_break()`	Set a breakpoint
`fX_delete()`	Delete a breakpoint
`fX_info_break()`	Get info about current breakpoints
Running services	
`fX_attach()`	Attach to a running process
`fX_detach()`	Detach from a running process
`fX_kill()`	Kill a target process
`fX_file()`	Load program and symbols into memory
`fX_symbol_file()`	Specify where to find program symbols
`fX_run()`	Run a program
`fX_step()`	Execute until the next instruction
`fX_next()`	Execute until the next instruction (considering a function call as a single instruction)
`fX_continue()`	Continue the execution
`fX_finish()`	Execute until returning from current stack frame
`fX_call()`	Call a function in the current context of the process
`fX_signal()`	Send a signal to a process
Data services	
`fX_set_variable()`	Change the contents of a variable
`fX_evaluate()`	Evaluate an expression in the current context of the target process
`fX_display()`	Evaluate and display an expression every time the process execution stops
`fX_undisplay()`	Undisplay an expression
`fX_info_display()`	Get info about current display expressions
`fX_info_locals()`	Get name and value of local variables
`fX_info_args()`	Get name and value of function argument
Stack services	
`fX_info_stack()`	Get info about current stack frames
`fX_up()`	Go up in the stack frame list
`fX_down()`	Go down in the stack frame list
`fX_frame()`	Select a specific stack frame
File services	
`fX_list()`	List the process source code
`fX_info_line()`	Get info about current line
`fX_info_program()`	Get info about current program
Thread services	
`fX_thread()`	Select a thread
`fX_info_threads()`	Get info about the existing threads
Miscellaneous services	
`fX_tty()`	Set a TTY for future IO
`fX_sendto()`	Send a command directly to a node debugger, bypassing the debugging engine

Table 3.2: The debugging engine API

3.5 The Architecture of the Debugging Engine

The debugging engine is, itself, a distributed program, consisting of a set of processes and libraries which cooperate to provide a set of distributed debugging services. The debugging engine is structured in functional layers whose functionalities and architecture are discussed in detail along this section.

3.5.1 Layer 0_s

This is the first (lowest) layer. It defines a set of debugging functionalities available in a single node (node-level services), namely:

i) Attach and detach the debugging engine to/from processes running on the local node;

ii) Launch new processes under the control of the debugging engine;

iii) Inspect the target processes, which may be single- or multi-threaded processes;

iv) Manipulate the execution status and memory map contents of the target processes, through operations such as breakpointing, single-stepping and the change of program variables.

This layer also provides some additional functionalities besides the above ones:

iv) *Symbolic naming of local processes*. For each target process, a symbolic identifier is generated by the debugging engine which will be used during an entire debugging session to refer to that process;

v) *Concurrent debugging of multiple local processes*. The debugging engine can be attached to more than one process, as long as they are executing in the same physical node as the debugging engine;

All the above functionalities are available through a well defined API and implemented in a software library, which must be linked to the client tool (the debugging user interface), as depicted in Figure 3.4 on the next page.

The main components of Layer 0_s software architecture are:

i) *Node debuggers*. Attached to each target process there is a node debugger. Each node debugger carries out the debugging operations upon its associated target process. New node debuggers are dynamically launched by the debugging engine core as required and terminated when no longer needed. The debugging engine core ensures these operations are fully transparent to the user.

Any existing sequential or parallel debugger which runs on the host where the process is being executed may be used as a *node debugger*. The only requirement is

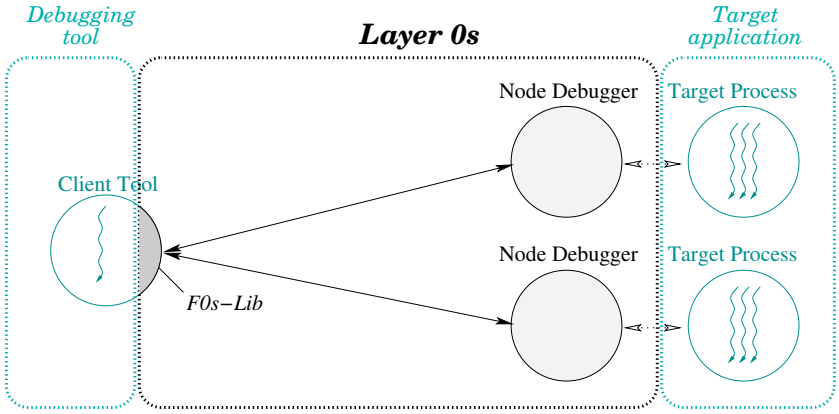

Figure 3.4: The Layer 0_s software architecture

that it supports some form of interaction with other programs, either by providing an API or by using a text oriented command language. This is, indeed, a minimal requirement for tool interaction. Still, several existing commercial tools only provide graphical user interfaces, thus are not amenable to interaction neither to be used as node debuggers in the debugging engine.

ii) *Layer* 0_s *service library* (`F0s-Lib`). This library implements the API as defined for this layer, plus all the internal management services, such as launching and terminating node debuggers whenever needed. At any time, only one thread in the client tool may be calling services from this library and, therefore, issuing service requests to the debugging engine.

To support the API, this service library must manipulate and adapt the API function call arguments, by converting them into a set of commands which are understandable by each node debugger. Also, the replies from the node debugger must be processed and transformed into return arguments to the API function call.

3.5.2 Layer 0_m

The system components known to this layer are identical to those known to Layer 0_s, as depicted in Figure 3.5 on the following page. The only difference is that there is a new service library which implements the new features supported by this layer.

There is a single new system component in Layer 0_m:

iii) *Layer* 0_m *service library* (`F0m-Lib`). This service library stands between the client tool and the Layer 0_s library (`F0s-Lib` in Figure 3.5 on the next page), provides thread-safe access to the debugging engine.

For example, multi-threaded client tools may explore this layer by using an individual thread to control each target process. Each of these threads may independently issue service requests to the debugging engine.

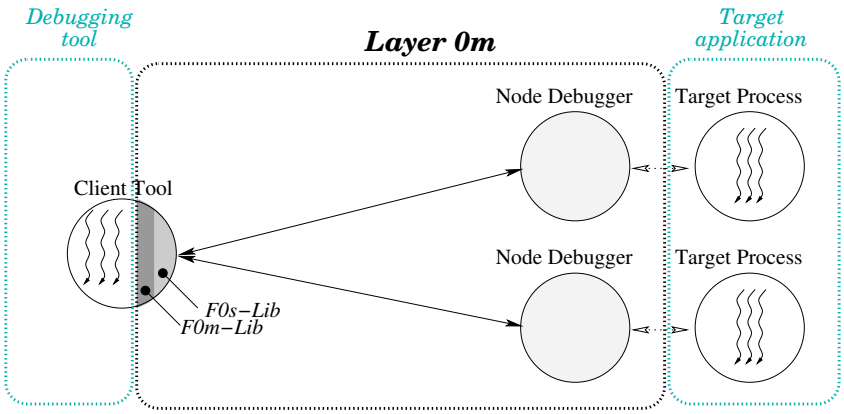

Figure 3.5: The Layer 0_m software architecture

3.5.3 Layer 1_m

This layer extends Layer 0_m by providing support for the debugging of remote processes. Its software architecture is presented in Figure 3.6.

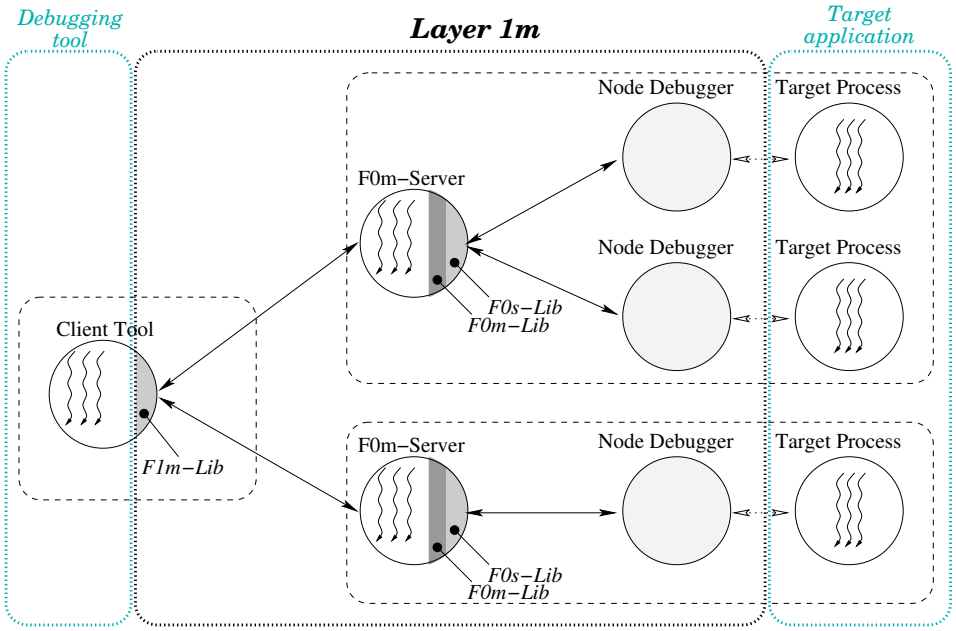

Figure 3.6: The Layer 1_m software architecture

The system components known to this layer include all of the above described for the previous layers, plus:

iv) *Node server* (`F0m-Server`). On each physical node there is a *node server* which manages all the local node debuggers. It is up to the node server to launch new node debuggers, to establish the communication channels to these new node debuggers, and to terminate the node debuggers when no longer needed. It is also up to the node server to manage the communication channel to the client tool. From the perspective of the client tool, the node server is the only intermediary for all the target processes in that node.

v) *Layer* 1_m *service library* (`F1m-Lib`). This service library provides a (multi-threaded) client tool with transparent access to remote target processes. It also introduces a symbolic global naming mechanism for all target processes.

3.5.4 Layer 2_m

This layer extends Layer 1_m by providing support for multiple concurrent client tools. Its software architecture is presented in Figure 3.7.

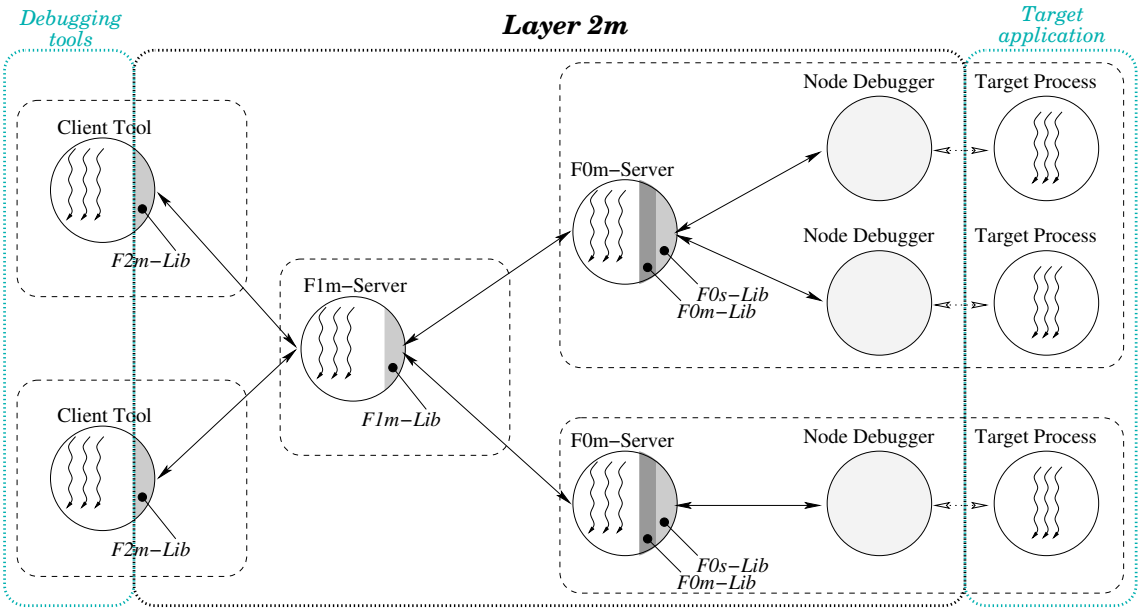

Figure 3.7: The Layer 2_m software architecture

The system components known to this layer include all of the above described for Layer 1_m, plus:

vi) *Main server* (`F1m-Server`). The main server acts as the access point to the debugging engine. It centralizes all the client connections at one point, and plays the role of a router, forwarding the service requests to the appropriate node server and resending the replies back to the client. It should be noticed that no heavy processing is done in this main server, avoiding it to become a bottleneck and allowing the number of clients and servers to scale;

vii) *Layer* 2_m *service library* (`F2m-Lib`). This service library provides a (multi-threaded) client tool with a non-exclusive access to the debugging engine. In this way, multiple client tools may be concurrently operating upon the same set of target processes, possibly (and hopefully) providing the developer with complementary views of the target application and extended debugging functionalities.

3.5.5 Layer 3_m

Layer 3_m is only partially defined. It aims at supporting an asynchronous calling model. Services requested to this layer will never block the caller, but return immediately a *service-request-id* to the client tool.

Such request-id may then be used by the client tool to inquire the debugging engine about the status of the associated service, or given as an argument to a call-back function which will be activated when the processing of the service terminates.

Call-back functions (if defined) will be executed when the processing of a service is terminated. Additionally, the client tool may also define call-back functions to be executed when the status of the debugging engine changes (e.g., there is a new client tool connected to the debugging engine) or when the status of one of the target processes changes (e.g., one process hit a breakpoint and switched from the running to the stopped state).

As such, four different classes of events relevant to the debugging activity were identified:

i) *A service was requested by another client tool* . Service requests from a client tool may, or may not, have consequences upon the other client tools. Thus, client tools should have the means to be notified and, therefore, to react, to other tools' activity;

ii) *The processing of a service request is terminated*. Some service requests may have no implications in the target application state neither in the debugging engine state. In such cases, it may be interesting to receive an acknowledge that the service was completed;

iii) *The target application state has changed*. This change may result from a change in its execution state, such when a breakpoint is reached, or from a change in its core image, such as when a program variable receives a new value;

iv) *The debugging engine state has changed*. This may be due to management tasks, such as when a new client tool connects to the debugging engine, or to program activity, such as when a new process is spawned.

Any of the above identified relevant occurrences are propagated to the debugging engine and to the remaining client tools as events. Client tools may define handlers to react to those events, changing their own state or storing event details in an internal queue/database for later processing.

Events may be processed by the client tools synchronously, where the debugging engine keeps the notification of the event pending until the client tool explicitly requests it, or asynchronously, where the tool defines event handlers, which are triggered by the event notification and executed by a new thread. In both cases, a description of

the event is passed as an argument to the handling function, so it can react appropriately to the event.

3.6 Extending the Debugging Engine

The architecture of a distributed debugging system generally imposes strong limitations on its ability to cooperate with other tools and to adapt to specific developer (or system) needs.

Multiple tools are used toward the software development of a distributed program. When these tools have different origins it is, in general, a programmer's task to make the bridge between these tools at both, conceptual and operational levels. Integrated development environments aim at providing a consistent set of tools which are able to exchange data and control information, and that ease the task of establishing the correspondence between concepts at distinct abstraction levels.

Aiming at facilitating such integrated developer environments, the debugging engine includes some relevant features, namely:

i) *Internal extensibility*. By changing the existing debugging engine libraries or by adding new ones. To keep the complexity of the debugging engine low, such kind of extensions should only be used to support new (and simple) basic services. For more complex services, external extensions should be used;

ii) *External extensibility*. By adding new modules to the debugging engine. Being less dependent from the debugging engine core, adding such modules will not increase significantly the complexity of the debugging engine. Thus, such external extensions are, therefore, appropriate to implement more complex services;

iii) *Cooperation and integration ability*. The debugging engine supports multiple client tools acting concurrently upon the same target processes. By making use of such feature, other software development tools may have access to debugging services by registering themselves as client tools of the debugging engine, share information with the debugging engine and get notifications from the debugging engine on state changes in the target processes.

In the following we discuss how such features are supported by the software architecture of the debugging engine.

3.6.1 Internal extensibility

The debugging engine is structured in functional layers and new layers can be added to provide additional functionalities. In this sense, $\mathsf{Layer}\,0_s$ is extended by $\mathsf{Layer}\,0_m$ which, in turn, is extended by $\mathsf{Layer}\,1_m$, and so on. Additional facilities, such as process and

thread groups management or API wrappers to other languages, can easily be added to the debugging engine as (internal) extensions. Figure 3.8 illustrates how internal extensions are incorporated into the debugging engine software architecture.

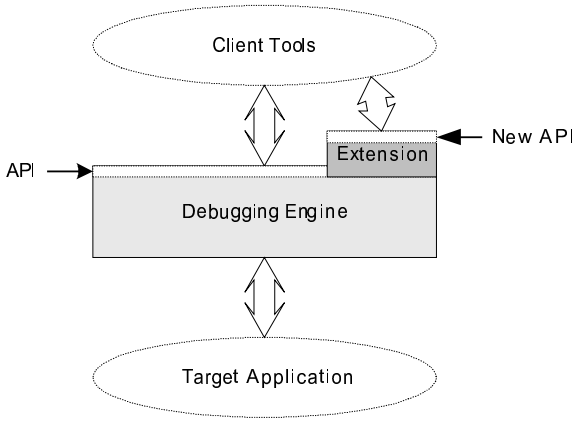

Figure 3.8: Internal extensibility of the debugging engine

Our experimental work on Fiddle_J (see Section 5.2.2), which implements an object-oriented API for the Java programming language, is a good example of an internal extension to the debugging engine.

3.6.2 External Extensibility

Although some new functionalities may be provided as internal extensions to the debugging engine, others require or recommend the client tool to have explicit knowledge or even to directly interact with some of the additional components of the new layer. These are called external extensions and are illustrated in Figure 3.9.

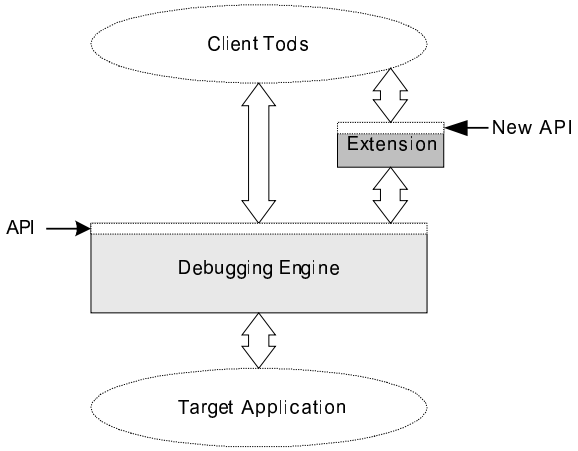

Figure 3.9: External extensibility of the debugging engine

Our experimental work on DEIPA$_2$ (reported in Section 5.5) is an example of such and external extension. DEIPA was developed as an intermediary between a testing tool and the debugging engine, aiming at supporting the testing and debugging sub-cycle of the software development process.

3.6.3 Cooperation and Integration Ability

The adequate combination of software development tools, sharing information and control data (see Figure 3.10) may contribute to make the software development process more effective.

For example, combining a *program visualizer* and a *distributed debugger* in order to cooperate by sharing data and control instructions may originate very interesting results. The former may present the software developer with an high-level view of the distributed program, focusing on processes interactions and state changes. The latter may provide the mechanisms for fine inspection and control of those processes, and to identify and locate their errors.

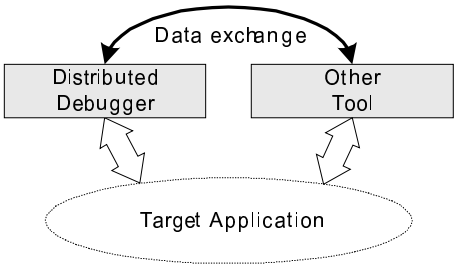

Figure 3.10: Cooperation ability of the debugging engine

Some efforts have already been made (although not completed yet) in providing a close cooperation between the debugging engine and the Pajé visualizer, developed at ID-IMAG, France. The benefits (and difficulties) of such cooperation are further discussed in Section 5.7.

Parallel software development environments aim at providing the software developer with a consistent and uniform set of tools. These tools should be able to cooperate and exchange data and control information. The proposed debugging engine may play a relevant goal in such integrated development environments, as illustrated in Figure 3.11.

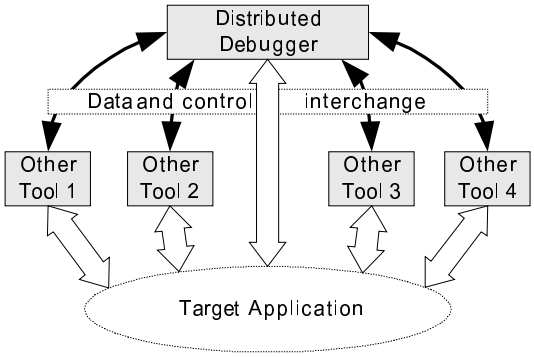

Figure 3.11: Integration ability of the debugging engine

Previous experimental work was done towards the integration of the debugging engine with graphical program editors, which were also used to provide animation of

the program execution to give debugging support to the developer. Such work on the integration of the debugging engine in GRADE and EDPEPPS is reported in Section 5.6.

3.7 Summary

Distributed debugging has to deal with an increased set of difficulties, when compared to sequential debugging, such as non-determinism, lack of global components (memory, clock, etc.), multiple execution flows, and variable communication delays.

The first and more natural approach to distributed debugging is to extend a sequential debugger to interact with more than one process, providing the software developer with a single debugging interface to access all the processes of the distributed program. Such an approach, however, needs to be complemented with services and functionalities which deal with the inherent characteristics of distributed programs and computations. In such a way, it is possible to provide the software developer with a complete debugging environment, which will help in the understanding of process interactions and, therefore, in localizing, isolating and correcting the errors in the distributed program.

In this Chapter we have presented a debugging engine for distributed programs. The services and functionalities proposed for the debugging engine were strongly suggested by the requirements posed by parallel software development environments which supported visual parallel programming languages and models.

This debugging engine was described in detail, by discussing its aims, software architecture, and how this software architecture allowed to fulfill those aims. One of the aims that deserved particular attention was extensibility, which motivated the layered software architecture of the debugging engine. The debugging engine provides a basic set of debugging functionalities and allows these functionalities to be complemented with others, which can be developed anew and integrated into the debugging environment.

4

The Fiddle Architecture and Implementation

Contents

This Chapter illustrates how the debugging engine described in the previous Chapter has been instantiated in two prototypes: the DDBG (Distributed DeBuGger) and Fiddle (Flexible Interface for Distributed Debugging: Library and Engine).

4.1 Introduction

The debugging engine proposed in the previous Chapter was the result of an evolutionary process which started with DDBG [CLD98, CLD01b, CLA99], our first design and implementation of a distributed debugger, followed by PDBG [LC98a, CLV+98], TDBG [LC98c] and, finally, Fiddle [LCM03, LC01, LC99].

Although being the first design/implementation of the debugging engine, DDBG already included many of its significant features. Much was learned with our experiences with DDBG, and a reevaluation of the debugging engine functionalities, specially concerning the basic services that should be provided and how they should be supported. This has driven to the design of DAMS and PDBG.

After our experiences with DDBG, the DAMS [CLV+98] (Distributed Application Monitoring System) research project was started. DAMS was much more ambitious than DDBG, aiming at becoming a distributed monitoring and control infrastructure, defining low level mechanisms which could be extended to support a set of services, such as distributed debugging. Any service using the DAMS infrastructure could access methods from other the services, easily exchanging data and control operations.

PDBG [CLV+98] (Process-level Debugger), a process-level distributed debugger that was defined as a DAMS service, aiming at validating the DAMS architecture and providing a distributed debugger would supersede the DDBG functionalities. TDBG [LC98c](Thread-level Debugger) design aimed at extending PDBG to support multi-threaded processes.

The first versions of DAMS and PDBG, although promising as research topics, had very unstable implementations, limiting considerably the exploitation of their potential as research topics. Research work on DAMS has been continued in [Dua04]. TDBG design relied on DAMS (like PDBG) as a basic infrastructure. The instability of the first DAMS prototype and their repercussions on the stability of PDBG allowed to infer that we would have similar repercussions on TDBG.

With Fiddle, we returned to a simpler software architecture and to a functionally much closer to the original DDBG, targeted exclusively to debugging.

This Chapter contains a brief discussion (description and evaluation) of DDBG and its software architecture, followed by a similar, although more detailed, discussion of Fiddle.

4.2 The DDBG Distributed Debugger

Being a predecessor of Fiddle, DDBG has, indubitably, some historical interest in the context of our research work. However, besides the historical interest, DDBG was a major step in our work, as most of the main features of the distributed debugging engine described in the Chapter 3 were already available in this tool.

4.2.1 The DDBG Architecture

The software architecture of DDBG has a distributed organization, consisting of multiple monitor/debugger instances which are scattered on the nodes of a distributed computing platform. Figure 4.1 illustrates such software architecture with three different kinds of processes involved: the target processes, the DDBG processes, and the user interface (client tool) processes.

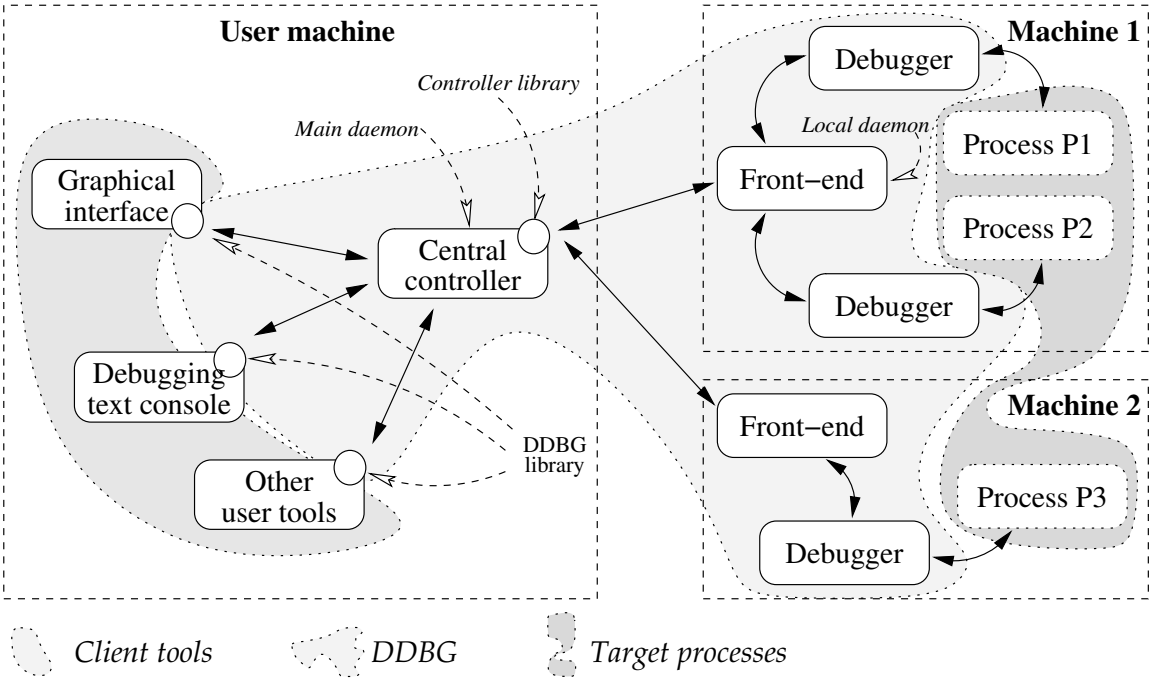

Figure 4.1: The DDBG software architecture

DDBG has a basic client-server architecture which follows the lines of the p2d2 design [Hoo96]. The client processes are depicted in Figure 4.1 as user debugging interfaces and other (debugging related) tools. These client processes are linked to the *DDBG Library*, which provides access to the central controller and to all DDBG debugging functionalities.

The target processes belong to the application being debugged. These application processes may be spread over multiple nodes, and nodes may have different hardware and/or operating systems. Heterogeneity concerns are handled by DDBG at the level of its internal communication layer. It uses PVM [GBD+98] for supporting the communication between the central controller and the local daemons and uses UNIX sockets for the interactions between each client tool and the central controller. Heterogeneity is also handled by allowing multiple possible types of local node debuggers to be integrated into the DDBG architecture.

The DDBG architecture internally consists of several component processes:

i) *Central Controller.* Coordinates the handling of the client requests, converting them into a set of commands and distributes them to the relevant local node debuggers.

It is also responsible for processing the local node debuggers' replies, and for sending them back to the client processes;

ii) *Local Front-ends.* There is one of these processes in each physical node with at least one target process. Besides some local interpretation of the debugging commands, it distributes them to the local debuggers and gets their answers back. The collected answers are then passed unprocessed to the central controller;

iii) *Local Node Debuggers.* A system-dependent sequential debugger, for a specific programming language and the underlying hardware. There is a local node debugger attached to each process of the target application processes, that applies the inspection and control commands to that process.

iv) *The Interface Library.* Any user tool can access the DDBG system as a client process that uses an interface library to interact with the central debugging controller. Through this interface library, client tools may control the DDBG system itself, the target processes, and even interact with the other client tools.

The debugging services provided by the central controller to operate upon the target processes are similar to those typically available in sequential debugging, such as breakpointing and single stepping. However, the central controller may access/control multiple target processes simultaneously. A detailed list of those services is provided in [CLA96b, CLA96a].

Detailed information about DDBG software architecture and its services may be found in [CLD01b].

4.2.2 Evaluation of DDBG

DDBG was used in a large set of experiments, where its design options have been tested and validated, some of them described in Chapter 5. From those experiments, we could conclude that:

i) The clear separation between the debugging engine and the user interface(s) provided the necessary freedom to allow the usage of DDBG in a wide (in scope and complexity) set of experiments;

ii) The library, as provided by DDBG, could be used by simple distributed debugging user interfaces as well as for supporting loosely-coupled interactions with third party tools;

iii) Integration of DDBG with other tools, which require much tightly-coupled relationships, required support for asynchronous event-based interactions. Such mechanisms were not included in the initial planning of the debugging engine;

iv) To use sequential debugging requirements as the basis to define a minimal set of services has proved to be an acceptable choice, as long as new (and more complex) services could be defined and integrated into the debugging engine;

v) Different clients have different requirements and needs concerning the debugging services to be supported by the debugging engine. To use a single layered software architecture has proved to be acceptable but also to lack the flexibility necessary to allow the extension of the debugging engine with new services;

vi) Experience has also shown that a stable implementation/prototype was a major requirement for further evaluation and validation of the debugging engine and for the development of its functionalities.

Based on the above analysis, we have redesigned the debugging engine software architecture. The result was described in Chapter 3 as the Fiddle debugging engine, whose internals of the software architecture will be analysed in the following.

4.3 The Fiddle Debugging Engine

The Fiddle debugging engine is, itself, a distributed program, and its multiple software layers are supported by a set of processes (daemons) and libraries which are distributed over the executing nodes.

The transfer of information between Fiddle layers implies the exchange of data between Fiddle components (libraries and daemons). Section 4.3.1 will discuss the Fiddle software architecture, with a detailed analysis of each component and some of the most relevant implementation issues. Section 4.3.2 focus in the discussion of how that data is exchanged between Fiddle components. This discussion includes the performance evaluation of two alternative communication protocols.

4.3.1 Fiddle Software Architecture

Functionally, each Fiddle layer supersedes the preceding one. Fiddle software architecture reflects this functional organization into layers, with each layer enclosing the software architecture of the preceding one. In the following, the software architecture is discussed in a bottom-up perspective, including a detailed explanation of each layer, its components and their interactions.

The Layer 0_s

In simple words, we can say that Layer 0_s, the lowest layer, provides a function-based interface to access a set of local node debuggers[1].

[1]The GNU GDB is the only node debugger supported in current Fiddle implementation.

The software architecture of this layer is presented in Figure 4.2, in the center, together with two target processes on the right side and a client tool on the left. In this example, the client tool (a debugging interface) is using Fiddle Layer0_s library to operate upon two local multi-threaded target processes.

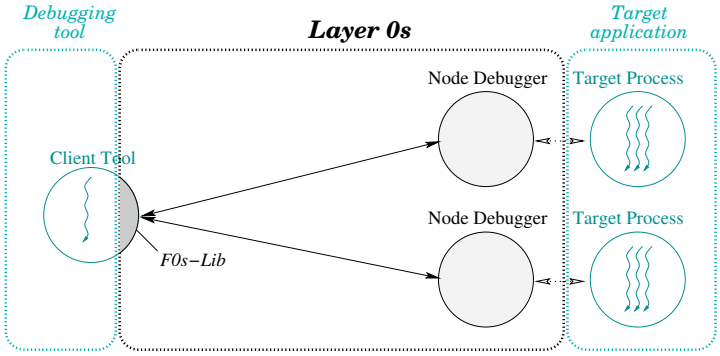

Figure 4.2: The Layer0_s software architecture

The Layer0_s Library (*F0s-Lib*). Using Layer0_s, it is possible to transparently launch a node debugger and attach it to a process already in execution, or to launch a new process under the control of a new node debugger. In both cases, the final result will always be identical: the target process will be stopped with a node debugger attached to it, and under the control of Fiddle.

Any bidirectional local communication channel can implement the links between the Fiddle Layer0_s library and the node debuggers. As an alternative to a bidirectional channel, two unidirectional channels may also be used. In the current implementation of Layer0_s, these communication links are alternatively supported by Unix pipes or Unix named pipes (*fifos*), selectable at compile time.

The internal data flow in the Layer0_s library follows the scheme described in Figure 4.3.

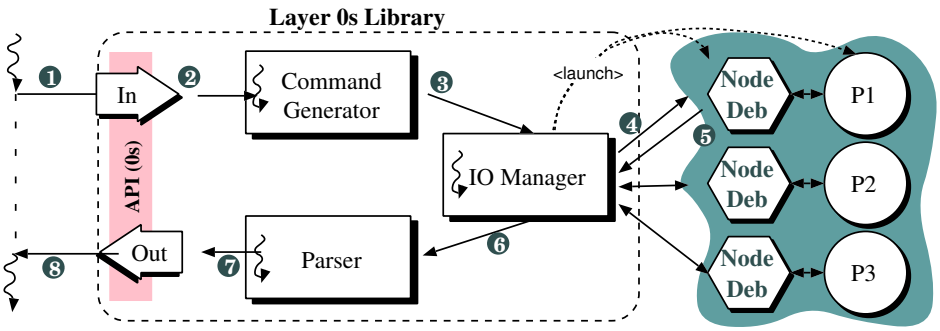

Figure 4.3: The Layer0_s internal data flow and processing

Service requests arrive in the input port in the form of a function call [1], whose arguments depend on the service requested. The service name and its arguments are passed to a command generator [2], which creates the appropriate command for the

node debugger being used to control the target process. This string is then forwarded to the IO manager [3], which is in charge of the communication with the node debuggers. The IO manager sends the command [4] to the appropriate node debugger and waiting for its reply [5]. It then forwards the received reply to the parser [6], which extracts all the relevant data in the reply into a `tkout` data structure. This structure is then returned to the caller of the service function [7] and [8].

It is assumed that there is only one execution flow (process/thread) interacting with Fiddle by using Layer0_s. If this is not the case, e.g., when multiple threads need to issue service requests simultaneously, then one of the upper layers must be used.

The Layer0_m

The Layer0_m is backwards compatible with Layer0_s, including the same set of services, with identical syntax and very similar semantics. The only significant difference is the additional support for simultaneous service requests from multiple programs flows (threads) in the single client tool. Figure 4.4 describes the software architecture of this layer (please note that, although very similar, Figures 4.4 and 4.2 are not identical).

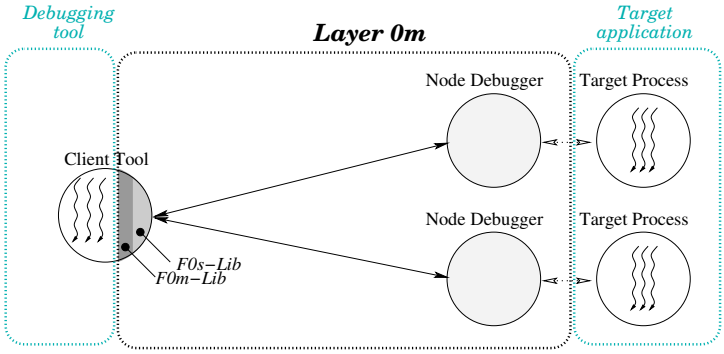

Figure 4.4: The Layer0_m software architecture

The Layer0_m Library (*F0m-Lib*). Relying upon Layer0_s, the Layer0_m is an internally multi-threaded library, which multiplexes the service requests from the multiple program flows of a Fiddle client tool to the multiple local target processes and vice-versa. Figure 4.5 on the next page describes the execution and data flows of the Layer0_m.

Layer0_m contains one input and one output queue for each target process being debugged, each holding the service requests to a specific target process and their replies respectively. An additional pair of input/output queues is reserved for general service requests which are not specific to any target process (yet), e.g., attaching Fiddle to a running process, or for services which deal with Fiddle internal status, e.g., list of processes currently under debugging with their names, and their real and symbolic process IDs.

As in Layer0_s, the client tool service requests arrive in the input port in the form of a function call [1], whose arguments depend on the service being requested. The arguments are then packed into a structure (the `tkin` structure) which is enqueued in

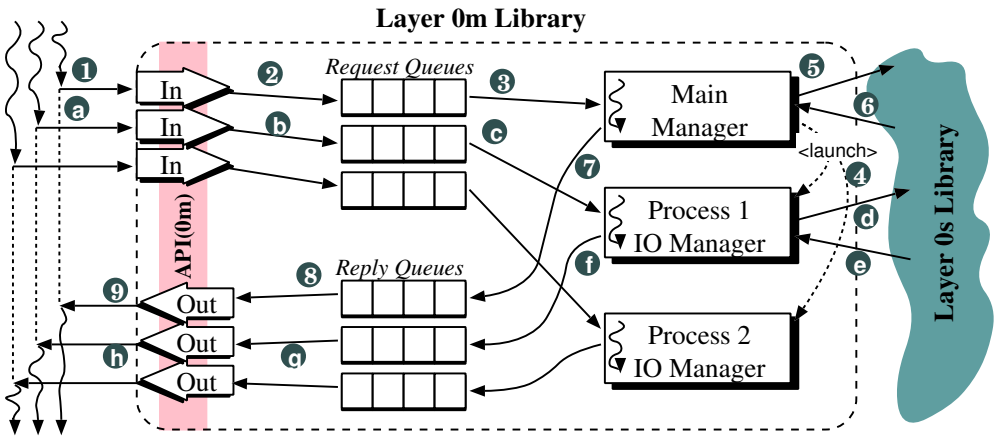

Figure 4.5: The Layer 0_m internal data flow and processing

the appropriate request queue [2] and the calling thread will block waiting for the reply to be available in the reply queue [8].

If the service requested by the client tool is not specific to any target process, it is directed to the *main manager* [3]. Some of the services processed by this component will add a new target process into Fiddle and, in these cases, a new *IO manager* thread is created [4], that will manage all future service requests directed to that specific target process. The *main manager* will then ask the lower layer (Layer 0_s) to process the service [5] and [6] and, once completed, the reply is enqueued into the output queue [7]. The original thread will then be unblocked and will receive the reply for the requested service [8] and [9].

However, the vast majority of the client tool service requests are directed to a specific processes [a] and, in this case, the request will be enqueued to the process specific request queue [b]. The *IO manager* thread, which was blocked waiting for requests to arrive on its queue, is resumed [c] and requires the lower layer (Layer 0_s) to process the service [d] and [e]. The received reply is enqueued in the process reply queue [f] and the *IO Manager* thread blocks again waiting for new service requests in its queue. The calling thread will then be resumed and will receive the reply for the requested service [g] and [h].

By holding requests (and replies) in thread-safe queues, it is possible to ensure the internal queues consistency even when multiple client tool threads try to enqueue service requests simultaneously. As there is a dedicated *IO Manager* for each target process, we also ensure that, at any time, at most one service request is being applied to a target process.

The Layer 1_m

The Layer 1_m is functionally backwards compatible with the previous layers (Layer 0_m and Layer 0_s), but with minor syntax and semantic changes. The semantic differences reflect the ability to debug remote processes, and the syntactic differences are limited to

the addition of one extra argument (with the node ID) to some of the services, namely those which refer to target processes that do not have (yet) a process symbolic identifier, e.g., attach Fiddle to a running process.

Figure 4.6 describes the software architecture of this layer.

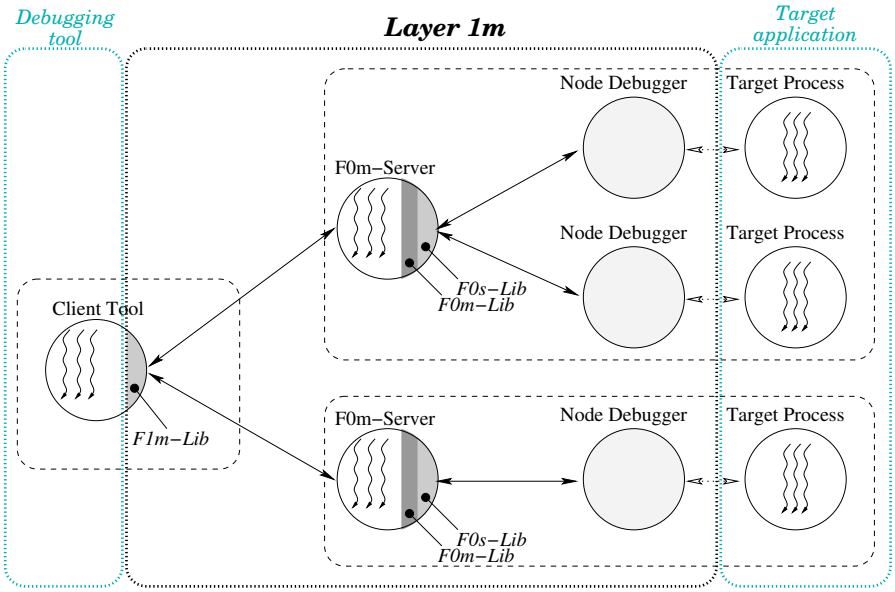

Figure 4.6: The Layer 1_m software architecture

The Server 0_m (*F0m-Server*). The Server 0_m is a small application that behaves as a Layer 0_m client, multiplexing its input port into as many output ports as target processes running on the local node. This means it forwards all service requests arriving in the local node to the appropriate target process, and sends back the associated replies. Figure 4.7 describes the execution and data flows in Server 0_m.

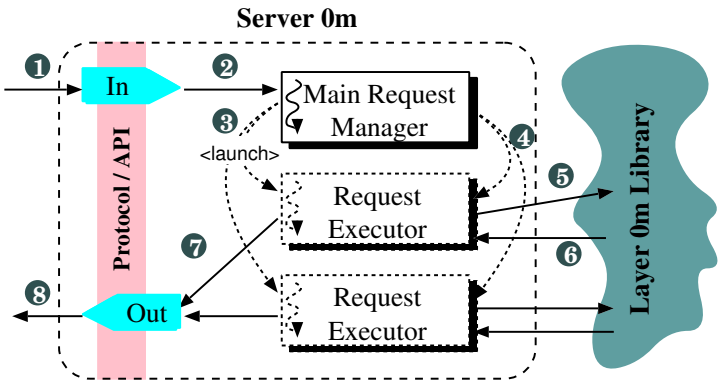

Figure 4.7: The Server 0_m internal data flow and processing

In Server 0_m, the client tool service requests arrive to its input port (a TCP/IP socket) as a serialized `tkin` [1]. This record is converted from the serialized representation into its internal (binary) format and given to the *main request manager* thread [2]. For each service request received by the *main request manager*, a new *request executor* thread

will be launched [3]. The *request executor* will receive the `tkin` record [4] and process the request [5] by directing it to the underlying layer (Layer 0_m). The reply received from Layer 0_m [6] is sent to the output port [7] and [8] and the *request executor* thread terminated.

The Layer 1_m Library (*F1m-Lib*). This library directs the service requests coming from the client tool to the appropriate Server 0_m (the one located in the same node as the processed targeted by the service). If the client tool is a multi-threaded program, this library is also able to deal with concurrent service requests being issued by different threads in the client tool. The replies received from the servers (Server 0_m) are also forwarded back to the waiting thread in the client tool.

Figure 4.8 describes the execution and data flows of the Layer 1_m library, and its relation with Server 0_m.

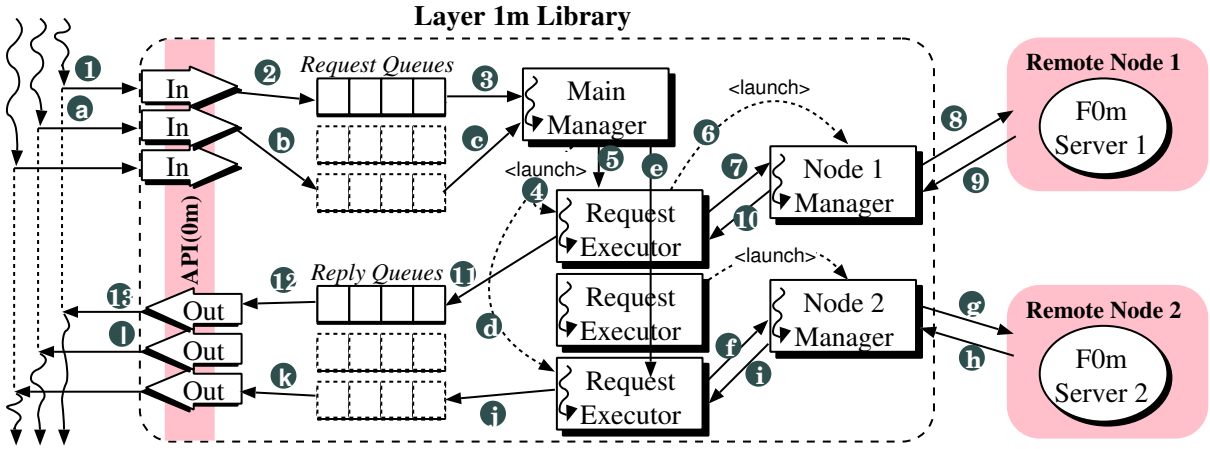

Figure 4.8: The Layer 1_m internal data flow and processing

Layer 1_m library contains a set of input and output queues, one for each Server 0_m. The input queues hold the service requests and the output queues hold the replies.

Just like in Layer 0_m, there is also an additional pair of input/output queues for general service requests which deal with Fiddle internal status, e.g., list of processes currently under control of Fiddle, and for service requests which deal with processes that do not have yet a Fiddle process ID, e.g., attaching Fiddle to a running process.

The service request will be processed by either the *main manager* or one of the *node managers*. The main manager will process the requests which do not have a specific target node (yet), e.g., obtain the list of all symbolic process identifiers, or access to processes in a node where a Server 0_m is not running yet. All the other services are processed by the node manager associated to the appropriate node.

Service requests arrive to the input port in the form of a function call [1]. The arguments of such function call are packed into a structure (the `tkin` structure) which is enqueued in the appropriate queue [2] and the calling thread is blocked waiting for the reply [13].

All queues are being monitored by the *main manager*. Once a request is found [3], a new *request executor* thread is created [4]. The request executor will live while the service is being processes, and will die afterwards. The request data (a `tkin` structure) is sent to the request executor, which will launch a *node manager* thread [6] if needed, and pass it the request data [7]. This request data will then be converted to an external data representation (XDR) format and sent to the Server 0_m [8] in the appropriate node.

Once a reply coming from the Server 0_m is received in the node manager [9], it is forwarded to the request executor [10] which will enqueue it in the right output queue [11] and will die afterwards. The reply will be extracted by the waiting thread [12] who made the service request [13].

If the service has a well defined target [a] and [b], it is also processed by the main manager [c], which will launch a new request executor [d], and pass it the request arguments [e]. The request executor will forward the data to the (already existing) node manager [f], which, as before, will convert it to an external data representation format and sent it to the appropriate node manager [g]. As before, once a reply is received by the node manager [h], it is forwarded to the request executor [i], which will enqueue it on the appropriate reply queue [j] and die afterwards. The reply will be extracted by the waiting thread [k] who made the service request [l].

It is assumed that the underlying layer (Layer 0_m in this case) is able to cope with multiple concurrently services requests targeting the same or different process. By creating a new *request executor* thread for each request that arrives in the queue, all service requests are processed concurrently.

The Layer 2_m

The Layer 2_m extends Layer 1_m to support multiple tools connecting simultaneously to Fiddle and concurrently issue services requests to the same target program, as shown in Figure 4.9 on the next page, which describes the software architecture for this layer.

The Server 1_m (*F1m-Server*). Layer 1_m accepts a single client multi-threaded client, but such functionality may not be enough for some applications. In Section 5.4.1 we can find an example were we need multiple client tools connected to the same target process). Server 1_m is a small application that multiplexes the client tools requests to Layer 1_m and demultiplexes the replies back to the client tools.

Server 1_m initializes itself by initiating a connection to Layer 1_m and then waits forever for connection requests from client tools. Once a client tools connects to this server, it can issue debugging service requests to Fiddle. While one client tool is accessing Fiddle, other client tools may also connect and operate concurrently with the existing ones, all of them making service requests to Fiddle and targeting the distributed program.

Figure 4.10 on the following page shows execution and data flows in Server 1_m.

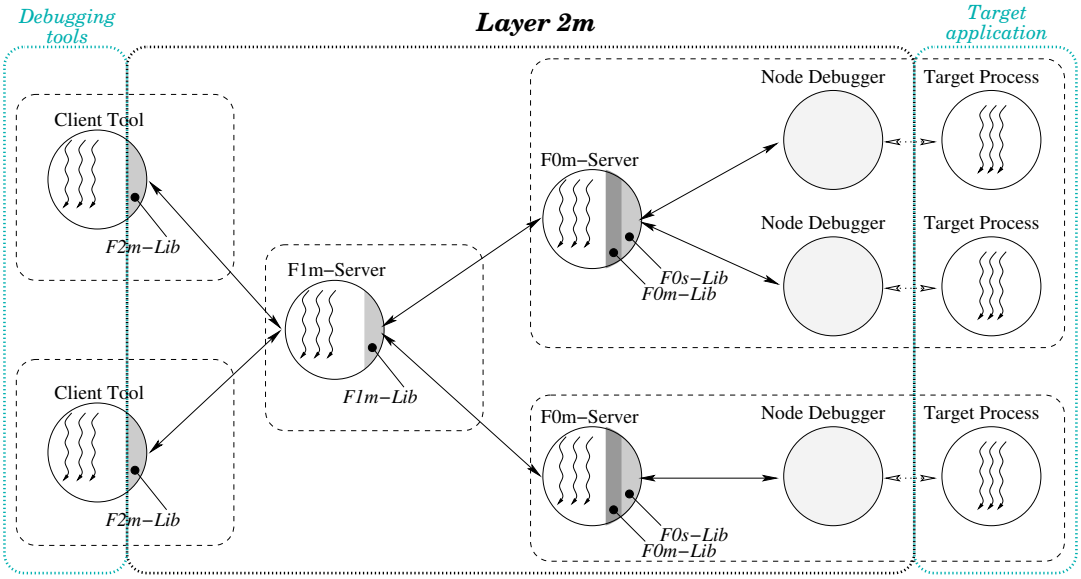

Figure 4.9: The Layer 2_m software architecture

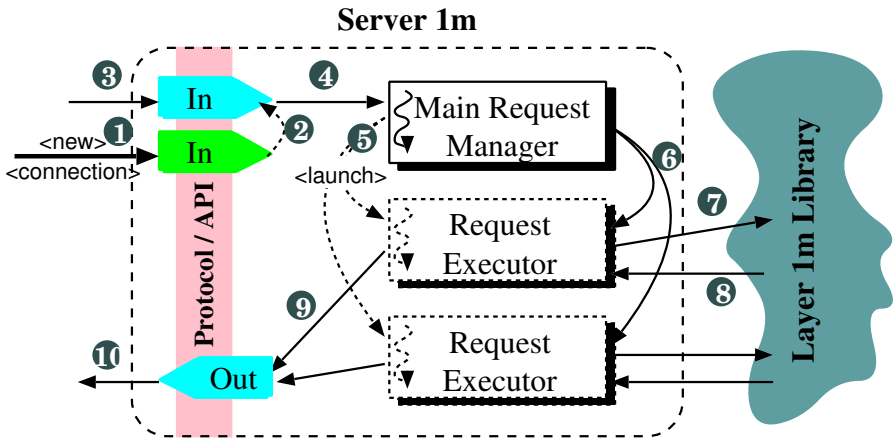

Figure 4.10: The Server 1_m internal data flow and processing

Processes register themselves as Fiddle clients by establishing a TCP/IP connection to a well defined (although configurable) port of a well defined (also configurable) host where Server 1_m is running.

In Server 1_m, connection requests arrive to its *connection port* [1] (a TCP/IP socket which is automatically replicated by the underlying OS), originating a new exclusive connection port [2]. Through each dedicated connection, Server 1_m handles service requests from a specific client, by receiving the service data in a `tkin` structure, coded in a network independent representation [3]. The received data is converted to the structure internal (binary) format and given to the *client manager* thread [4]. For each service request received, the client manager will launch a new *request executor* thread [5], which will receive the `tkin` structure [6] and will process the service requested by calling the services in Layer 1_m [7]. The reply received [8] is written into the process exclusive output port [9] and [10], and the *request executor* thread terminated.

The Layer 2_m Library (*F2m-Lib*). Any program linked to this library may act as a Fiddle client tool, issuing debugging service requests to the Fiddle debugging engine. This access to Fiddle services is not exclusive, and any other program linked to this same library may also act concurrently as a Fiddle client tool.

The Layer 2_m library operation is limited to establish a connection with the unique (to Fiddle) Server 1_m server, redirect any service request from the client tool to the server and wait for the reply from the server. It is assumed that the execution flow which issued the service request will remain blocked waiting for the reply. If such execution flow is a thread, then just this thread in the client tool will be blocked.

Figure 4.11 describes the execution and data flows of the Layer 2_m Library.

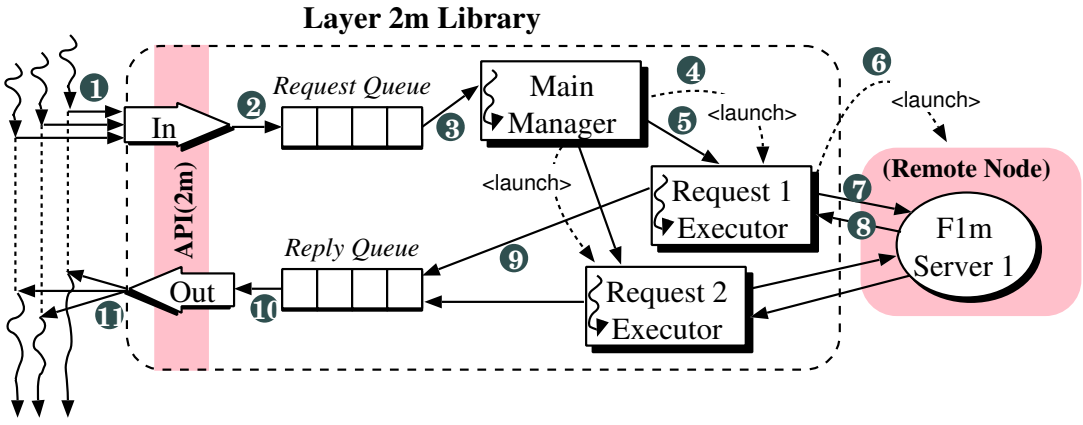

Figure 4.11: The Layer 2_m internal data flow and processing

Layer 2_m Library contains a single input queue and another single output queue, both common to all client tool threads. Any thread aiming at accessing a debugging service from Fiddle will enqueue its request in the unique input queue and will receive a *request ID*. Then, the client tool thread will block waiting for the reply associated to that specific *request ID* to arrive in the output queue.

The service requests arrive in the input port in the form of a function call [1]. The function call arguments are then stored in a structure (the `tkin` structure) which is enqueued [2] and the calling thread blocked in the output queue [11], waiting for the reply. Replies are sent back to the calling thread as soon as they become available, independently of the order in which the services were requested. This may cause a reply to a second service request, which was available before, to be returned before the reply to the first service request, which was available only after. The calling thread will remain blocked until the reply to his service request becomes available.

Meanwhile, the service request will be processed by the *main manager* thread [3]. Each time a new service is requested, the main manager launches a new *request executor* thread [4] that receives the service data [5] and will be in charge of its execution. If the Server 1_m is not running, it may be transparently launched by the Layer 2_m [6]. Once the Server 1_m is running, the *request executor* sends it the request data [7] and blocks waiting for the reply [8].

Once the reply is available, it is enqueued in the reply queue [9], from where it will be removed by the calling thread [10] and [11].

Creating a *request executor* for each new request arriving to the queue, ensures that all service requests are processed concurrently. It is up to the lower layers (Layer 1_m in this case) to take care of concurrent requests directed to the same target processes.

Although not represented in Figure 4.11 on the preceding page, the Server 1_m accept simultaneous connections from multiple Layer 2_m libraries, which also means from multiple client tools.

The Layer 3_m

Layer 3_m implementation is too immature, so we will refrain ourselves from speculatively discussing the implementation issues of this level.

4.3.2 Internal Communication in Fiddle

In the layers where Fiddle provides access to remote processes, i.e., from Layer 1_m upwards, service requests and their replies must be transmitted over the network between Fiddle components (i.e., between the library and the daemons). The service requests are originated in the client tools by calling a library function. Inside this library function, the identifier of the requested service and its arguments are packed into an *input token* (a `tkin` record, described below) and sent to Fiddle core. The reply to such service request is also packed into an *output token* (a `tkout` record, also describe below) and sent back to the client tool.

In the following we will discuss the internal (binary) formats for such input and output tokens, how their transmission over the network is operated, and also evaluate the performance of such transmission.

The input and output tokens

The input token (`tkin` record) aggregates all the necessary information to represent a debugging service request, namely the service identifier and its arguments. Such token is packed in the library linked to the client tool and sent to the Fiddle core. Figure 4.12 shows a graphical representation of the input token.

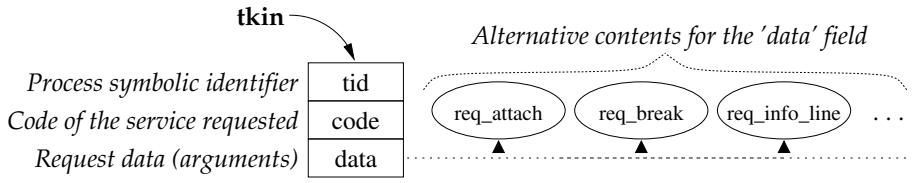

Figure 4.12: The `tkin` structure

Being represented by a quite simple structure, the transfer of input tokens over the network does not raise major difficulties, and the ones raised are a subset of those raised by the transfer of the output token, which is discussed below.

The output token (`tkout` record) aggregates all the information needed for replying to a service request. Considering that the service request may fail at library, debugging engine, or node debugger levels, and may also succeed with a considerable number or different replies, the output token structure will, necessarily, be quite more complex than its input counterpart.

A graphical representation of the `tkout` structure is presented in Figure 4.13 on the following page.

We will concentrate our study and discussion in the output token, as its complexity and the problems raised by its transfer over the network are a superset of the ones raised by the input token.

Fiddle external data representation

As shown in Figure 4.13 on the next page, the output token contains sub-structures which may contain dynamic lists and lists of lists. So, the `tkout` structure which represents the token is not contiguous in memory, and its transfer over the network demands a *serializing* service on the sender. This converts this complex structure into a plain stream of bytes, and an *assembling* service on the receiver converts the byte stream into a `tkout` structure again.

In an homogeneous environment, the serializing service could generate a binary stream, but this would not be acceptable in an heterogeneous environment where, most probably, the binary representation of the serialized token would be different in distinct nodes. In this sense, heterogeneity adds another degree of complexity, as data must be converted from internal (binary) representation to some form of architecture independent *external data representation* (XDR) prior to the emission, and converted back to the internal (binary) representation upon reception.

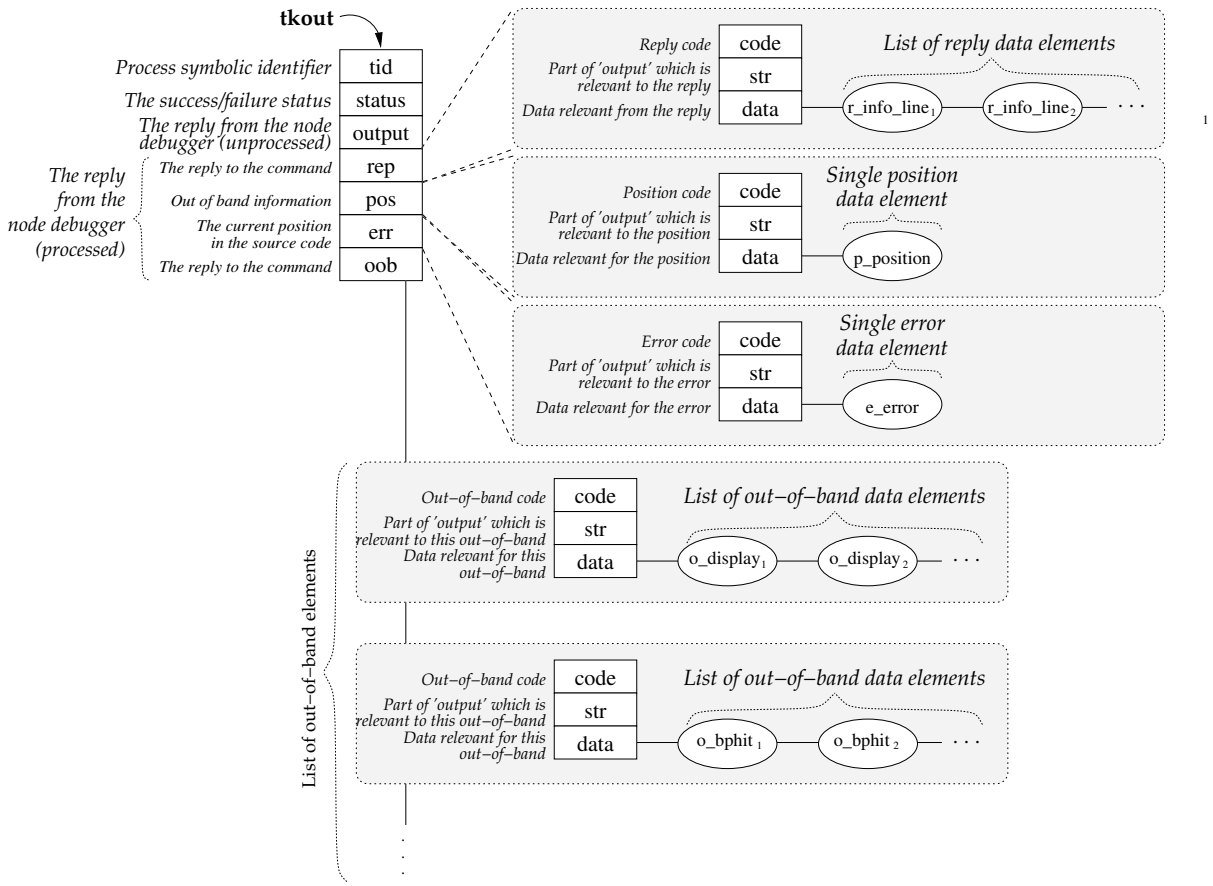

Figure 4.13: The `tkout` structure

Existing message passing systems, such as PVM [GBD+98] and MPI [For94], could provide both a protocol independent communication layer and a XDR data format for transferring data in heterogeneous environments. Nevertheless, these systems would just look too complex for such a simple job. Therefore, and assuming that Fiddle internal communication performance was not a main issue, the choice for the XDR data format has fallen to a text based, human readable, format. Such format also has the advantage of easiness of debugging.

Eight (four plus four) basic functions were developed to convert `tkin` and `tkout` structures from binary to text and vice-versa.

i) `tkin_2_jml()`, `jml_2_tkin()`, `tkout_2_jml()` and `jml_2_tkout()`; and

ii) `tkin_2_xml()`, `xml_2_tkin()`, `tkout_2_xml()` and `xml_2_tkout()`.

The first set of these serializing functions uses a proprietary (non-standard) text-based encoding for those structures, in a format named 'JML'[2]. The second set uses an alternative encoding in XML [W3C98], using a parser based in the SAX [Meg00] technology. Both encoding formats were studied in detail, their performance analyzed and reported succinctly here. A more detailed analysis may be found in [Mor02].

[2]The name 'JML' (*João's Markup Language*), was suggested by the student who developed the alternative codification based in the 'XML' standard.

Current Fiddle implementation supports both encoding formats, and the choice of the format to use can be done at Fiddle startup time, by using command line arguments or by setting an environment variable.

The JML format When developing the JML format for external data representation of Fiddle basic structures `tkin` and `tkout`, attention was focused in human readability and easiness of debugging rather than performance. In this sense, an indented text based format was defined.

Figure 4.14(a) on the following page shows an example of an input token coded in JML format, which represents an "*info_break*" service request. This service requests Fiddle to report on all current breakpoints set in all target processes.

Figure 4.15(a) on the next page shows an output token containing a possible reply to the above service request, also encoded in the JML format. The reply contains a list with details on the current active breakpoints in the target application.

The XML format Being in a proprietary format, the JML codification raised serious difficulties to external agents (human or software) who were interested in the transmitted data. This would require a reverse-engineering process to decode the communication protocol and data format for all possible service requests and replies. As JML is a proprietary data format, any slight change in its specification would require more reverse-engineering work and consequent changes in the agents to support the new version.

To minimize these difficulties, an alternative codification to the `tkin` and `tkout` structures was developed using the XML [W3C98] standard and implemented using the SAX technology. In addition, the usage of a standard for information exchange between Fiddle components also facilitates the interaction with other tools which, for some reason, are not able to use Fiddle libraries. Almost every programming language nowadays has support for generation and processing of XML files.

Figure 4.14(b) on the following page shows an example of an input token codified in the XML format. This token represents exactly the same service request as the one on Figure 4.14(a) on the next page, although in a different format.

Figure 4.15(b) on the following page shows a possible reply to the service request described by this `tkin`, by presenting a `tkout` structure also encoded in the XML format. This token represents exactly the same reply as the one on Figure 4.15(a) on the next page.

By comparing Figures 4.14 and 4.15, it is easy to find the similarities (and the differences) between the JML and XML data representation formats. Both formats are syntactically very similar and semantically equivalent.

```
tkin = {
  tid = 1
  code = 31
  info_break = {
  }
}
```

(a) JML format

```
<?xml version="1.0" standalone="yes"?>
<tkin>
  <tid>1</tid>
  <code>31</code>
  <data/>
</tkin>
```

(b) XML format

Figure 4.14: The `tkin` structure

```
tkout = {
  tid = 1
  str = 193 Num Type           Disp Enb \
Address    What
1   breakpoint     keep y   0x080484aa \
in main at /home/jml/Infinito.c:38
2   breakpoint     keep y   0x08048414 \
in sfake at /home/jml/Infinito.c:16

  status = 0
  rep = {
    str = 149 1   breakpoint     keep y   \
0x080484aa in main at /home/jml/Infinito.c:38
2   breakpoint     keep y   0x08048414 \
in sfake at /home/jml/Infinito.c:16

    code = 66
    chain[2]:data = {
      info_break = {
        bpid = 1
        type = 10 breakpoint
        disp = 4 keep
        enabled = 1
        address = 0x80484aa
        function = 4 main
        file = 20 /home/jml/Infinito.c
        line = 38
        hit_counter = -1
        ign_counter = -1
        condition = 0
      }
      info_break = {
        bpid = 2
        type = 10 breakpoint
        disp = 4 keep
        enabled = 1
        address = 0x8048414
        function = 5 sfake
        file = 20 /home/jml/Infinito.c
        line = 16
        hit_counter = -1
        ign_counter = -1
        condition = 0
      }
    }
  }
  chain[0]:oob = {
  }
  pos = {
  }
  err = {
  }
}
```

(a) JML format

```
<?xml version="1.0" standalone="yes"?>
<tkout>
  <tid>1</tid>
  <status/>
  <str>Num Type           Disp Enb \
Address    What
1   breakpoint     keep y   0x080484aa \
in main at /home/jml/Infinito.c:38
2   breakpoint     keep y   0x08048414 \
in sfake at /home/jml/Infinito.c:16
</str>
  <rep>
    <code>66</code>
    <str>1   breakpoint     keep y   \
0x080484aa in main at /home/jml/Infinito.c:38
2   breakpoint     keep y   0x08048414 \
in sfake at /home/jml/Infinito.c:16
</str>
    <data>
      <node>
        <r_info_break>
          <bpid>1</bpid>
          <type>breakpoint</type>
          <disp>keep</disp>
          <enabled>1</enabled>
          <address>134513834</address>
          <function>main</function>
          <file>/home/jml/Infinito.c</file>
          <line>38</line>
          <ign_counter>-1</ign_counter>
          <condition/>
        </r_info_break>
      </node>
      <node>
        <r_info_break>
          <bpid>2</bpid>
          <type>breakpoint</type>
          <disp>keep</disp>
          <enabled>1</enabled>
          <address>134513684</address>
          <function>sfake</function>
          <file>/home/jml/Infinito.c</file>
          <line>16</line>
          <ign_counter>-1</ign_counter>
          <condition/>
        </r_info_break>
      </node>
    </data>
  </rep>
  <oob/>
  <pos/>
  <err/>
</tkout>
```

(b) XML format

Figure 4.15: The `tkout` structure

Performance evaluation of Fiddle internal communication

In [Mor02] there is a complete study on the performance of the serialization of the output token into plain text using the JML and XML formats, and the reverse operation into binary. The main goals of the performance evaluation were to evaluate the penalty of using a text-based data representation for the serialized `tkout`, and the impacts of using a standard versus proprietary formats.

In this sense, we evaluated the time spent in the serialization of output tokens to both (JML and XML) formats and their conversion back into binary. Experimentation has shown significant performance variations depending on the contents of the token: if it contained long lists of data (e.g., the reply to a source code listing request) or not (e.g., setting a breakpoint), and this was also subject to detailed evaluation. Finally, we also aimed at checking whether performance behavior was linear on the size of the token.

Table 4.1 shows the items subject to performance evaluation.

Parameter	Options
Operation	i) Serialization of `tkout` ii) Codification of serialization `tkout` into binary
Format after serialization	i) JML ii) XML
`tkout` contents	i) With long lists ii) Without lists
Serialized `tkout` size	256, 512, 1024, 2048, 4096 and 8192 bytes

Table 4.1: Items subject to performance evaluation

Figure 4.16 on the next page shows the time taken to serialize two variants of the `tkout` structure, with and without long lists of data, to formats JML and XML. From the analysis of this Figure it is possible to conclude that the operation of serializing the `tkout` structure into XML outperforms the serialization into JML. Both serialization functions were "hand coded" and part of the source code has been shared, but the serialization to JML was done before, while the serialization to XML was done later and some efforts were made to optimize its performance. This explains some of the performance differences between in these functions.

Figure 4.17 on the following page shows the time necessary to convert a `tkout` from its serialized form (JML or XML) to binary. From the analysis of this graphic it is easy to conclude that the generation of a binary `tkout` from the XML representation is considerably slower than from the JML representation.

As there was a significant difference in the time needed to convert serialized `tkout`'s in JML and XML formats to their binary representation (the latter was much slower), some more experiments were made to determine its origin. These experiments consid-

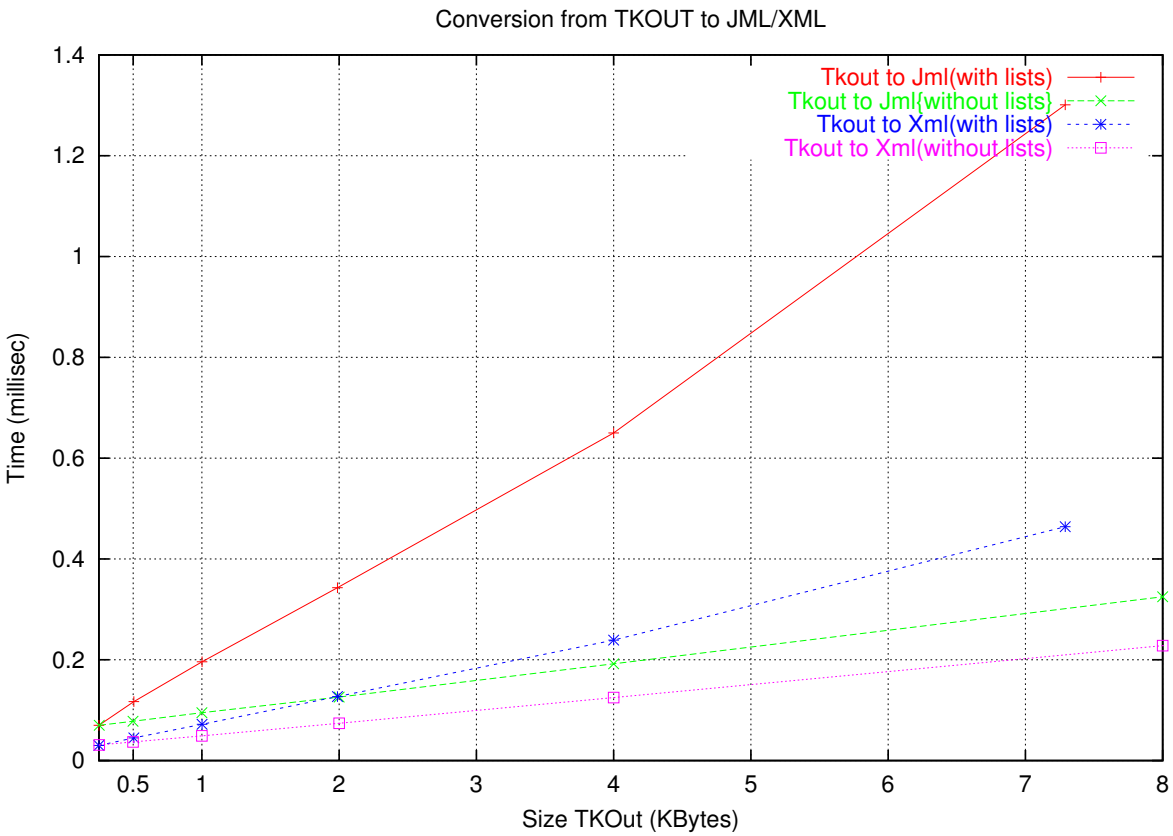

Figure 4.16: Serialization of `tkout` into JML and XML formats

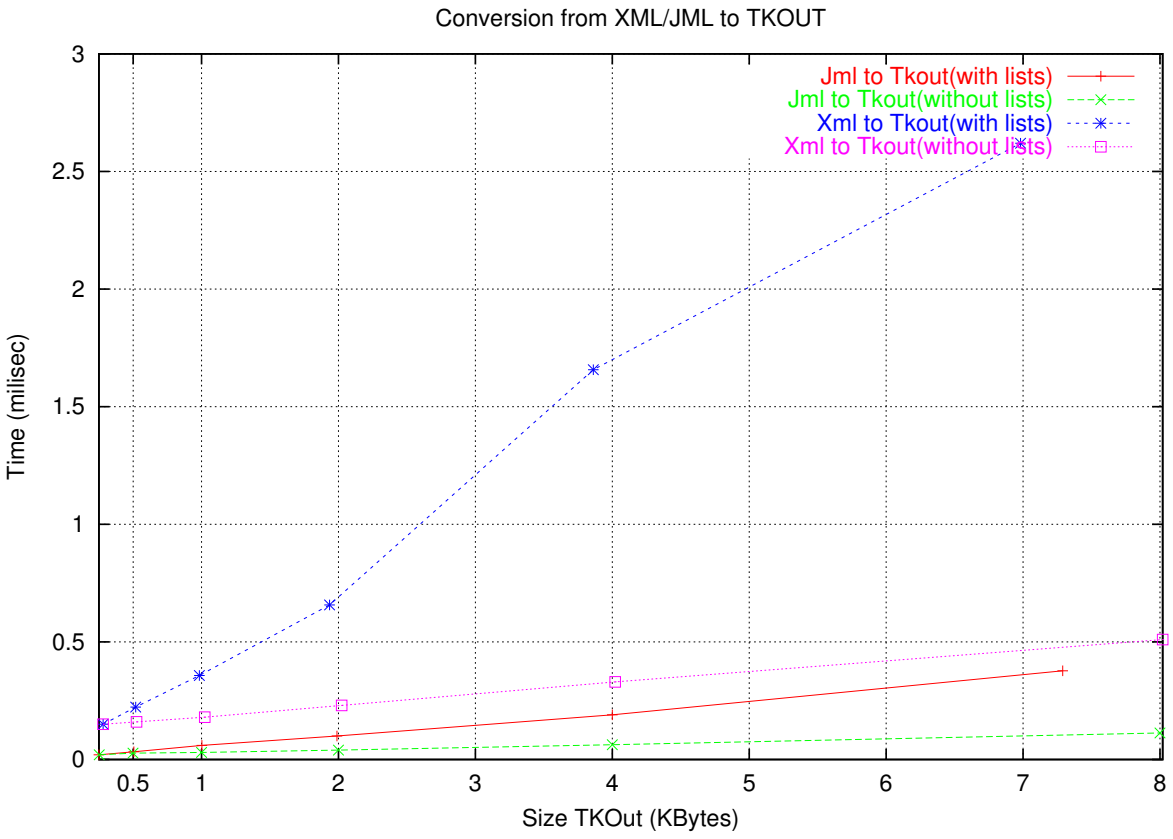

Figure 4.17: Codification from JML and XML formats into binary `tkout`

ered three different kinds of SAX based parsers: an empty parser (without processing the tags and data fields), a "database" parser (which processes the tags and data fields and builds an internal database with such information, but this database has no further processing and the corresponding `tkout` structure is not generated), and a full parser (fully processing the tags and data fields and generating the corresponding binary `tkout`). The results of such experiments are shown in Figure 4.18.

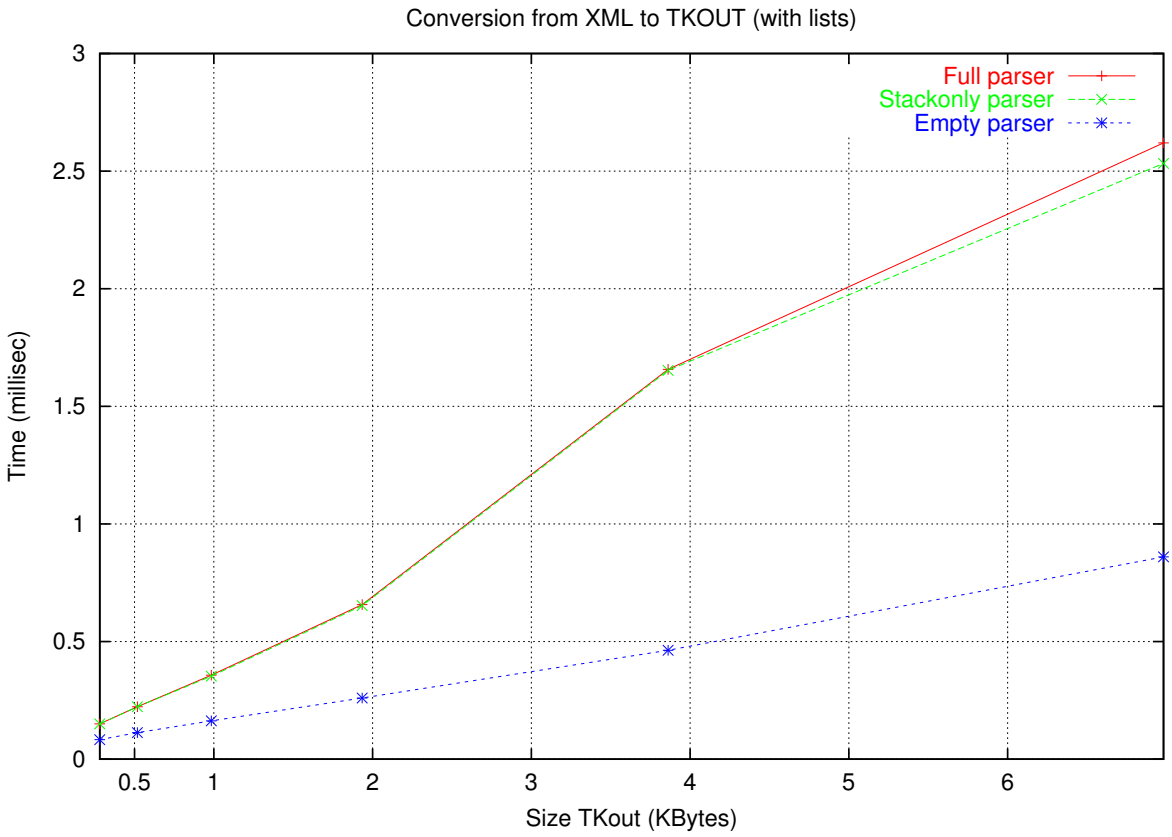

Figure 4.18: Processing times for XML file

By analysing this Figure one can infer that the SAX library for processing XML files is very time consuming (nevertheless, the library we used was the most efficient XML processing library available at that time). Fully converting a serialized token from JML to binary is quicker than just processing the serialized token in XML format with an empty parser. When we add some processing to this parser, the time needed to process the file increases considerably. This study allowed us to conclude that by using XML we benefit from the usage of a popular standard, but are also considerably penalized in performance.

Facing these results, we have decided to keep support for both codification protocols in Fiddle. The user may choose which protocol should be used, at Fiddle startup time.

4.4 Summary

The discussed debugging engine results from an initial proposal, which led to the development of DDBG. Some of the most relevant of the current features of the debugging engine were already defined in its initial version, and available in the DDBG prototype. DDBG was used intensively in a number of research projects, which allowed us to evaluate both the debugging engine design and the prototype. This evaluation led to further developments of the debugging engine and of new prototypes, in PDBG and TDBG. With successive refinements, the debugging engine evolved to its current status, as discussed in Chapter 3, and its implementation to the current prototype (Fiddle), as discussed in this Chapter.

The current implementation of the debugging engine, Fiddle, was also used in a number of research projects and its features and its performance and functionalities evaluated and compared, whenever possible, with its initial implementation in DDBG.

DDBG had many limitations, in both the debugging engine design and in its implementation, such as having all the debugging engine management concentrated in the main daemon, the inability to deal with different node debuggers and with long-time-to-complete services. Also, clients had to be single-threaded programs and the API used necessarily a synchronous calling model. Also, initially the debugging engine had been defined with a flat architecture, making it hard to be extended with new functionalities or to be adapted to the intricacies of new execution environments.

In the current stage, the debugging engine overcame the above limitations by making use of a layered architecture, by supporting multi-threaded client tools and by defining an API with an event-based callback model (although this last functionality is not implemented yet).

The latest specification of the debugging engine, based in a layered architecture, is implemented in Fiddle. In its current implementation, Fiddle contains almost 20.000 lines of source code, of which over 17.700 lines are C source code, which implements all the debugging engine layers from Layer0_s to Layer2_m, and a bit over 1.700 are Java, which implements a Java native interface to Fiddle libraries. Figure 4.19 on the facing page shows some software metrics for Fiddle, generated using David A. Wheeler's 'SLOCCount'[3].

Fiddle is, itself, a distributed program, whose multiple components communicate using a text-oriented, human readable, protocol over TCP-IP channels. The initial version of such protocol was proprietary. A second version based in the XML standard was also implemented. The performance of both implementations was studied and compared, which allowed to conclude that the initial proprietary protocol was, in average, faster than the second version.

[3] http://www.dwheeler.com/sloccount/

```
SLOC    Directory       SLOC-by-Language (Sorted)
5565    f0m             ansic=5565
4004    f0s             ansic=4004
3669    fiddle_j        ansic=1812,java=1724,awk=123,sh=10
2466    f1m             ansic=2466
2156    proto-console   ansic=2156
979     util            ansic=979
648     f2m             ansic=648
89      examples        ansic=89


Totals grouped by language (dominant language first):
ansic:         17719 (90.51%)
java:           1724 (8.81%)
awk:             123 (0.63%)
sh:               10 (0.05%)


Total Physical Source Lines of Code (SLOC)                = 19,576
Development Effort Estimate, Person-Years (Person-Months) = 4.54 (54.52)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.95 (11.42)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 4.77
Total Estimated Cost to Develop                           = $ 613,694
 (average salary = $56,286/year, overhead = 2.40).
```

Figure 4.19: Sofwtare metrics for Fiddle

[This page was intentionally left blank]

5

Validation of the Debugging Engine

Contents

This Chapter Presents a set of case studies, where one of the debugging engine implementations (DDBG or Fiddle) have been used, and how they contributed to the operational and functional validations of the debugging engine and its implementations.

5.1 Introduction

The assessment of a software package should include two distinct dimensions, the *operational validation*, which aims at ensuring the program operates correctly and according to its specification, and the *functional validation*, which aims at verifying if an operationally correct software package fulfills all the initially identified requirements. The assessment of the debugging engine proposed in Chapter 3, whose implementations are described in Chapter 4, should also be verified according to those dimensions.

The debugging engine instances described in the previous Chapter (namely DDBG and Fiddle) have been used in a number of research projects, which resulted in the development of a considerable set of client tools. Some of these tools were developed specifically to be client tools of the debugging engine, while others were simply adapted to interact with the debugging engine. This multitude of experiments played a very important role in the operational and functional validation of the debugging engine, as it widened the scope of the functional and operational requirements put upon the debugging engine. A significant aspect is that the debugging engine was actually used and evaluated by external users, both senior researchers and students.

The functional validation of the debugging engine was supported mainly by the experiments involving DDBG, and were also confirmed by some of the experiments involving Fiddle. The operational validation of the debugging engine depends on each specific implementation, and was constantly assessed by the multiple client tools in both implementations. One of the implicit goals of the Fiddle reimplementation of the debugging engine was to obtain a stable and operationally reliable prototype.

Depending on the time window, some of the research projects which helped verifying the functional and operational consistency of the debugging engine used DDBG while some others used Fiddle. Because Fiddle clearly supersedes DDBG, both functionally and operationally, we believe that the experiments which took place with DDBG would also be feasible with Fiddle as the debugging engine. Our experience with the DEIPA [LCK+97] tool (see Section 5.5), which was initially developed using DDBG and later re-engineered (and reprogrammed) in DEIPA$_2$ [Mor02, LCM03] using Fiddle, is an excellent example of such feasibility. Along the text, it will be clear which instance of the debugging engine, DDBG or Fiddle, was used in the project/prototype being reported .

Debugging (and debugging-related) tools, developed both locally and externally, contributed to the operational validation of the debugging engine, by exploring its services and functionalities in all layers, detecting malfunctions and deviations to the specification. These tools and their interactions with the debugging engine are described in the following sections.

Figure 5.1 on the next page illustrates some of the experimental work developed around the debugging engine.

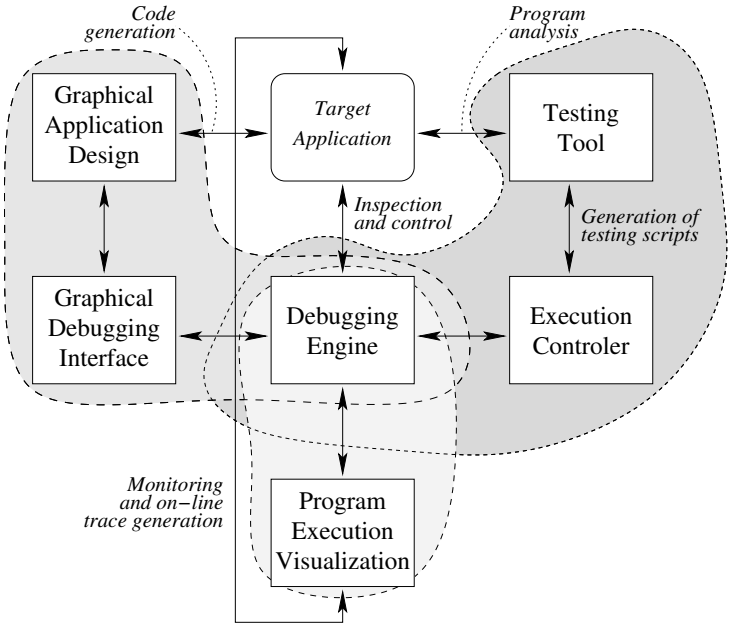

Figure 5.1: A debugging engine as the center of a testing and debugging environment

We would like to recall that the work on the Fiddle internal validation (Section 5.2), including Fiddle_J (Section 5.2.2), on Fiddle Consoles (Section 5.3), on FGI (Section 5.4.1) and on DEIPA and DEIPA$_2$ (Section 5.5), took place locally. The work on PADI (Section 5.4.2) and in the major part of the work on the DDBG integrations (Section 5.6) took place elsewhere, by third party research groups, but was closely followed by the author of this dissertation. The work on Fiddle integration with the (Pajé) visualizer (Section 5.7) was, essentially, at design level, and in cooperation with their developers.

5.2 Internal Validation

Each new Fiddle layer is built upon the previous (underlying) one, adding some functionalities. The new functionalities added by a layer are specific to that layer, but all functionalities from the underlying layer(s) are "inherited" and also supported by the new one. This means that there is a successive dependency and usage of any layer on the ones below.

5.2.1 Functional and Operational Dependencies Between Layers

According to the operational semantics defined for the debugging engine layers, from Layer0_s to Layer2_m all the services follow a synchronous model, i.e., the client thread invoking the Fiddle service will remain blocked until its completion and a reply (completion notification) is received. For Layer3_m, the intended operational semantic is event based, where the client thread, when requesting a service, receives a *requestid* and proceeds its computation. When a reply from Fiddle is available, a service handler in the client will be activated to process it.

The incremental development of the Fiddle layered software architecture became its first validation step, with each new layer validating the one immediately below. Fiddle layers are supported by a set of daemons and libraries, and any misconception or operational malfunction found at a specific layer would also be propagated and reflected on the upper layers.

5.2.2 Fiddle_J: A Java Object Oriented Wrapper for Fiddle Libraries

Access to Fiddle services is provided through a set of C libraries. As C++ is compatible with C, these libraries may also be used by client tools developed in C++. However, in this case, a procedural programming paradigm will be used instead of the expected object-oriented paradigm.

Access to Fiddle services from other programming languages which are not directly compatible with the C libraries must be supported on a case by case basis. Motivated by the external development of the PADI tool (see Section 5.4), the Fiddle_J library was developed to overcome such difficulties for Java based client tools, as illustrated in Figure 5.2.

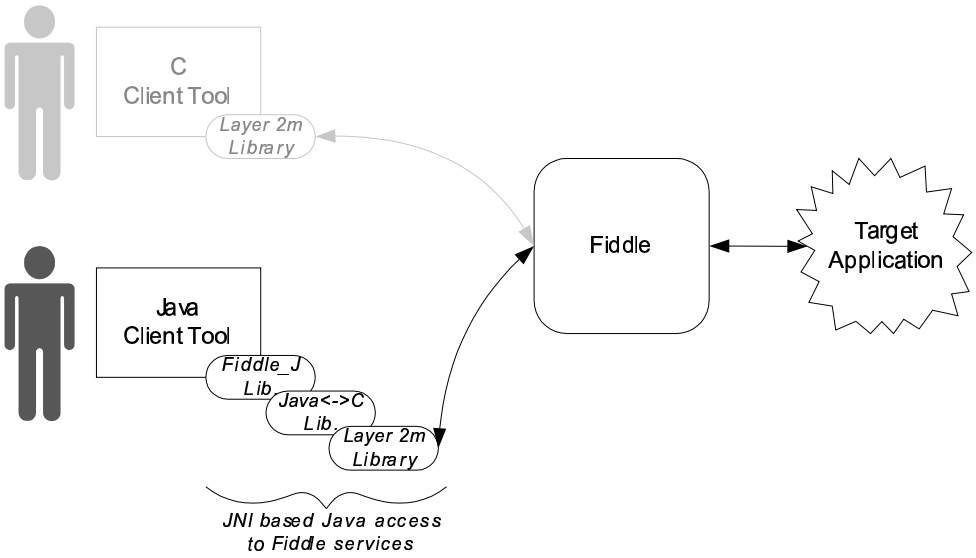

Figure 5.2: Java object oriented wrapper for Fiddle libraries (Fiddle_J)

Fiddle_J uses the Java Native Interface (JNI) [Coo97] technology to build a Java based object oriented representation of the Fiddle concepts and services available in the C libraries. By using the Fiddle_J library, Java applications that register as Fiddle client tools may coexist with other C and C++ based client tools, all concurrently interacting with the target program.

5.3 Debugging Consoles

As explained before, the debugging engine itself does not specify how the user will access its services. However, for testing purposes and as an example of how to develop client tools, the current Fiddle distribution includes a set of text-oriented command-line user debugging interfaces (the debugging consoles). There are, currently, four console versions, one for each debugging engine layer. The multiple consoles versions share a very large percentage (more than 95%) of the source code and are maintained along with Fiddle source code itself, being immediately updated to reflect any change in the specification of the debugging engine or in its implementation.

An example of a very short session with Fiddle consoles (using Layer 2_m) is shown in Figure 5.3.

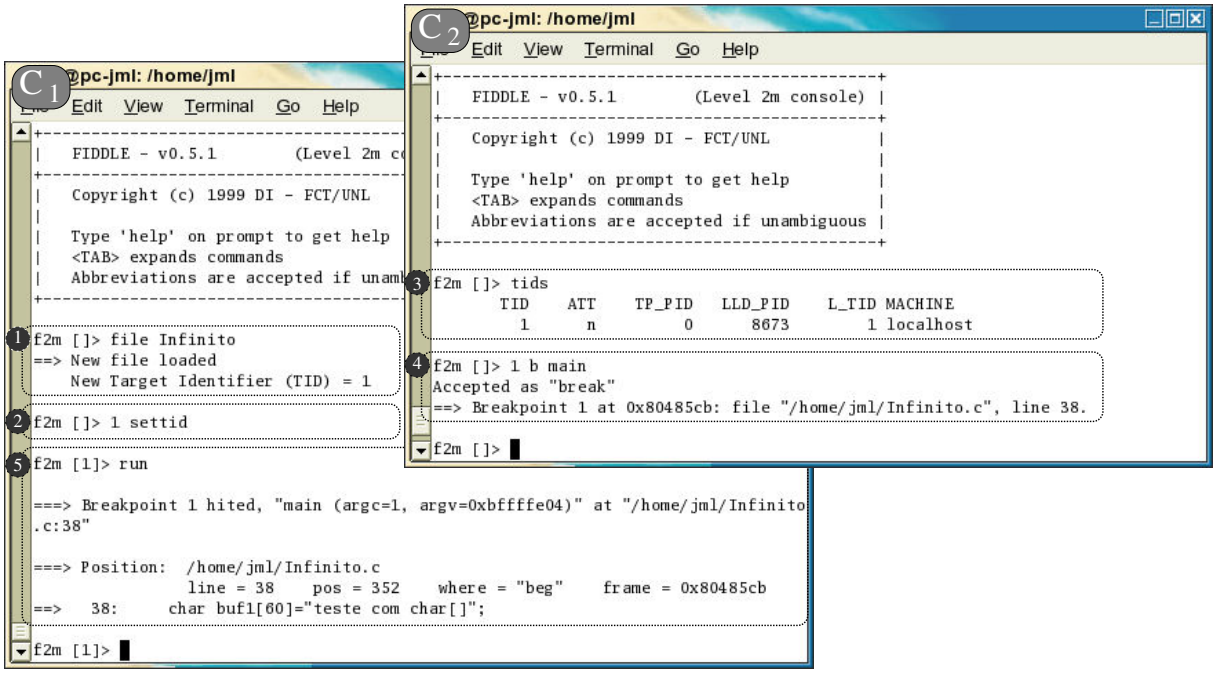

Figure 5.3: Two Fiddle (Layer 2_m) consoles operating upon the same target process

In this example, two Fiddle Layer 2_m consoles are operating upon the same target process. The session starts in console C_1 (on the left hand), by [1] loading a program named "`Infinito`" into memory, to which Fiddle gives the `ID=1`; then a debugging console specific command is used [2], to define the program just loaded into memory as the default target for the future commands; the two next commands were issued in console C_2 (on the right hand), [3] inquiring Fiddle about the IDs of all the current target processes and lists them (in this case there is a single target process, "`Infinito`", which was loaded from C_1 in [1]); and [4] which sets a breakpoint in the first line of the function "`main()`" in "`Infinito`"; finally, back to C_1 there is the last command [5], which was a "`run`" command. After this command, "`Infinito`" runs and stops at the breakpoint defined in [4] while using C_2.

The interaction between the debugging consoles just described illustrates that the debugging engine supports more than one client tool operating concurrently upon the same target process, as commands [3] and [4] depend on the previous command [1], and [5] also depends on command [4].

Figure 5.4 describes how Fiddle supports the two debugging consoles operating concurrently upon the same target process.

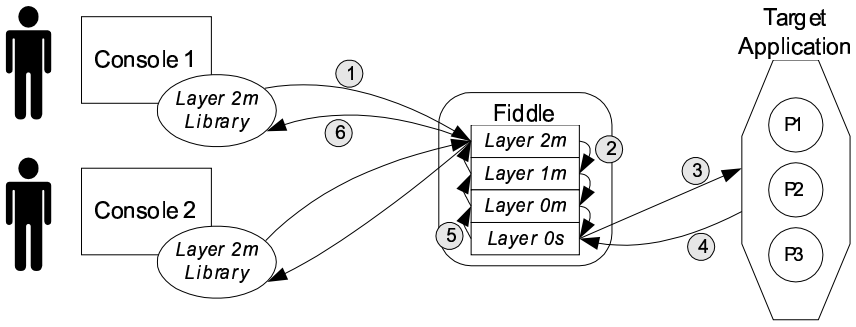

Figure 5.4: The interaction between the debugging consoles and Fiddle

In Figure 5.4, a Fiddle Layer 2_m service is requested by the client tool (a debugging console) by calling a library function. The library will transparently transfer the service request to the Fiddle core [1]. In the Fiddle core, the service is successively processed and transfered to the underlying layers [2], which includes a possible transfer over the network to the node where the target process is running. Then, the service is applied to the target process [3] and the result of such operation is sent back to the client tool [4], [5] and [6].

Besides showing a very simple example of the human interface of Fiddle consoles, this example shows how two Fiddle clients can operate upon the same target process, thus contributing to the validation of one of the main debugging engine functional requirements: the support of multiple client tools operating upon the same target processes.

5.4 Fiddle Graphical User Interfaces

Although simple and having small requirements on computational resources, text oriented debugging interfaces usually are not much appealing to novice users. The limitations on which information can be displayed in such environments and how it can be done imposes strong limitations on the assistance they may provide to the user. Additionally, text oriented debugging interfaces are not much user-friendly and, therefore, require a considerable effort from novice users to master them.

On the other hand, graphical user interfaces tend to be easy to use and new users can adapt to them very quickly. The ability to display information graphically, easing their interpretation by the user, is also a very important added value on the GUiS.

Concerning Fiddle, two separate efforts to produce more functional and user-friendly user interfaces led to FGI, developed at Universidade Nova de Lisboa, and to PADI, developed at Universidade Federal do Rio Grande do Sul. Both tools are described below.

5.4.1 Fiddle Graphical Interface (FGI)

The Fiddle Graphical Interface [Aug03] is, as the name suggests, a client tool for Fiddle that aims at providing a graphical user interface as an alternative to the Fiddle debugging console(s).

FGI provides two complementary main views for a running distributed program. One of these main views is organized in columns and provides the application browser, where the distributed program is observable as a set of conceptual layers, and where each layer provides a different view over the distributed program by using a specific set of abstractions. The second main view provides a browser which deals with the abstractions inherent to the specific layer.

The FGI user interface is illustrated in Figure 5.5, where examples of both main views can be depicted, namely the application browser and a process layer browser.

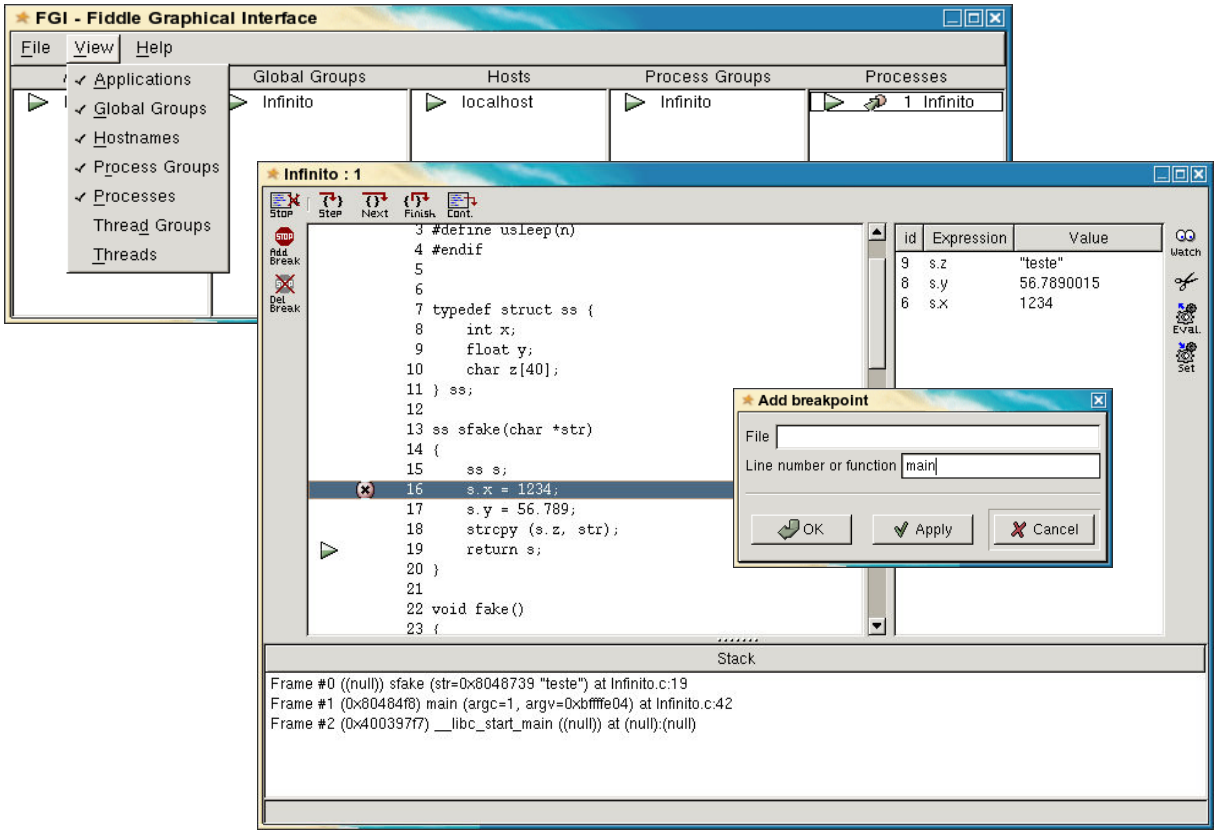

Figure 5.5: Fiddle Graphical Interface (FGI)

The FGI program browser considers the following layers:

i) *Application*. This layer includes all the processes under debugging (target processes), which form subset of the application processes. As current implementations of Fiddle and FGI assume that there is a single application being debugged, this layer could be omitted because it always contains a single component. However, such assumption will, probably, be relaxed in future releases, and this layer will then not only make sense but also be necessary. Any command applied at application level is broadcast to all target processes;

ii) *Global groups*. Correspond to an unrestricted dynamic association of target processes. Global groups may be freely created and destroyed by the FGI user at debugging time. The user may dynamically associate target processes to global groups, and a process may belong simultaneously to as many global groups as desired. There is always, at least, one global group which includes all the target processes. Any command applied to a group is broadcast to all target processes in that group;

iii) *Hosts*. Correspond to a dynamic association of processes to the computing nodes where they are running. Although dynamic, as target processes may be dynamically created or die, such correspondence is imposed by the execution environment and FGI, and the user has no control over it. As in previous cases, any command applied to a host group will be broadcast to all target processes in that group/host;

iv) *Process groups*. Correspond to a dynamic association of target processes in a single computing node. Process groups may be freely created and destroyed, and their contents defined by the FGI user at debugging time. Again, any command applied to a process group will be broadcast to all target processes in that group;

v) *Processes*. For process oriented distributed applications, processes are the basic unit of execution and are, therefore, at the lowest conceptual layer. For thread oriented programs, then the two layers below are also used and this layer is considered as a group which includes all the process threads. In this case, any command applied to the process will be broadcast to all its threads;

vi) *Thread groups*. Correspond to a dynamic association of threads. As with process groups, thread groups may be freely created and destroyed by the user at debugging time, but may only contain threads from a single process. The user may dynamically associate threads to thread groups, and a thread may belong simultaneously to as many thread groups as desired. Any command applied to a thread group will be broadcast to all the threads in that group;

vii) *Threads*. Sometimes there are multiple execution flows within a process. Threads are the basic unit of execution in such cases and are, therefore, at the lowest con-

ceptual layer. Threads and thread groups are to be used only when a thread oriented programming model is being used in the target program.

The content of each layer browser depends on the abstractions managed by that layer, but all of them are quite simple and similar except the process browser, that can be depicted in Figure 5.5 on page 81.

The process browser provides a more conventional interface to interactive (distributed) debugging, being divided into three panes: the top-left pane shows the process source file, the current line and existing breakpoints; the top-right pane displays the result of the evaluation of expressions, which may contain local and/or global variables; and the bottom pane displays the current execution call stack.

The operational relationship between FGI and Fiddle is identical to that of the debugging consoles and Fiddle. FGI may be used to debug any kind of distributed programs, independently of the communication model/libraries being used. However, FGI may explore the Fiddle ability to handle multiple client tools (in Layer 2_m) to provide a graphical debugging interface to PVM programs. Figure 5.6 shows how such support achieved.

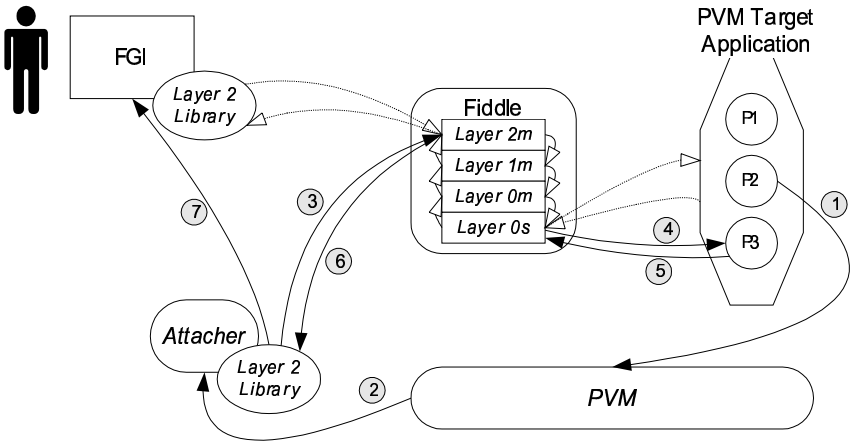

Figure 5.6: FGI support for debugging PVM programs

The spawning of a new process (P_3) by the currently running process (P_2) is intercepted [1] by PVM (such interception is based in the redefinition of the `PVM_DEBUGGER` environment variable) which will launch a user defined program [2]. In our case, this program is the new Fiddle client (*attacher*). This new program will register itself as a Fiddle client tool and request Fiddle [3] to spawn the new program P_3 [4] under Fiddle control. Once the process is loaded into memory and stopped under control of Fiddle [5], a notification of service completion is sent by Fiddle to the *attacher* [6]. The *attacher* will then notify FGI about the new Fiddle symbolic identifier [7] and will terminate. Upon receiving such notification, FGI will incorporate the provided information and update its display to reflect the newly created process.

Evaluation of FGI After some experimentation with FGI, we have concluded that the actual layer organization of the application browser could be improved, and a new set of layers is already planned, but not implemented yet. This alternative design will gather the previously defined *global-*, *hosts-* and *process-groups* into a single new *process groups* layer.

As before, the user will be allowed to create and destroy process groups and to dynamically associate processes to as many process groups as desired. In this new organization, FGI will automatically create as many process groups as computing nodes with at least one target process running (replacing, in this way, the "hosts" layer). The user may, in this way, easily check which processes are running where, and address all of them simultaneously by addressing the associated "host" group.

A more detailed description of FGI, its interface, functionalities and implementation, may be found in [Aug03].

5.4.2 PArallel Debugger Interface (PADI)

The first prototype of Parallel Debugger Interface was developed at the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS), by Denise Stringhini as part of the requirements for her PhD thesis [Str02, SNC00].

Quoting the PADI home page[1]:

> The PADI project aims at developing a debugging tool for support of parallel and distributed applications. It's a debugger interface that runs on top of PDBG library, that coordinates multiple process level debuggers attached to application processes. PADI is responsible to get processes information from PDBG environment and to present these information in a suitable way to the user.

Due to historical reasons, the PADI documentation refers to PDBG as the underlying debugging engine as in the above quote of PADI home page. However, such references to PDBG are wrong as Fiddle has been used as the debugging engine behind PADI from the very beginning.

The Figure 5.7 on the next page shows the look of PADI, with its two main windows: the *main view* and the *process view*. The former provides an application inspection and control environment, while the latter provides identical functionality for individual target processes.

PADI was developed in Java, which raised the question: "how to make it interact with Fiddle, whose libraries were C oriented?". The immediate solution for the PADI author was to have PADI assuming the role of Fiddle main daemon, and interacting

[1] http://www-gppd.inf.ufrgs.br/projects/padi/

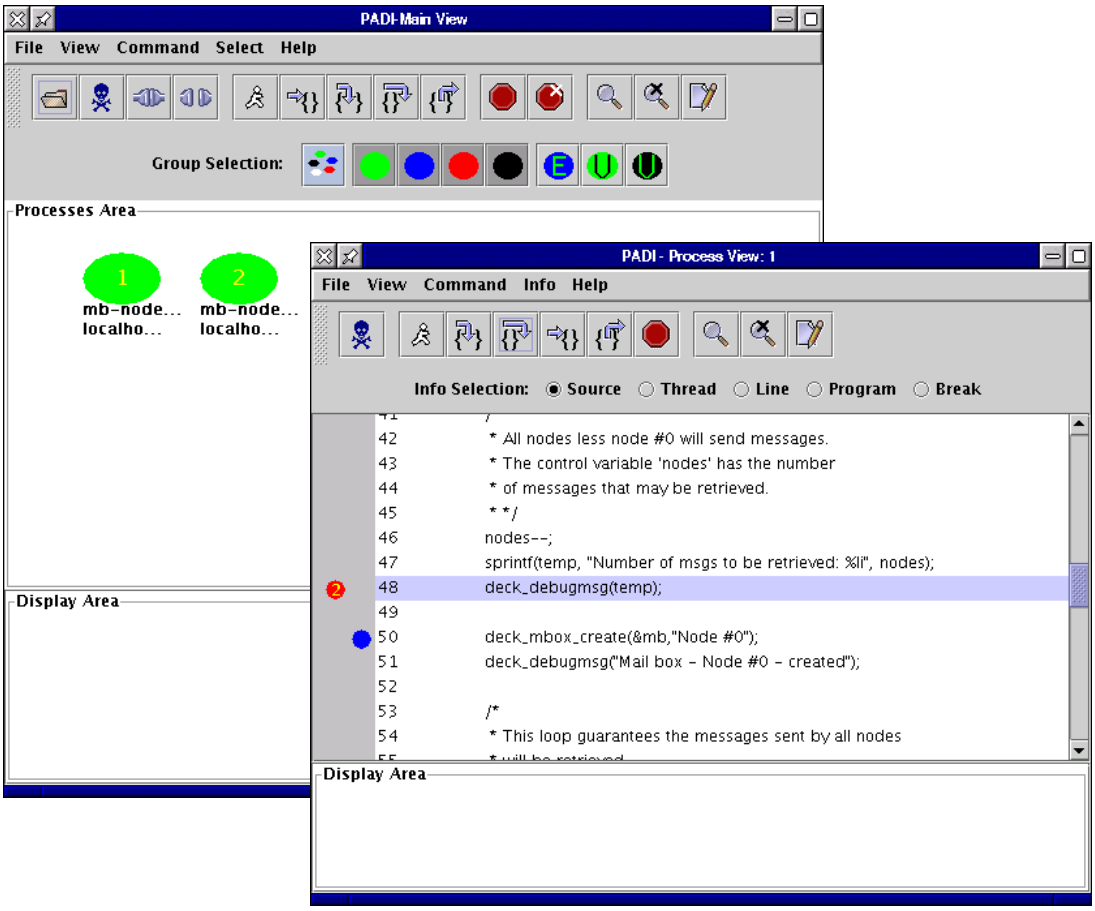

Figure 5.7: PArallel Debugger Interface (PADI)

with Fiddle local daemons by sending service requests in the internal (JML) codification format, and by decoding (from the same format) and processing the their replies. Figure 5.8 on the following page exemplifies this relationship.

Such design option imposed, however, strong limitations on the flexibility and maintenance of PADI. Some interesting features of Fiddle became unavailable in this way, such as the support for multiple concurrent client tools, and even a light change in the (supposedly internal) JML codification format would require the adaptation of PADI. To face such obstacles, a new object oriented Java library (FiddleJ, described in Section 5.2.2) was designed and implemented, relying on the Java Native Interface (JNI) technology. There is ongoing work on UFRGS to adapt PADI to use Fiddle_J instead of accessing the Fiddle local daemons directly as in the current implementation.

5.5 Composition of Testing and Debugging Tools

Although the development of high-level abstractions for distributed programming has contributed to ease the task of program development, there are still many opportunities for programming errors, posing the need for support tools.

Many programming errors can be detected, in a more or less automatic way,

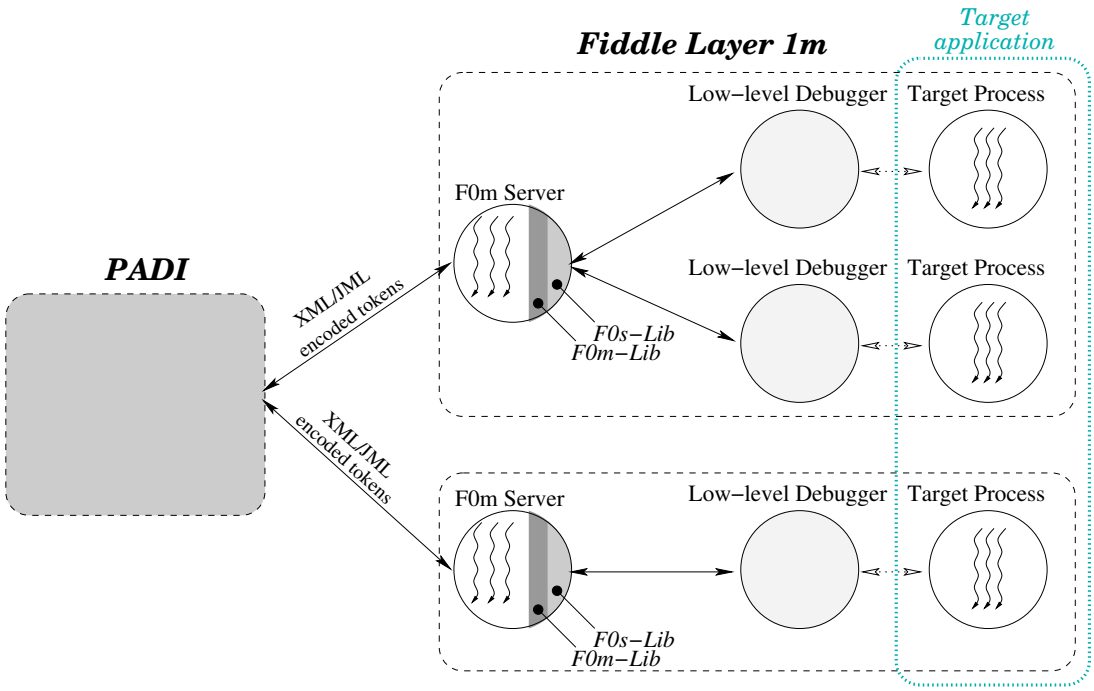

Figure 5.8: Relationship between PADI and Fiddle

through a static analysis of the program source text. Such analysis can also assist the programmer in the prediction of the program behavior concerning specific correctness properties. However, the program source text is not always available. Also, the global program behavior is often the result of a combination of behaviors, depending on the operating system or runtime support layers of a computing environment.

This explains the significance of approaches for dynamic analysis, which are centered upon the observation of real execution. Such approaches assume, from the beginning, the incomplete nature of this process, due to the usually huge number of computation states which can be generated by distributed program execution. As such, they are typically based on the selection of a finite set of representative test configurations, followed by an observation of the results of program execution. The definition of such test scenarios is, of course, dependent upon the classes of errors or program properties that one is trying to check.

The main goal of a debugging tool is to help analysing erroneous program behavior, possibly identified by a previous testing stage, and to assist the programmer in the formulation or confirmation of hypotheses on the causes of errors, and in the tracing of their origins in the program text.

Testing and debugging are naturally intertwined. On one hand, testing helps identifying errors whose causes must later be traced in a debugging stage. On the other hand, after a successful debugging session, one typically needs to reconsider the set of test scenarios. Debugging can also help identifying unforeseen situations which may require the design of new testing scenarios.

The above aspects have been recognized for a long time, leading to many proposals

of methodologies and tools for combined testing and debugging. See [KW99, CKW00] for more complete surveys on this topic.

5.5.1 Deterministic Execution and Interactive Program Analysis (DEIPA)

DEIPA [LCK+97] was initially developed to aid in the closing of a test-and-debugging development cycle for PVM programs, by allowing the composition of two separately developed tools, STEPS [KW96], developed at the Technical University of Gdansk, Poland, and DDBG [CLA99, CLD98]. Later it was redesigned and reimplemented using Fiddle in DEIPA_2, as described in [Mor03, LCM03]. Figure 5.9 illustrates how such cooperation was achieved.

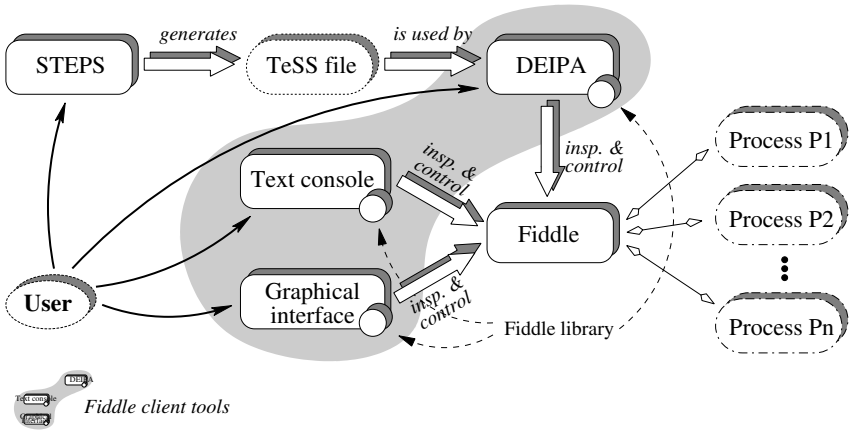

Figure 5.9: Tool composition of STEPS and Fiddle using DEIPA

Based in static and dynamic analysis of the target program, STEPS generates a behavior specification file, the TeSS file. Figure 5.10 on the following page shows a toy example of a PVM program and the corresponding TeSS file. This file includes the definition of a list of *global breakpoints* (collection of consistent local breakpoints). Some of those global breakpoints have special instructions to modify one or more process variables, which is a necessary operation to correctly drive the application flow path when in presence of conditional branches or loops, e.g., on `if` and `while` statements. For a more detailed explanation of the example presented in Figure 5.10 on the next page see [LCM03].

To force the application execution to conform to the specification in the TeSS file, DEIPA_2 loads this specification file and generates the necessary sequence of debugging (control and inspection) commands to Fiddle, setting a local breakpoint in each target process and individually driving them until the global breakpoint is reached. Once a process is stopped in a breakpoint, its internal status (e.g., variable contents) are changed if needed, according to the TeSS specification.

At any time, Fiddle ability to handle multiple clients may be explored, and the user may switch to another tool, such as Fiddle's debugging console or graphical interface,

echo_client

```c
#include <pvm3.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>


int main ()
{
  int mytid;
  int totid =0;
  int value;


  mytid = pvm_mytid ();

  value = pvm_spawn ("echo_server", NULL,
            PvmTaskDebug, ".", 1, &totid);

  /* Sending 0 will force the server to
     exit, and the client will wait
     forever for the reply */

  value = 0;
  pvm_initsend (0);
  pvm_pkint (&mytid,1,1);
  pvm_pkint (&value,1,1);
  pvm_send (totid, 1);

  /* Get the reply from server */
  pvm_recv (-1,-1);
  pvm_upkint (&value,1,1);
  printf ("Received value %d\n", value);

  pvm_exit();

  return (0);
}
```

echo_server

```c
#include <pvm3.h>
#include <stdio.h>
#include <stdlib.h>


int main ()
{
  int mytid;
  int from, value;

  mytid = pvm_mytid();

  pvm_recv (-1, -1);
  pvm_upkint (&from,1,1);
  pvm_upkint (&value,1,1);

  if ((value%2)==0)
    exit(0);
  else
    value=-1;

  pvm_initsend (0);
  pvm_pkint (&value, 1, 1);
  pvm_send (from, 1);

  pvm_exit();

  return 0;
}
```

TeSS file

```
START_FILE:
    echo_client

SPAWN_TABLE:
    {
        0   0   0   0   1   echo_client echo_client.c
13142 4,
        1   16  0   1   2   echo_server echo_server.c   59634 2
    }

INITIAL:
    [{ (1,1,17) }],
    [{ (1,1,28) }],
    [{ (2,1,28) }],
    [{ (2,1,28),(1,2,13) }],
    [{ (2,1,28),(2,2,13) }],
    [{ (2,1,28),(1,2,17,[2,2,"value","1"]) }],
    [{ (2,1,28),(2,2,17) }],
    [{ (1,1,31),(1,2,24) }],
    [{ (2,1,31),(2,2,24) }],
    [{ (1,1,35),(1,2,26) }],
    [{ (2,1,35),(2,2,26) }];
```

Figure 5.10: Sample PVM programs and TeSS file

and perform a more detailed inspection and control of the target processes.

On completing such a fine (process-level) debugging, the user may return to DEIPA$_2$ and proceed with the controlled execution mode, or release/stop the target program if no more debugging is needed.

As described before with FGI, DEIPA$_2$ also makes use of the Fiddle ability to support additional concurrent clients to intercept newly created (spawned) PVM processes and acquire control over them. Such operation in DEIPA$_2$ is identical to the one in FGI, as described in Figure 5.6 on page 83, as if the client tool FGI had been replaced by DEIPA$_2$.

5.6 Integration in Software Development Environments

The development of distributed applications requires the programmer to define and specify a set of program components which will interact, most probably, by accessing some common memory (repository) or by sending and receiving data messages. In any case, this high-level planning of the application is well suited for the use of graphical specification/programming languages.

Graphical programming languages are naturally supported by graphical editors, which allow to express the language constructs. The graphical program will, most probably, be converted to some more conventional programming language such as C or C++ by way of an automatic code generator, and just then compiled to machine code. Such code generators frequently rely upon some communication layer/package, such as OpenMP, PVM or MPI.

The development of distributed applications can be further improved by associating the graphical editor to a full set of other complementary tools, such as program simulators, monitors, visualizers and debuggers. The SEPP and HPCTI European projects [WK94] led to the development of two Integrated Development Environments (IDE) for distributed applications, the GRADE [KDL00] and EDPEPPS [D$^+$00] environments.

In both of these projects, the DDBG distributed debugger, a predecessor of Fiddle, was integrated into the environment and used to provide graphical debugging and execution animation support for the (graphical) programming languages of both environments.

The experiments reported below have not been repeated by replacing DDBG with Fiddle. The full source code for those toolsets is not available anymore, and the cooperation with the toolset developers has also terminated. Although any real attempt to replace DDBG with Fiddle in this context is, therefore, impossible, we also include a description on how this replacement of DDBG with Fiddle could be achieved.

5.6.1 Integration of DDBG in GRADE

GRADE [KDL00] stands for Graphical Application Development Environment, and is an IDE for distributed applications, developed as a cooperation effort between multiple partners of the SEPP and HPCTI European projects [WK94].

GRADE was composed of the following tools:

i) *GRAPNEL* [GD95a], developed at KFKI-MSZKI, Hungary. GRAPNEL is a hybrid language, using both graphical and textual representations to describe a distributed application. The graphical entities are used to describe the main program components and communication channels. Textual declarations are used to describe short sequential code segments. The main purpose of the graphical representation was to give a high level outline (or abstraction) of the distributed program where the key points are the communication operations among the processes;

ii) *GRED* [GD95b], developed at KFKI-MSZKI, Hungary. In GRADE, parallel programs can be developed according to the syntax and semantics of GRAPNEL language by using the GRED editor;

iii) *GRP2C* [DDK00], developed at Research Laboratory for Mining Chemistry, Hungarian Academy of Sciences. GRAPNEL graphical programs are saved in a structured textual representation, and processed by GRP2C to generate C code which use the PVM library for communication support;

iv) *DDBG* [CLA99], developed at Universidade Nova de Lisboa, Portugal. A distributed debugger with interesting capabilities on tool integration;

v) *Tape/PVM* [Mai95], developed independently at LMC-IMAG, Grenoble, France. A monitoring and visualization tool, able to generate trace files during the execution of a PVM application. Tools such as ParaGraph [HF97] or PROVE (see below) can be used to present graphical displays of such traces.

vi) *PROVE* [KCMV00], developed at KFKI-MSZKI, Hungary. A visualisation tool to analyse and interpret the Tape/PVM trace file information, and present it graphically to the programmer.

The GRAPNEL model, a graph-based parallel programming language, supports a structured style for designing parallel applications. In this integrated development environment there is a requirement that the debugging commands and output information be directly related to the GRAPNEL model, such that only GRAPNEL abstractions should be handled by the user. This requires a high-level interface to the user, such that the information on specific debugging commands is directly related to the

GRAPNEL source program, e.g., by highlighting corresponding entities in the graphical representation, and their corresponding lines of source code in the textual program representation.

For each debugging action specified in GRED (the GRAPNEL editor), process names are converted to real PVM task identifiers and the necessary DDBG services are invoked. This is exemplified in Figure 5.11, where process "`worker11`" was stopped at a breakpoint, in the fourth source line of its code block.

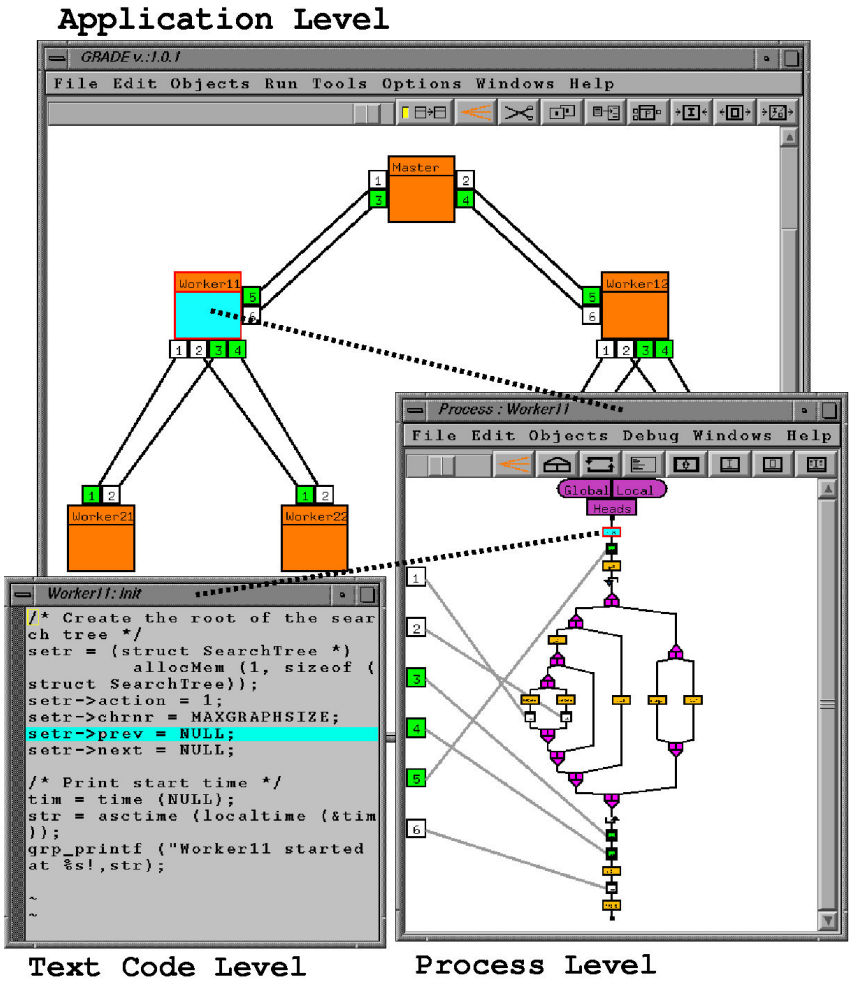

Figure 5.11: The integration of DDBG within GRADE

During this integration work, an important issue was raised. It was related to the limitations imposed by DDBG for the client tool to be a single-threaded program in conjunction with the RPC model for debugging service requests. The GRED visual programming editor was an event oriented single-threaded program (as the generality of the X-Windows applications) and the RPC model for debugging services was blocking temporarily the graphical interface until the service was completed. For the majority of the debugging services this was not a critical issue, as they would either succeed or fail in a limited time window, but a few services, such as *continue execution until a breakpoint is found*, would take an unpredictable time to be completed, during which the graphical user interface could not be frozen.

The solution found was to make the GRED tool to rely upon a TCP/IP socket based communication channel to interact with the DDBG system, where GRED would receive notifications of the completion of those potentially long time running debugging services. Figure 5.12 illustrates the approach just described.

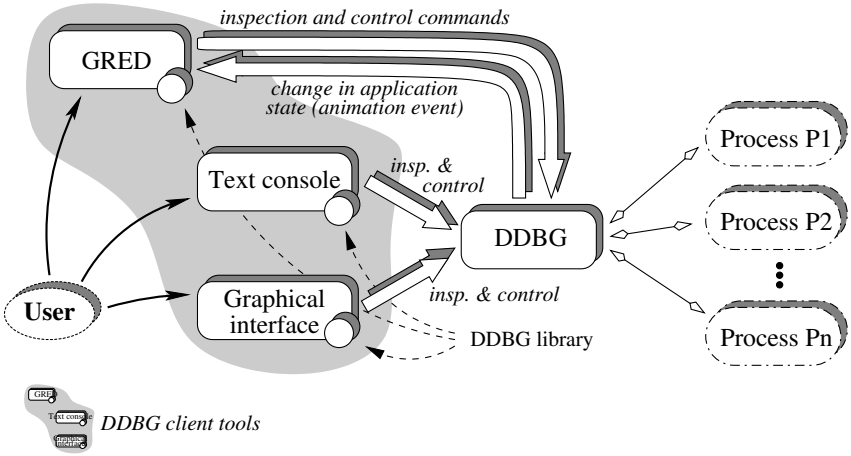

Figure 5.12: Support of long time running debugging services in GRED

Most of the debugging service requests issued by the GRED editor follow the RPC model. They are transparently sent by way of the DDBG library [1] to DDBG, where the service request is processed. DDBG sends a reply back to the DDBG library [2] with the result(s) of the requested debugging service.

While the above operations take place, the graphical user interface (GRED) would be blocked waiting for the reply from DDBG. In order to avoid this behavior, the small number of long time running debugging services (such as "continue execution until a breakpoint is hit") are also sent to DDBG by the DDBG library [1]. A reply informing that the requested service was accepted is immediately sent back by DDBG [2] to the client tool. Such reply unblocks the graphical interface, which will, in this way, be available while the service is being executed by DDBG. Once the requested service is completed, its reply is sent to the special communication channel (TCP/IP socket). When there are pending replies, GRED periodically inquires the communication channel to check their availability.

The above solution has proved its feasibility when the integration was successfully achieved. However, it was an *ad-hoc* solution, with special requirements upon the client tool, namely the ability of periodically checking the communication channel and, certainly, such requirements could still be applied to some other client tools, but not to all of them.

5.6.2 Integration of DDBG in EDPEPPS

The EDPEPPS [D+00] toolset is based on a rapid prototyping philosophy, such that outline designs can easily be experimented, and evolve to full distributed programs.

EDPEPPS offers many advantages over traditional parallel design methods, like a rapid prototyping approach to parallel software development, offers modularity and extensibility through layered partitioning of the program, supports the user on deciding on the size and scalability of the target platform, but also has the ability to do some performance analysis without accessing the target platform, and allows the software designer to perform the development cycle of design-simulate/execute-analysis without leaving the toolset environment.

All the above functionalities of the EDPEPPS toolset are based on the following components:

i) The PVM graphical programming language (PVMGL), which allows PVM applications to be developed using a combination of graphical objects and text;

ii) A graphical design tool (*PVMGraph*) [Jus96], which allows the user to edit PVMGL programs;

iii) A simulation utility (*PVMPredict*) [D$^+$00], based on discrete-event simulation, to simulate the execution platform and multiple software (and communication) layers;

iv) A visualization tool (*PVMVis*) [D$^+$00], for the animation of program execution based on traces generated by the simulator, and for visualization of platform and network performance measures and statistics;

v) A debugger (*PVMDebug*) [Aud98], which allows to execute a PVMGL application under control of a debugger and animate this execution.

PVMDebug (see Figure 5.13 on the next page) development was based on DDBG, and similarly to the integration of DDBG in GRADE, provides debugging support and execution animation within the EDPEPPS integrated debugging environment.

Like in GRADE, programs in PVMGL are automatically converted to C source code, which is displayed in a pane in PVMDebug highlighting the current execution pointer (line). The basic debugging functionalities, such as breakpointing and single-stepping, are available through a set of buttons available in the same pane as the source code.

PVMDebug also sends events to PVMGraph to animate the target program execution by changing the look of displayed PVMGL entities, reflecting in this way the changes in the status of the target application. PVMDebug allows the programmer to set/clear breakpoint from the source code pane or from PVMGL entities

In order to debug a distributed PVM application and to control its tasks, the debugger needs to know all PVM task identifiers (`tids`). PVMGraph cannot anticipate which `tid` will be given to each application process, as they are generated by PVM at runtime and change at every run. Therefore, a wrapper program was developed to manage the link between the graphical objects and the PVM execution tasks. The wrapper program

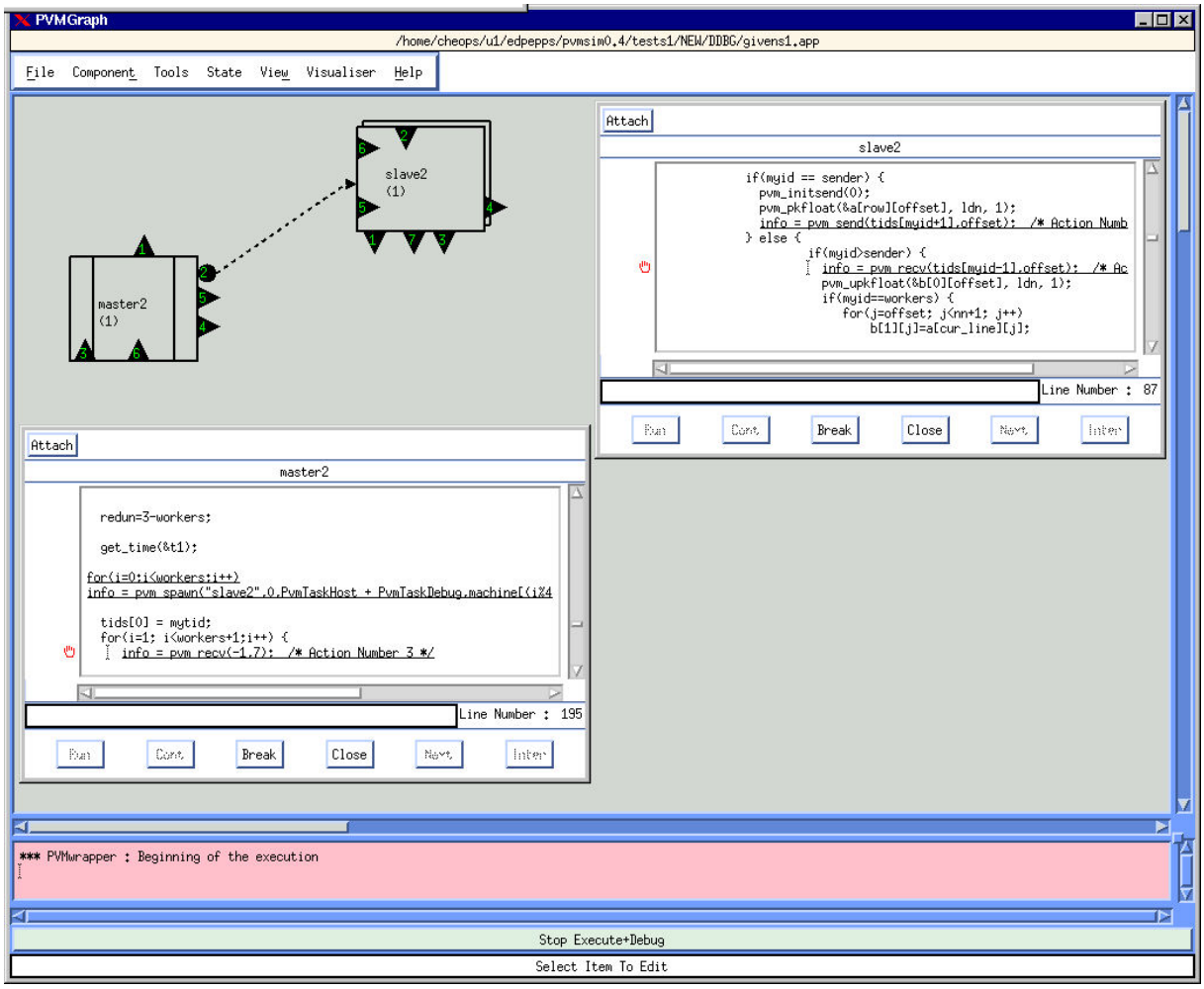

Figure 5.13: The EDPEPPS PVMDebug main window

uses the on-line monitoring facility in PVM to collect the PVM events from the application and to route them to PVMDebug which manages the mapping list of PVMGraph symbolic identifiers and PVM `tids`.

The integrations of DDBG in EDPEPPS and in GRADE raised similar problems, which were solved in a similar way in both cases. In particular, the most significant of those problems, the blocking of the user interface when long time running debugging services were requested to DDBG, were also solved in EDPEPPS as described before for GRADE.

5.6.3 DDBG vs. Fiddle Support for Debugger Integration in PSDE

Although both integrated development environments, GRADE and EDPEPPS, were developed by different research teams, they had similar goals and used similar approaches to the development of parallel/distributed programs. As such, similar problems were also detected when the work to integrated DDBG in such environments was initiated, and identical solutions were also applied in both cases.

The details of the problems and solutions encountered for both, GRADE and EDPEPPS, are described in [Aud98] and [KCD+97] respectively. A summary of those

problems and their solutions is listed below. Along with each item, we also included a suggestion of a solution in case Fiddle were to be used instead of DDBG.

i) *The mapping of high-level (graphical) language constructs to generated source code*. Both EDPEPPS and GRADE are based on graphical programming languages with high-level constructs, and use code generators to produce C/PVM source code. DDBG, operating at source code level, couldn't deal wit those graphical constructs. The necessary correspondence between the graphical constructs and the C/PVM source code, being toolset dependent, should be done at code generation time. However integrated in the development environments, the debugging requests directed to DDBG would, in both cases, use the C source code as the basic reference.

   If Fiddle were to be used instead of DDBG, the same references to the C source code would be acceptable (and desired), and all the above statements could (and would) be applied. This means such replacement would be simple and with no major implications;

ii) *Mapping from symbolic to real process identifiers*. As software design tools, PVMGraph and GRED deal with symbolic program identifiers, which are constant. PVM, as an execution environment, deals with real process identifiers, which may (and most certainly will) change at every run.

   DDBG could only identify processes by a pair "(hostname, pid)", and the symbolic process identifiers used in the design phase had to be mapped into the real process identifiers used by the runtime system. This functionality was achieved by the addition of a mapping module to the toolset.

   Fiddle, would also need a similar identification module, where the pair "(hostname, pid)" used in DDBG would be replaced with the unique process identifier used by Fiddle;

iii) *Handling of multiple concurrent client tools*. This was one of the key issues in the design of the debugging engine, and both DDBG and Fiddle support such functionality. Fiddle, due to the layered software architecture and extensibility, is more versatile than DDBG and the integration work would be considerably easier. However, for the toolset user (software developer) there would be no significant difference;

iv) *A PVM event handler*. This small program is designed to gather and filter PVM events, such as process creation and termination, and inform the graphical user environment of such changes. The information supplied is crucial for the implementation of the symbolic to real process identifiers mapping referenced above. Again, this is independent from the debugging system being used, whether it would be DDBG or Fiddle, and needed in both cases;

v) *Grabbing of new (dynamically created) application processes*. Some distributed programs assume (or are limited to) a initialisation phase, where a static set of application processes are created, a computation phase, where these processes cooperate to achieve the common goal, and a termination phase, for the partial results gathering and graceful termination of the application. If such organization is promoted or even enforced, such as when using MPI, then there is no dynamic creation or termination of processes during computation phase and this problem does not arise.

However, on some other systems, such as PVM, the dynamic creation and termination of processes during the life-time of the distributed program is allowed. In such cases, if the application is under debugging (and unless stated otherwise), the developer expects any newly created process to also be under control of the debugging engine.

To debug PVM programs with DDBG, this desired functionality was achieved by a combination of PVM and DDBG features. PVM allows newly created processes to be started under control of a debugger, and this debugger is user configurable. DDBG, on the other hand, supports multiple client tools and DDBG specific client tool, the *pvm_spawn_handler*, was defined to be the PVM debugger. This spawn handler receives, as an argument, the pathname of the program to be launched as a new process, registers itself as a DDBG client, asks DDBG to load the specified file, and terminates. In this way, the newly created process is under control of DDBG.

The above described technique has also been in Fiddle, to allow FGI and DEIPA$_2$ to handle PVM programs.

vi) *Handling of potentially long completing time debugging services*. All services provided by DDBG used a synchronous calling model. This means the calling process would remain blocked until service completion. This was not a problem for quickly handled services, e.g., setting a breakpoint or listing a set of source code lines, that would either succeed or fail in a short period of time.

Debugging services, that would possibly take a long time to succeed, e.g., continue execution until next breakpoint is reached (which may as well take a fraction of a second as a few hours or days to happen), became a problem when using this calling model, as it could block the entire graphical user interface for a long time. Such blocking was very annoying and unacceptably limited the user access to the target program during this period.

To avoid such undesired behavior, DDBG was changed so that potentially long completion time services would return immediately a success status, terminating immediately the RPC call. The service completion notification/data would be sent later through an alternative communication channel (a TCP/IP socket). The

graphical user interface was responsible for periodically polling this communication channel (the socket descriptor) and react when data was present.

If Fiddle Layer 2_m were used instead of DDBG in these integrations, the extra TCP/IP socket would be superfluous. Fiddle handles multi-threaded clients, so, the graphical user interfaces just had to start a new thread in every service request to Fiddle or, at least, to every request of a potentially long time completion service, leaving to the main tread the control of the graphical user interface. Once the service was completed, the thread that was waiting for the completion notification could insert a new completion event in the event queue of the main thread. This model of solution was used with success in FGI.

If Fiddle Layer 3_m were to be used (we recall that Layer 3_m is not implemented yet), a completion event would be generated by Fiddle and handled synchronously or asynchronously by the GUI, according to the event management model chosen by the GUI. It could either block waiting for the notification event, proceed and periodically poll for its existence, or have a thread to be automatically created when the event was received. In any case, when the notification was received, a new event could be inserted in the event queue of the main thread, to have the GUI reacting to the notification.

vii) *Adaptation of the graphical editor to support program animation and debugging functionalities*. Both toolsets, GRADE and EDPEPPS, were adapted to become DDBG client tools. As DDBG client tools, these toolsets could, periodically, inquire DDBG about possible changes in the status of any of the processes being controlled and react accordingly. For example, in both toolsets, when a process hits a breakpoint and, thus, stops running, its associated color in the GUI is also changed to reflect the new process status.

The improvements in Fiddle to handle debugging services which may take a long time to complete, as described above, and in particular its event notification mechanisms and support for multi-threaded client tools, could be explored by the toolset GUI as an easier way to implement program execution animation.

5.7 Integration with a Visualizer

Distributed program visualizers [AG, MHC94, KGV96, KS00] work, mainly, as post-mortem tools. At run-time, program (communication) events are collected using low-intrusion tools/libraries and dumped to trace files. Visualizers read such trace files, process their contents and generate different kinds of graphical displays, aiming to help the user better understand the program behavior. However, once an error is detected, it is a programmer's duty to determine where in the source code such misbehavior is generated.

An adequate combination of the functionalities provided by a visualizer with those of a distributed debugger can improve significantly the productivity of the software developer. These tools may correlate the information provided by the visualizer, such as a point in a graph shown by the visualizer, with the program source code line or a message buffer.

Such combination require multiple adaptations to the original tools. The trace file generator needs to be adapted to produce an event stream which will be consumed on-line. The visualizer must also be adapted to operate on-line with a running application (opposed to the usual *post-mortem* operation), which means it has to generate significant displays with a partial knowledge of the event history.

For example, if by analysing a process-time graphical displays of a running program, provided by the visualizer, the user detects that a message is being delivered to the wrong process, it would be desirable to stop the application, select the send and/or receiving event(s), and have the debugger display the associated points in the source code of both processes, along with the processes stack frames at the moment (if still available).

There are plans to realize such kind of integration using Fiddle and the Pajé [KS00] visualizer (developed at LMC-IMAG, France, in the context of the ATAPASCAN project), probably using the DAMS [CLV+98, DLC01] (from UNL) infrastructure to support the application monitoring and event dissemination to the tools. Some preliminary work has been done under a cooperation agreement between UNL and LMC-IMAG, and another cooperation agreement between UNL and Universidade Federal do Rio Grande do Sul and the Universidade de Santa Catarina, both in Brazil, was already submitted and waits for approval from the Portuguese and Brazilian states.

5.8 Summary

The debugging engine was designed with special concerns on tool cooperation and integration. The goal was to allow the debugging engine to interact not only with specific client tools, but also with other third party (set of) tools. Such aims were validated by using DDBG and Fiddle in a set of projects, ranging from interaction with simple client tools, such as command line and graphical debugging interfaces, to its integration in full parallel software development environments.

There were many experiments with the debugging engine involving different kinds of tool inter-relations:

i) *Specific clients*. The debugging engine includes a APIs (one for each level), but no debugging user interface. To ensure the correctness of its implementation and the effectiveness of its APIs, specific debugging engine clients have been developed. Some were in the form of new APIs, such as those provided by the internal upper layers, i.e., $\mathsf{Layer}0_m$ is a client of $\mathsf{Layer}0_s$ and uses its API, $\mathsf{Layer}1_m$ is a $\mathsf{Layer}0_m$

client and uses its API, and so on. Others were simple client tools, such as the Fiddle debugging consoles and FGI.

These experiments proved that the layered software architecture was a convenient and flexible approach to the organization of the debugging services to be provided by the debugging engine. Each new layer was also the first testbed for the preceding one;

ii) *Loosely-coupled cooperation with other tools*. Tools may share little or no knowledge at all of each other internals or behavior. This frequently results in a unidirectional (pipeline) cooperation based in some intermediate file(s) with output/results which pass data from one stage to the next.

A good example of such loosely-coupled cooperation involves the trace file generators, which collect and log the (communication) events in a distributed computation, and the computation visualizers, which display graphical representations (for example, space-time diagrams) of the computational trace, as described by the previously generated log. Once the trace file format is defined, the trace generator may ignore how the data it produces will be used, and the visualizer may ignore how the data it uses was collected.

The experiments of DEIPA (and DEIPA_2), which explored the cooperation between the STEPS testing tool with DDBG (and Fiddle), are another example of such loosely-coupled cooperation, with a file—the TeSS file—playing a main role as intermediary between the testing and the debugging tools.

iii) *Tightly-coupled cooperation with other tools*. Some tools share a very close relationship with the debugging engine and deal with different concepts and/or abstractions the ones initially provided by the debugging engine. This is the case of PADI, which provides the user with a set of group oriented functionalities, by relying on Fiddle for the basic (non-grouping) debugging services. This illustrates a tightly-coupled cooperation between two tools, PADI and Fiddle.

Tools tightly-coupled may also share even a tighter relationship, such as direct interdependency. In this case, tools will share a deep knowledge on how each other operates and what functionalities they provide. This knowledge will allow the involved tools not only to cooperate, but also to give access to their own functionalities which better complement the ones provided by the other tool.

Two of the previously described cooperations, DDBG/GRED and DDBG/PVMGraph, are examples of such interdependency, where adaptations were made to both tools to enable a fruitful cooperation;

iv) *Full integration in PSDEs*. Tool interactions may also involve a full set of development tools, with complementary functionalities, which cooperate to assist the

users with an integrated development environment for parallel/distributed applications. In this case, there is an increased complexity level, due to the cross inter-relations between tools.

The integrations of DDBG within the GRADE and EDPEPPS parallel software development environments fall into this category. More important than the additional adaptations that may have been (or not) necessary to obtain a full integration, it is the fact that a consistent and uniform user interface has been provided to the developer, supporting the high-level development concepts and abstractions and hiding, by principle, the lower level details, such as automatically generated code.

6

Conclusions and Future Work

Contents

This Chapter summarizes the main achievements of the research work described in this dissertation, and discuss some still open issues, which will be considered in our future research work.

6.1 Conclusions

This thesis proposes a debugging engine, resulting from a research work which was strongly motivated by its context, namely international cooperations to design and develop a set of cooperating tools which could be presented to the user as an integrated parallel software development environment (PSDE). The proposed debugging engine had a number of goals, has described in Chapter 1.

We have designed a debugging engine which could fulfill the debugging requirements posed by other tools in a SDE, and planned and implemented its software architecture. Such software architecture is based in a set of layers, where each layer includes a set of functionalities, which are extended by each new layer. The need for future extensions was also predicted in the basic mechanisms provided by the debugging engine, and such possibility was included in its design.

The debugging engine has support for the observation and control of individual distributed processes, supporting interactive correctness debugging and providing symbolic (source level debugging) functionalities. Support to handle some of the problems inherent to distributed computations, such as the non-determinism, were also provided in the form of extensions to the basic debugging engine. We have also described two different implementations of the debugging engine.

DDBG was an early prototype of an early specification of the debugging engine, which was used in a set of successful experiments. These experiments range from the simple provision of distributed debugging command line interfaces, to its application in the support of the testing and debugging sub-cycle of the software development process, and to the support of visual parallel programming languages integration with graphical editors.

Fiddle was developed later, as a re-design of the debugging engine, and led to the version described in this dissertation. It includes support for: traditional debugging services, distributed debugging services, integration into PSDEs, and also the possibility of being extended with new debugging services. Fiddle was also successfully used in a set of experiments, which helped to validate its design and implementation, as described in Chapter 5.

6.2 Future Work

In what concerns the future, many things are to be done. Some of them are very well identified and closely related to the Fiddle implementation, while some others have a larger scope, targeting future research initiatives.

Concerning the debugging engine and its current implementation, we plan to do some more work on the following aspects.

i) $Layer3_m$. Although generically defined, in what concerns its functionalities and

operational mechanisms, there are still some open issues concerning this layer such as the identification and enumeration of the events which will be reported by the debugging engine, their classification and organization in sets, the definition of the supported notification mechanisms, and how will these notification should be handled in the client tools. Once the decisions concerning the debugging engine are made, they also have to be instantiated in the current implementation of Fiddle engine. We anticipate that some minor modifications to the already existing layers and data structures will be needed to efficiently accommodate the operational requirements of Layer 3_m;

ii) *Further evaluation of XML as an external data representation*. Some work has already been done to evaluate the communication performance in Fiddle, by comparing the originally defined (proprietary) JML protocol to the alternative implementation which used the XML standard [Mor03]. However, some more work could still be done to further evaluate the latency and communication times using alternative XDR protocols, such as RPC-XML and the ones supported by message passing systems, such as MPI, and compare them to the already obtained results;

iii) *Support of GDB's MI_2 command language*. Since GDB version 5, an alternative command language directed towards the use of GDB as a sub-process receiving commands from another program (instead of a human user) named "Machine Interface (MI)" has been under development [GDB]. The most recent version of GDB (version 6) includes a new version of the machine interface language, the MI_2, which we would like to test and evaluate [SP93];

iv) *Support for alternative node debuggers*. All our experiments with DDBG and Fiddle relied on GDB as the node debugger. We anticipate that Fiddle can easily be adapted to support node debuggers other than GDB, however, such experiments have not been conducted yet;

v) *Support for alternative node debugging engines*. A possible alternative to the use of node debuggers is to use node debugging engines, such as Dyninst [BH], available as a library. This could easily be achieved by replacing the node debugger with Dyninst library and by replacing the command generation and reply processing modules from Fiddle with equivalent ones for the Dyninst library;

vi) *Further testing of Fiddle_J* . A first implementation of Fiddle_J is finished, to which some basic tests with toy examples have already been applied. However, further testing is necessary, probably by the development or adaptation of a full client tool which will make use of this library to access Fiddle services;

vii) *Support for other message passing libraries*. All the experiments until now, with both DDBG and Fiddle and involving message passing based distributed programs, have

been based in the PVM system. Minor experiments have been made to verify the ability of Fiddle to support also MPI based distributed programs, but further testing is necessary.

In what concerns existing Fiddle client tools, there are also some open issues:

i) *Continue FGI development*. The current FGI prototype was limited in its objectives and development time. There is space for a lot more improvements in its user interface, and on the functionalities it provides;

ii) *Adaptation of PADI to use Fiddle_J* . Although PADI was the main motivation to develop Fiddle_J, due to the independent development scheduling of PADI, its current implementation does not uses Fiddle_J. PADI authors have already demonstrated their interest in adapting its tool to use Fiddle_J, and we expect work on such adaptation to be started soon;

Finally, the current implementation of the debugging engine has proved to be a stable software package. Such stability is essential to pursue further research work concerning distributed debugging, such as:

i) *Integration of the debugging engine and a visualizer*. Some prior studies have already been taken considering the possibility of integrating Fiddle with the Pajé visualizer. One of the basic needs to achieve such integration concerns the adaptation of Pajé to support on-line monitoring (as a complement of the actual *post-mortem* support). Such adaptation has not yet been taken by Pajé authors, but such integration work is in our plans, waiting for the appropriate *time window*;

ii) *Automatic debugging*. The existence of a stable debugging engine is a base for further experiments concerning debugging support. One of our goals is, inspired in our experiments on DEIPA$_2$, to work further on the support for automatic debugging, by developing a system which supports the verification of user defined annotations in the source code;

iii) *Fiddle integration in PSDE*. The successful experiments of integrating DDBG in two PSDE (GRADE and EDPEPPS) were very motivating, and we would like to work further on tool integration, specially on debugging support for PSDE. This is a quite open issue for us, as currently we have no schedule for such research work.

A

The Fiddle API

Contents

This Appendix makes a brief description of the API for all the currently implemented Fiddle layers: Layer0_s, Layer0_m, Layer1_m and Layer2_m.

A complete and detailed description of Fiddle API, with some examples, is available in Fiddle Users Manual [LC99].

A.1 Fiddle Utilities Library

When a client tool requests a service to Fiddle debugging engine, it will receive a reply in the form of a `tkout_t` structure. To keep this structure as simple as possible, we have decided to use a very simple, although thread-safe, generic *double linked list* (`chain_t`) as the basic data structure to hold sets of elements. The operations to manipulate such double linked lists are available in the additional `fiddleutil` library.

Additionally, this library also provides some warning and error message printing service, where each message is tagged with the Process ID, a very relevant information in a multi-threaded distributed program.

A.1.1 Double Linked List (`chain_t`)

The `fiddleutil` library implements an abstract data type and an API to manipulate a *double linked list*. This double linked list is reentrant (protected by *mutexes* and *condition variables* where needed) to allow concurrent accesses to the same list by different threads.

This library is used internally in the Fiddle libraries that are linked to the client tools and make use of services and data typed defined in the `pthreads` library. As such, the Fiddle client tools, which must be linked to Fiddle libraries, must also link to the `pthreads` library.

Data type

`chain_t`

Definition:

This is an opaque data type.

Description:

The type which represents the list.

Data type

`chain_node_t`

Definition:

This is an opaque data type.

Description:

The type for the list nodes. Keep track of the nodes inserted in the list.

Data type

`chain_collect_t`

Definition:

This is an opaque data type.

Description:

Applying a walker to a chain using `chain_walk_collect()` will return a new chain with nodes of this type. It contains a pair of type (A, B), where A is a pointer to the

original element in the chain, and *B* is a pointer to the value returned by the walker when it was applied to *A*.

Data type

chain_walk_f

Definition:

```
typedef int (*chain_walk_f) (void *data, void *args)
```

Description:

The type for *walker* functions. Walker functions will be applied by iterators to all or part of the elements in a chain. The argument *args* will be passed when calling the iterator, to be used at walker's will.

If *walker* returns a negative number the traversing stops immediately.

Data type

chain_walk_collect_f

Definition:

```
typedef int (*chain_walk_collect_f) (void *data, void *args)
```

Description:

A second type for *walker* functions, specific for being called from the *chain_walk_collect* iterator. The argument *args* will be passed when calling the iterator, to be used at walker's will.

If *walker* returns a negative number the traversing stops immediately.

Data type

chain_cmp_f

Definition:

```
typedef int (*chain_cmp_f) (void *node, void *what, void *args)
```

Description:

Some functions, such as *chain_find()*, traverse the chain searching for an element in *node* which compares successfully with *what*. The argument *args* will be passed when calling the iterator, to be used freely by the comparing funtion.

Should return `-1` if element in *node* is smaller (less than) *what*, `0` (zero) if they are equal, and *1* otherwise.

The extra argument will be passed when calling the iterator, and may be used or ignored by the comparing function.

Function

`chain_t` * **chain_create** (`void`)

Creates a new empty chain.

Returns a pointer to an opaque typed (`chain_t *`) chain, or `NULL` on failure.

Function

`void` **chain_destroy** (`chain_t` **chain*)

Deletes the *chain*, assuming tat all the data in the chain nodes was already cleaned

or is accessible from some other data structure.

Returns nothing.

Function

int **chain_length** (`chain_t` **chain*)

Returns the number of nodes in *chain*.

Function

`chain_node_t` * **chain_first** (`chain_t` **chain*)

Returns a pointer to the last node in *chain*, or `NULL` if the chain is empty.

Function

`chain_node_t` * **chain_next** (`chain_t` **chain*, `chain_node_t` **node*)

Returns a pointer to the node following *node* in *chain*, or `NULL` if *node* in the last one on the chain.

Function

`chain_node_t` * **chain_prev** (`chain_t` **chain*, `chain_node_t` **node*)

Returns a pointer to the node before *node* in *chain*, or `NULL` if *node* in the first one on the chain.

Function

`chain_node_t` * **chain_enqueue** (`chain_t` **chain*, `void` **data*)

Creates a new node to keep *data* and insert it at the tail of *chain*.

Returns a pointer to the created node, or `NULL` on failure.

Function

`void` * **chain_dequeue** (`chain_t` **chain*)

Removes (deletes) a node from *chain*. The data referenced by the node remains untouched, i.e., is not deleted.

Returns a pointer to the data in the (deleted) node, or `NULL` if the chain is empty.

Function

`void` * **chain_dequeue_cond** (`chain_t` **chain*, `chain_walk_f` *cond*, `void` **args*)

Blocks the calling thread until there is a node in *chain* for which the evaluation of *cond(args)* returns negative, then remove this node. The data kept in the node is not deleted.

Returns a pointer to the data in the (deleted) node, or `NULL` if the chain is empty.

Function

`void` * **chain_remove** (`chain_t` **chain*)

Removes (deletes) a node from *chain*. The data referenced by the node remains untouched, i.e., is not deleted.

Returns a pointer to the data in the (deleted) node, or `NULL` if the chain is empty.

Function

`chain_node_t` * **chain_prune** (`chain_t` **chain*, `chain_walk_f` *prunner*, `void` **args*)

Remove all nodes from *chain* for which *pruner(args)* returns positive, leaving the node untouched if *pruner(args)* returns `0` (zero). Stops the pruning if *pruner(args)* returns negative. If the data in the node was dynamically allocated, the *pruner()* may release that memory before returning.

Returns a pointer to the node where the prunning stopped, or `NULL` if all the chain was traversed.

Function

`chain_t` * **chain_merge** (`chain_t` **chain1*, `chain_t` **chain2*)

Append *chain2* to *chain1*.

Returns a pointer to *chain1*.

Function

`chain_node_t` * **chain_find** (`chain_t` **chain*, `chain_cmp_f` **compare*, `void` **args*)

Find the first node in *chain* for which *compare(args)* returns `0` (zero).

Returns a pointer to the node found or `NULL` if none was found.

Function

`chain_node_t` * **chain_walk** (`chain_t` **chain*, `chain_walk_f` **walker*, `void` **args*)

Function

`chain_node_t` * **chain_rwalk** (`chain_t` **chain*, `chain_walk_f` **walker*, `void` **args*)

Traverses *chain* from head to tail (*chain_walk()*) or from tail to head (*chain_rwalk()*), applying *walker(args)* to each node. Stops the traversing if *walker(args)* returns a negative value.

Returns a pointer to the node where the traversing stopped, i.e., to the last node to which *wkaler* was applied, or `NULL` if the all list was traversed.

Function

`chain_t` * **chain_walk_collect** (`chain_t` **chain*, `chain_walk_collect_f` **walker*, `void` **args*)

Traverses *chain* from head to tail applying *walker(args)* to each node, and collecting the return values of *walker(args)* into a new chain. Stops the traversing if *walker(args)* returns `NULL`.

Returns a pointer to the new created chain with the values returned by *walker(args)* (this new will have the same length as *chain*). Returns `NULL` on failure.

Function

`void` * **chain_collect_origin** (`const chain_walk_collect_f` **chain*)

Returns a pointer to the chain element to which a *walker* was applied.

Function

`void` * **chain_collect_data** (`const chain_walk_collect_f` **chain*)

Returns a pointer to the result of applying a walker to the chain element returned by *chain_collect_origin()*.

A.1.2 Warning or Fatal Error Message Display

Function

`int` * **msg_pid_write** (`const char` **fmt*, ...)

Executes a 'printf()', but write process identifier (PID) before.

Returns the same as printf().

Function

`int` * **msg_pid_fatal** (`const char` **fmt*, ...)

Similar to *msg_pid_write*, but also terminates the calling process.

Theoretically returns the same as *printf()*, but in practice this function will never return (the calling process will die).

A.2 Fiddle Layer 0_s Services

This Section describes the basic data types which the client tools must manipulate to access Fiddle services and to process the replies to the service requests. It also describes the API for the services provided by Fiddle Layer 0_s.

These services are available in the `fiddle0s` library.

A.2.1 Basic Data Types

In this and the following sections, it is assumed that the reader has good knowledge of the auxiliary data type(s) (specially the `chain_t` type) and associated management functions, defined in the `utils` library and described in Section A.1.

In the following we will use a top-down approach, i.e., the main data type (`tkout_t`) will be descried first, and then we will describe all of its components.

Data type

tkout_t

Definition:

```
1 typedef struct tkout_s {
2     int           tid;      /* The TID of the target process */
3     code_t        status;   /* The status of the request */
4     const char   *str;      /* The ourput of the Node Debugger */
5     data_t       *rep;      /* The reply (when successful) */
6     chain_t      *oob;      /* The out-of-band data (when successful) */
```

```
7     data_t      *pos;     /* The current position in the source code */
8     data_t      *err;     /* The error data (when unsuccessful) */
9 } tkout_t;
```

Description:

The base return type for Layer 0_s service functions.

Line 2) **int tid**
Contains the *Symbolic Task ID* of the target process;

Line 3) **const char *output**
Contains a full copy of the output message received from the node debugger;

Line 4) **code_t status**
Indicates the success or failure in the processing of the request according to the following rules:

> **status > FIDDLE_OK** — The request was not accepted due to, for example, an invalid set of arguments. In this case, *status* contains the error code, and the correspondig error message can be obtained by calling *f0s_err_msg()* with the *status* variable as argument;
>
> **status == FIDDLE_OK** — The request was accepted and sent to the node debugger. Once the node debugger sends the reply string, it will be parsed to extract the relevant data to fill the *rep*, *alw*, *pos* and *err* fields;

Line 5) **`data_t` *rep**
If the service request was processed successfully, this field contains the data extracted from the node debugger reply. Its detailed are explained below;

Line 6) **`chain_t` *oob**
If the service request was processed successfully, this field contains out-of-band data extracted from the node debugger reply.

If no out-of-band data was detected in the node debugger reply, this field will contain an empty list;

Line 7) **`data_t` *pos**
Contains the data relative to the current location in the source files, e.g., source file name and line number;

Line 8) **`data_t` *err**
Contains the data relative to the error messages generated by the node debugger.

If the service processing was successful, this field will point have the value `NULL`.

All Layer0_s service functions (except initialization and termination) return a pointer to such a `tkout_t` structure, whose contents will be filled according to the success/failure of the service requested and to the output received from the node debugger.

To fill the contents of the `tkout_t` structure, memory is allocated dynamically as needed by Fiddle. A service functions is provided for the user to release the memory used by the `tkout_t` structure when no longer needed.

Data type

code_t

Definition:

```
1  typedef enum code_e {
2  /* ################################################################
3   *       GENERIC Codes
4   * ################################################################ */
5      FIDDLE_OK = 0,                /* Success */
6      E_RTS_OK = FIDDLE_OK,         /* Success */
7
8  /* ################################################################
9   *       ERROR Codes
10  * ################################################################ */
11     /* Errors generated by the Parsifal-39 run-time system */
12     E_RTS_SEND_RECEIVE,
13     E_RTS_INVALID_TP_PID,
14     E_RTS_INVALID_ND_PID,
15     E_RTS_NO_CLD_AVAILABLE,
16     E_RTS_COMMAND_GENERATION,
17     E_RTS_INVALID_TID,
18     E_RTS_STAT_FILE,
19     E_RTS_FILE_PERM,
20     E_RTS_STAT_TTY,
21     E_RTS_TTY_PERM,
22     E_RTS_ONE_LINE_FUNCTION,
23     E_RTS_NO_COMMAND,
24     E_RTS_ASTR_CREATE,
25     E_RTS_INIT_REGEX,
26     E_RTS_INVALID_LINE_NUMBER,
27     E_RTS_INVALID_ARGUMENT,
28     E_RTS_INVALID_OPTION,
29     E_RTS_MEMORY_ERROR,
30     E_RTS_INVALID_SIGNUM,
31     E_RTS_SIGNAL,
32     E_RTS_INVALID_REQID,
33     E_RTS_CANNOT_START_DAEMON,
34     E_RTS_UNKNOWN_OUTPUT,
35
36     /* Errors generated by parsing the output */
37     E_UNKNOWN_CMD_CODE,
```

```
38      E_TOP,
39      E_NOT_RUN,
40      E_NO_FRAME_SELECTED,
41      E_NO_PROCESS,
42      E_NOT_PERMITTED,
43      E_NO_SYMBOL,
44      E_BOTTOM,
45      E_INVALID_EXPRESSION,
46      E_NO_SYMBOLS,
47      E_NO_SYMBOL_TABLE,
48      E_NO_DEF_SOURCE_FILE,
49      E_INVALID_BPID,
50      E_INVALID_LINE_NUMBER,
51      E_FUNCTION_NOT_DEFINED,
52      E_LINE_OUT_OF_RANGE,
53      E_NO_BOUNDS,
54      E_INVALID_SYMBOL,
55      E_INVALID_DISPLAY_NUMBER,
56      E_NO_DISPLAY_EXPRESSION,
57      E_INVALID_ARITHMETIC_OPERATION,
58      E_PARSE_ERROR,
59      E_TOO_FEW_ARGS,
60      E_NO_STACK,
61      E_NO_BREAK_OR_WATCH,
62      E_NO_LINE_NUM_INFO,
63      E_JUNK_AT_END_OF_LINE,
64      E_INVALID_THREAD_ID,
65      E_NO_THREAD_ID,
66      E_NO_THREAD_COMMAND,
67      E_SIGSEGV,
68      E_LAST,
69
70  /* ################################################################
71   *       REPLY Codes
72   * ################################################################ */
73      R_ATTACH,
74      R_BREAK,
75      R_BREAK_NO_SOURCE,
76      R_CALL_VOID,
77      R_CONTINUE,
78      R_DELETE,
79      R_DETACH,
80      R_EVALUATE,
81      R_INFO_LOCALS,
82      R_FILE,
83      R_FINISH,
84      R_INFO_BREAK,
85      R_INFO_BREAK_WATCH_DATA,
86      R_INFO_DISPLAY,
```

```
87      R_INFO_LINE,
88      R_INFO_LINE_NO_CODE,
89      R_INFO_PROGRAM,
90      R_INFO_STACK,
91      R_INFO_THREADS,
92      R_INFO_WATCHPOINT,
93      R_KILL,
94      R_LIST,
95      R_NEXT,
96      R_RUN,
97      R_SENDTO,
98      R_SET_VARIABLE,
99      R_STEP,
100     R_SYMBOL_FILE,
101     R_THREAD,
102     R_TIDS,
103     R_ADD_DEBUGGER,
104     R_TTY,
105     R_UNDISPLAY,
106     R_NEW_THREAD,                   /* For line processing */
107     R_CALL,                         /* Not used! */
108     R_EVALUATE_ONE_LINE,            /* For internal use */
109     R_EVALUATE_MANY_START,          /* For internal use */
110     R_EVALUATE_MANY_MIDDLE,         /* For internal use */
111     R_EVALUATE_MANY_END,            /* For internal use */
112     R_INFO_LOCALS_ONE_LINE,         /* For internal use */
113     R_INFO_LOCALS_MANY_START,       /* For internal use */
114     R_INFO_LOCALS_MANY_MIDDLE,      /* For internal use */
115     R_INFO_LOCALS_MANY_END,         /* For internal use */
116     R_DISPLAY_EXPRESSION,           /* For internal use */
117     R_DISPLAY_ONE_LINE,             /* For internal use */
118     R_DISPLAY_MANY_START,           /* For internal use */
119     R_DISPLAY_MANY_MIDDLE,          /* For internal use */
120     R_DISPLAY_MANY_END,             /* For internal use */
121     R_CLIENTS,                      /* Used just in level f2m */
122     R_REGISTER_CLIENT,              /* Used just in level f2m */
123
124 /* ################################################################
125  *       POSITION Codes
126  * ################################################################ */
127     P_POSITION,
128
129 /* ################################################################
130  *       OUT OF BAND Codes
131  * ################################################################ */
132     O_DISPLAY,
133     O_BPHIT,
134     O_SIGNAL,
135     O_NEW_THREAD,
```

```
136     O_SWITCH_TO_THREAD,
137 } code_t;
```

Description:

All service requests will have a reply from Fiddle. In the status field returns a success code (which depends from the service requested) or an error code.

This data type enumerates all the possible *status* values. Some entries have a comment "*internal*", meaning that the entry is used internally, and this value should never be returned to the user.

Data type

`data_t`

Definition:

```
1 typedef struct data_s {
2     code_t          code;     /* The success/error code */
3     const char     *str;      /* The relevant part of the output */
4     chain_t        *data;     /* The data extrated from the output */
5 } data_t;
```

Description:

Contains the data extracted from the reply of a successful service request.

Line 2) **`code_t` code**
The status code;

Line 3) **const char *str**
The substring of the reply received from the node debugger from where the data was extracted;

Line 4) **`chain_t` *data**
A list with the data extracted;

A.2.2 Management Services

Function

`int` * **f0s_initialize** (`void`)

Initializes the debugging engine.
Returns `0` zero on success, or a negative number on failure.

Function

`int` * **f0s_terminate** (`void`)

Shuts down the debugging engine.
Returns `0` zero on success, or a negative number on failure.

Function

`tkout_t` * **f0s_tids** (`const char` **options*)

Fiddle will assign a *Symbolic Task Identifier* to each target process. Currently, no options are available, so the *options* argument must be set to `NULL` or to the empty string (`""`).

Returns a `tkout` on success, or `NULL` on failure.

Function

`int` **f0s_tkout_delete** (`tkout_t` **tk*)

Releases the memory associated with the output token *tk*. The fields of *tk* which point to other dynamically allocated memory areas will also be released.

Returns `0` zero on success, or a negative number on failure.

Function

`const char` **f0s_err_msg** (`code_t` **errno*)

Returns a pointer to a string which described the error with code *errno*.

Returns a pointer to the string, or `NULL` on failure.

A.2.3 Process Control Services

Function

`tkout_t` **f0s_attach** (`int` **tp_pid*)

Launches a new node debugger and attach it to the process with PID *tp_pid*.

The process I/O will use the same terminal it was using before the attachment.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_detach** (`int` *tid*)

Detaches the debugging engine from the target process *tid*. The process is left to run uncontrolled as it was before the attachment, and the node debugger is terminated.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_kill** (`int` *tid*)

Kills the target process *tid*.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_symbol_file** (`int` *tid*, `const char` **file*)

Loads the symbols table for the target process *tid* from *file*.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_file** (`const char` **file*)

Loads the executable *file* into memory under control of a node debugger.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_run** (`int` *tid*)

Starts the execution of a program previously loaded with *f0s_file*.

The target process will stop if a breakpoint is hit, a signal is received or if program terminates.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_step** (`int` *tid*)

Continues the execution of the target process *tid* until next instruction (at source level) is reached. If the current instruction is a function call, the execution will **step into** the called functions.

The target process will stop at the next instruction, or before if a breakpoint is hit, if a signal is received or if the program terminates.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_next** (`int` *tid*)

Continues the execution of the target process *tid* until next instruction (at source level) is reached. If the current instruction is a function call, the execution will **step over** the called functions.

The target process will stop at the next instruction, or before if a breakpoint is hit, if a signal is received or if the program terminates.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_continue** (`int` *tid*)

Continues the execution of the target process *tid* until a breakpoint is hit, a signal is received or the program terminates.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_finish** (`int` *tid*)

Continues the execution of the target process *tid* until the end of the current function is reached, a breakpoint is hit, a signal is received or the program terminates.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_signal** (`int` *tid*, `int` *signo*)

Sends signal with code *signo* to the target process *tid*.

The information about the signal received will be available in the *oob* field of the returned `tkout_t`.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_break** (`int` *tid*, `const char` **file*, `int` *line*, `const char` **function*)

Sets a breakpoint in *line* or in *function* of process *tid*. The *line* and *function* arguments are exclusive, so, when requesting this service, either *line == -1* or *function == NULL* . If *file* is omitted (*file == NULL*) the current source file is considered as the default. If *function != NULL* and there is only one function with such name in the program, the file which containing the source of the function is selected automatically.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_delete** (`int` *tid*, `int` *bpid*)

Deletes breakpoint *bpid* in process *tid*.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_set_variable** (`int` *tid*, `const char` **variable*, `const char` **expression*)

Sets *variable* in *tid* to the result of evaluating *expression*.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_call** (`int` *tid*, `const char` **expression*)

Evaluates *expression* in the current context of *tid*, where *expression* is the name of a function with its arguments.

Returns a `tkout` on success, or `NULL` on failure.

A.2.4 Process Inspection Services

Function

`tkout_t` **f0s_evaluate** (`int` *tid*, `const char` **expression*)

Evaluates *expression* in the current context of *tid*.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_display** (`int` *tid*, `const char` **expression*)

Evaluates *expression* in the current context of *tid* every time the process execution stops.

The expression to be evaluated and its value will be available in the *oob* field of the returned `tkout_t`.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_undisplay** (`int` *tid*, `int` *dispid*)

Stops evaluating the expression with ID *dispid* every time the process *tid* stops.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_list** (`int` *tid*, `const char` **file*, `int` *start_line*, `const char` **function*, `int` *nlines*)

Lists *nlines* of *tid* source file, starting in *line* or in *function*. These two arguments are exclusive, so when calling this function, either *line == -1* or *function == `NULL`* . If *file* is omitted (by giving *file == NULL*) the current source file is considered as the default. If *function != `NULL`* and there is only one function with such name in the program, the file which contains the source of the function is selected automatically.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_info_program** (`int` *tid*)

Gets information about target process *tid*.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_info_stack** (`int` *tid*)

Gets the current *tid* stack frames.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_up** (`int` *tid*)

Sets the upper stack frame (closer to the *main* function) as the current one.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_down** (`int` *tid*)

Sets the lower stack frame as the current one.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_frame** (`int` *tid*, `int` *frameno*)

Sets stack frame *frameno* as the current one.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_info_display** (`int` *tid*)

Gets information about the current display expressions in *tid*.

Returns a `tkout` on success, or `NULL` on failure.

A.2.5 Thread-related Services

Function

`tkout_t` **f0s_thread** (`int` *tid*, `int` *threadid*)

Selects thread *threadid* of *tid* as the target for the next services requested.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_info_threads** (`int` *tid*)

Gets information about all the current threads in *tid*.

Returns a `tkout` on success, or `NULL` on failure.

A.2.6 Miscellaneous Services

Function

`tkout_t` **f0s_tty** (`int` *tid*, `const char` **device*)

Redirects the standard I/O channels of *tid* to *device*. This redirection does not apply if *tid* is already running, being valid only for the next re-execution of *tid*.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f0s_sendto** (`int` *tid*, `const char` **command*)

Sends a command directly to the node debugger associated with *tid*, and gets its plain reply as a string (not processed) .

This command may be used to access functionalities available in the node debugger which have no mapping in Fiddle services.

Returns a `tkout` on success, or `NULL` on failure.

A.3 Fiddle Layer 0_m Services

The API for the services provided by Fiddle Layer 0_m is identical to the API for Fiddle Layer 0_s. The basic data types are still the same. All the functions have the same name and arguments, except that the prefix *f0s* in the name must be replaced for *f0m*, e.g., Layer 0_s's *f0s_run(...)* was renamed to *f0m_run(...)* in Layer 0_m. The Fiddle Layer 0_m API is, therefore, listed with no further comments or explanations.

These services are available in the `fiddle0m` library.

A.3.1 Management Services

Function

`int *` **f0m_initialize** (`void`)

Function

`int *` **f0m_terminate** (`void`)

Function

`tkout_t` * **f0m_tids** (`const char` **options*)

Function

`int` **f0m_tkout_delete** (`tkout_t` **tk*)

Function

`const char` **f0m_err_msg** (`code_t` **errno*)

A.3.2 Process Control Services

Function

`tkout_t` **f0m_attach** (`int` **tp_pid*)

Function

`tkout_t` **f0m_detach** (`int` *tid*)

Function

`tkout_t` **f0m_kill** (`int` *tid*)

Function

`tkout_t` **f0m_symbol_file** (`int` *tid*, `const char` **file*)

Function

`tkout_t` **f0m_file** (`const char` **file*)

Function

`tkout_t` **f0m_run** (`int` *tid*)

Function

`tkout_t` **f0m_step** (`int` *tid*)

Function

`tkout_t` **f0m_next** (`int` *tid*)

Function

`tkout_t` **f0m_continue** (`int` *tid*)

Function

`tkout_t` **f0m_finish** (`int` *tid*)

Function

`tkout_t` **f0m_signal** (`int` *tid*, `int` *signo*)

Function

`tkout_t` **f0m_break** (`int` *tid*, `const char` **file*, `int` *line*, `const char` **function*)

Function

`tkout_t` **f0m_delete** (`int` *tid*, `int` *bpid*)

Function

`tkout_t` **f0m_set_variable** (`int` *tid*, `const char` **variable*, `const char` **expression*)

Function

`tkout_t` **f0m_call** (`int` *tid*, `const char` **expression*)

A.3.3 Process Inspection Services

Function

`tkout_t` **f0m_evaluate** (`int` *tid*, `const char` **expression*)

Function

`tkout_t` **f0m_display** (`int` *tid*, `const char` **expression*)

Function

`tkout_t` **f0m_undisplay** (`int` *tid*, `int` *dispid*)

Function

`tkout_t` **f0m_list** (`int` *tid*, `const char` **file*, `int` *start_line*, `const char` **function*, `int` *nlines*)

Function

`tkout_t` **f0m_info_program** (`int` *tid*)

Function

`tkout_t` **f0m_info_stack** (`int` *tid*)

Function

`tkout_t` **f0m_up** (`int` *tid*)

Function

`tkout_t` **f0m_down** (`int` *tid*)

Function

`tkout_t` **f0m_frame** (`int` *tid*, `int` *frameno*)

Function

`tkout_t` **f0m_info_display** (`int` *tid*)

A.3.4 Thread-related Services

Function

`tkout_t` **f0m_thread** (`int` *tid*, `int` *threadid*)

Function

`tkout_t` **f0m_info_threads** (`int` *tid*)

A.3.5 Miscellaneous Services

Function

`tkout_t` **f0m_tty** (`int` *tid*, `const char` **device*)

Function

`tkout_t` **f0m_sendto** (`int` *tid*, `const char` **command*)

A.4 Fiddle Layer 1_m Services

The API for the services provided by Fiddle Layer 1_m is very similar to the previous APIs for Layer 0_m and Layer 0_s. The basic data types are the same, and the majority of the functions have the same name and arguments, except that the prefix *f0m* in the name must be replaced for *f1m*, e.g., Layer 0_s's *f0m_run(...)* was renamed to *f1m_run(...)* in Layer 0_m.

The few functions that have minor changes in the number of arguments are described in detail. All the others are just listed with no further comments or explanations.

These services are available in the `fiddle1m` library.

A.4.1 Management Services

Function

`int *` **f1m_initialize** (`void`)

Function

`int *` **f1m_terminate** (`void`)

Function

`tkout_t` * **f1m_tids** (`const char` **options*)

Function

`int` **f1m_tkout_delete** (`tkout_t` **tk*)

Function

`const char` **f1m_err_msg** (`code_t` **errno*)

A.4.2 Process Control Services

Function

`tkout_t` **f1m_attach** (`int` **tp_pid*, `const char` **node*)

In remote machine *node*, launches a new node debugger and attach it to the process with PID *tp_pid*.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f1m_detach** (`int` *tid*)

Function

`tkout_t` **f1m_kill** (`int` *tid*)

Function

`tkout_t` **f1m_symbol_file** (`int` *tid*, `const char` **file*)

Function

`tkout_t` **f1m_file** (`const char` **node*, `const char` **file*)

In remote machine *node*, loads the executable *file* into memory under control of a node debugger.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f1m_run** (`int` *tid*)

Function

`tkout_t` **f1m_step** (`int` *tid*)

Function

`tkout_t` **f1m_next** (`int` *tid*)

Function

`tkout_t` **f1m_continue** (`int` *tid*)

Function

`tkout_t` **f1m_finish** (`int` *tid*)

Function

`tkout_t` **f1m_signal** (`int` *tid*, `int` *signo*)

Function

`tkout_t` **f1m_break** (`int` *tid*, `const char` **file*, `int` *line*, `const char` **function*)

Function

`tkout_t` **f1m_delete** (`int` *tid*, `int` *bpid*)

Function

`tkout_t` **f1m_set_variable** (`int` *tid*, `const char` **variable*, `const char` **expression*)

Function

`tkout_t` **f1m_call** (`int` *tid*, `const char` **expression*)

A.4.3 Process Inspection Services

Function

`tkout_t` **f1m_evaluate** (`int` *tid*, `const char` **expression*)

Function

`tkout_t` **f1m_display** (`int` *tid*, `const char` **expression*)

Function

`tkout_t` **f1m_undisplay** (`int` *tid*, `int` *dispid*)

Function

`tkout_t` **f1m_list** (`int` *tid*, `const char` **file*, `int` *start_line*, `const char` **function*, `int` *nlines*)

Function

`tkout_t` **f1m_info_program** (`int` *tid*)

Function

`tkout_t` **f1m_info_stack** (`int` *tid*)

Function

`tkout_t` **f1m_up** (`int` *tid*)

Function

`tkout_t` **f1m_down** (`int` *tid*)

Function

`tkout_t` **f1m_frame** (`int` *tid*, `int` *frameno*)

Function

`tkout_t` **f1m_info_display** (`int` *tid*)

A.4.4 Thread-related Services

Function

`tkout_t` **f1m_thread** (`int` *tid*, `int` *threadid*)

Function

`tkout_t` **f1m_info_threads** (`int` *tid*)

A.4.5 Miscellaneous Services

Function

`tkout_t` **f1m_tty** (`int` *tid*, `const char` **device*)

Function

`tkout_t` **f1m_sendto** (`int` *tid*, `const char` **command*)

A.5 Fiddle Layer 2_m Services

The API for the services provided by Fiddle Layer 2_m is identical to the API for Fiddle Layer 1_m. The basic data types are still the same. All the functions have the same name and arguments, except that the prefix *f1m* in the name must be replaced for *f2m*, e.g., Layer 0_s's *f1m_run(...)* was renamed to *f2m_run(...)* in Layer 0_m.

The Fiddle Layer 2_m API is, therefore, listed with no further comments or explanations.

These services are available in the `fiddle2m` library.

A.5.1 Management Services

Function

`int *` **f2m_initialize** (`void`)

Function

`int` * **f2m_terminate** (`void`)

Function

`tkout_t` * **f2m_tids** (`const char` **options*)

Function

`int` **f2m_tkout_delete** (`tkout_t` **tk*)

Function

`const char` **f2m_err_msg** (`code_t` **errno*)

A.5.2 Process Control Services

Function

`tkout_t` **f2m_attach** (`int` **tp_pid*, `const char` **node*)

In remote machine *node*, launches a new node debugger and attach it to the process with PID *tp_pid*.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f2m_detach** (`int` *tid*)

Function

`tkout_t` **f2m_kill** (`int` *tid*)

Function

`tkout_t` **f2m_symbol_file** (`int` *tid*, `const char` **file*)

Function

`tkout_t` **f2m_file** (`const char` **node*, `const char` **file*)

In remote machine *node*, loads the executable *file* into memory under control of a node debugger.

Returns a `tkout` on success, or `NULL` on failure.

Function

`tkout_t` **f2m_run** (`int` *tid*)

Function

`tkout_t` **f2m_step** (`int` *tid*)

Function

`tkout_t` **f2m_next** (`int` *tid*)

Function

`tkout_t` **f2m_continue** (`int` *tid*)

Function

`tkout_t` **f2m_finish** (`int` *tid*)

Function

`tkout_t` **f2m_signal** (`int` *tid*, `int` *signo*)

Function

`tkout_t` **f2m_break** (`int` *tid*, `const char` **file*, `int` *line*, `const char` **function*)

Function

`tkout_t` **f2m_delete** (`int` *tid*, `int` *bpid*)

Function

`tkout_t` **f2m_set_variable** (`int` *tid*, `const char` **variable*, `const char` **expression*)

Function

`tkout_t` **f2m_call** (`int` *tid*, `const char` **expression*)

A.5.3 Process Inspection Services

Function

`tkout_t` **f2m_evaluate** (`int` *tid*, `const char` **expression*)

Function

`tkout_t` **f2m_display** (`int` *tid*, `const char` **expression*)

Function

`tkout_t` **f2m_undisplay** (`int` *tid*, `int` *dispid*)

Function

`tkout_t` **f2m_list** (`int` *tid*, `const char` **file*, `int` *start_line*, `const char` **function*, `int` *nlines*)

Function

`tkout_t` **f2m_info_program** (`int` *tid*)

Function

`tkout_t` **f2m_info_stack** (`int` *tid*)

Function

`tkout_t` **f2m_up** (`int` *tid*)

Function

`tkout_t` **f2m_down** (`int` *tid*)

Function

`tkout_t` **f2m_frame** (`int` *tid*, `int` *frameno*)

Function

`tkout_t` **f2m_info_display** (`int` *tid*)

A.5.4 Thread-related Services

Function

`tkout_t` **f2m_thread** (`int` *tid*, `int` *threadid*)

Function

`tkout_t` **f2m_info_threads** (`int` *tid*)

A.5.5 Miscellaneous Services

Function

`tkout_t` **f2m_tty** (`int` *tid*, `const char` **device*)

Function

`tkout_t` **f2m_sendto** (`int` *tid*, `const char` **command*)

[This page was intentionally left blank]

Bibliography

[Abr96] J. R. Abrial. *The B Book, Assigning Programs to Meaning*. Cambridge University Press, 1996. ISBN: 0-521-49619-5.

[acm89] *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24 of *ACM SIGPLAN Notices*. ACM Press, January 1989.

[acm91] *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 26 of *ACM SIGPLAN Notices*. ACM Press, 1991.

[acm93] *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 28 of *ACM SIGPLAN Notices*. ACM Press, 1993.

[AG] Pallas Software AG. `http://www.pallas.com/e/products/vampir/index.htm`.

[Aud98] A. Audo. Integration of the DDBG Distributed Debugger within the EDPEPPS Toolset. Technical report, Centre for Parallel Computing, University of Westminster, London, June 1998. Final Year BEng Project Technical Report.

[Aug03] P. Augusto. Fgi – fiddle grphical interface. First degree diploma project, Departamento de Informática da Universidade Nova de Lisboa, Lisboa, Portugal, December 2003. (In portuguese).

[Bat88] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of ACM Workshop on Parallel and Distributed Debugging*, volume 24 of *ACM SIGPLAN Notices*, pages 11–22. ACM Press, January 1988.

[Bat95] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995. ISSN:0734-2071.

[BFR95] O. Babaoğlu, E. Fromentin, and M. Raynal. A unified framework for the specification and run-time detection of dynamic properties in distributed computations. Technical Report UBLCS-95-3, University of Bologna, Italy, June 1995.

[BH] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *To appear Journal of Supercomputing Applications and High Performance Computing*.

[BM93] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. J. Mullender, editor, *Distributed Systems*, chapter 4, pages 55–96. Addison-Wesley, 2^{nd} edition, 1993.

[BR94] Ö. Babaoglu and M. Raynal. Specification and verification of dynamic properties in distributed computations. Technical Report TR UBLCS-93-11, Laborathory for Computer Science, University of Bologna, Italy, May 1993, Revised May 1994.

[CG98] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[Cho78] T. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[CKW00] J. C. Cunha, P. Kacsuk, and S. Winter, editors. *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environment*. Nova Science Publishers, Inc., 2000.

[CL85] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[CLA96a] J. C. Cunha, J. Lourenço, and T. Antão. DDBG: A distributed debugger — user's guide. In *SEPP Project, Copernicus Programme, 5^{th} Progress Report*. University of Westminster, London, UK, September 1996.

[CLA96b] J. C. Cunha, J. Lourenço, and T. Antão. A debugging engine for a parallel and distributed environment. In KFKI Hungarian Academy of Sciences, editor, *Proceedings of the 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'96)*, pages 111–118, Misckolc, Hungary, October 1996.

[CLA99] J. C. Cunha, J. Lourenço, and T. Antão. An experiment in tool integration: the DDBG parallel and distributed debugger. *Euromicro Journal of Systems Architecture*, 45(11):897–907, 1999. Elsevier Science Press.

[CLD98] J. C. Cunha, J. Lourenço, and V. Duarte. Using DDBG to support testing and high-level debugging interfaces. *Computers and Artifitial Inteligence*, 17(5):429–439, 1998. Slovak Academic Press.

[CLD01a] J. C. Cunha, J. Lourenço, and V. Duarte. *Parallel Program Development for Cluster Computing: Methodologies, Tools and Integrated Environments*, chapter Debugging of Parallel and Distributed Programs, pages 97–129. Nova Science, 2001. ISBN: 1-56072-865-5.

[CLD01b] J. C. Cunha, J. Lourenço, and V. Duarte. *Parallel Program Development for Cluster Computing: Methodologies, Tools and Integrated Environments*, chapter The DDBG Distributed Debugger, pages 279–290. Nova Science, 2001. ISBN: 1-56072-865-5.

[CLV+98] J. C. Cunha, J. Lourenço, J. Vieira, B. Moscão, and D. Pereira. A framework to support parallel and distributed debugging. In *Proceedings of the International Conference on High-Performance Computing and Networking (HPCN'98)*, volume 1401 of *Lecture Notes on Computer Science*, pages 708–717, Amsterdam, The Netherlands, April 1998. Springer-Verlag.

[Clá03] A. Cláudio. *Modelo Conceptual para o Depuramento de Programas Distribuídos por Troca de Mensagens*. PhD thesis, Department of Informatics, University of Lisbon, July 2003. DI/FCUL TR-03-22.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 26 of *ACM SIGPLAN Notices*, pages 167–174. ACM Press, December 1991.

[Coo97] Sun Cooporation. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.htm%l`, May 1997.

[D+00] T. Delaitre et al. *Parallel Program Development for Cluster Computing: Methodologies, Tools and Integrated Environments*, chapter Tools of EDPEPPS, pages 357–365. Nova Science, 2000. ISBN: 1-56072-865-5.

[DDK00] D. Drótos, G. Dózsa, and P. Kacsuk. *Parallel Program Development for Cluster Computing: Methodologies, Tools and Integrated Environments*, chapter GRAPNEL to C Translation in the GRADE Environment, pages 249–263. Nova Science, 2000. ISBN: 1-56072-865-5.

[Dica] Hyper Dictionary. `http://www.hyperdictionary.com`.

[Dicb] Hyper Dictionary. `http://http://info.astrian.net/jargon/`.

[DLC01] V. Duarte, J. Lourenço, and J. C. Cunha. Supporting on-line distributed monitoring and debugging. *Journal of Parallel and Distributed Computing Practices, Special Issue on Monitoring Systems and Tool Interoperability*, 4(4), 2001.

[Dua04] V. Duarte. *Monitoring of Paralel and Distributed Programs*. PhD thesis, Departamento de Informática, Universidade Nova de Lisboa, Portugal, 2004. To appear.

[Duc95] M. Ducassé, editor. *AADEBUG'95, 2nd International Workshop on Automated and Algorithmic Debugging*, Saint Malo, France, May 1995. IRISA-CNRS.

[Duc97] M. Ducassé, editor. *AADEBUG'97, 3rd International Workshop on Automated and Algorithmic Debugging*, 1997.

[Duc01] M. Ducasse, editor. *AADEBUG'00, 4th International Workshop on Automated Debugging*, January 2001.

[Etn00] Etnus Inc., Framingham, MA. *TotalView User's Guide (v4.1)*, June 2000. http://www.etnus.com/.

[Fau03] Danny Faught. `http://www.testingfaqs.org/t-static.html`, April 2003.

[FCdK95] A. Fagot and J. Chassin-de Kergommeaux. Optimized execution replay mechanism for rpc-based parallel programming models. Technical report, LMC-IMAG, 1995.

[Fet88] J. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[For94] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[Fri93] P. Fritzson, editor. *AADEBUG'93, 1st International Workshop on Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1993.

[GBD+98] A. Geist, A. Beguelein, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM Users's Guide and Reference Manual*. Engineering Physics and Mathemathics Division. Oak Ridge Laboratory, 3.3.10. ornl/tm-12187 edition, 1998.

[GCMK96] V. Garg, C. Chase, J. R. Mitchell, and R. Kilgore. *Tools and Environments for Paralell and Distributed Systems*, chapter Efficient Detection of Unstable Global Conditions Based on Monotonic Channel Predicates, pages 195–226. Kluwer Academic Publishers, 1996.

[GD95a] P. Kacsuk G. Dozsa, T. Fadgyas. A graphical programming language for parallel programs. Hpcti progress report 1, KFKI-MSZKI, Research Institute for Measurement and Computing Techniques, April 1995.

[GD95b] P. Kacsuk G. Dozsa, T. Fadgyas. Software specification of the GRED program and related interfaces. Hpcti progress report 2, KFKI-MSZKI, Research Institute for Measurement and Computing Techniques, October 1995.

[GDB] The GNU debugger, version 5.0. http://sources.redhat.com/gdb.

[HF97] M. Heath and J. E. Finger. Paragraph: A tool for visualizing performance of parallel programs. The National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, available at http://www.ncsa.uiuc.edu/Apps/MCS/ParaGraph/manual/manual.html, 1997.

[Hoo96] 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.

[HPR93] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 28 of *ACM SIGPLAN Notices*, pages 32–42. ACM Press, December 1993.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[Jus96] G. Justo. PVMGraph: A Graphical Editor for the Design of PVM Programs. Technical report, Centre for Parallel Computing, University of Westminster, London, February 1996. EDPEPPS EPSRC Project (GR/K40468) D2.3.3, EDPEPPS/5.

[KCD+97] 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(13):1747–1770, 1997. Elsevier Science Press.

[KCMV00] P. Kacsuk, J. Chassin de Kergomeaux, É. Maillet, and J.-M. Vincent. *Parallel Program Development for Cluster Computing: Methodologies, Tools and Integrated Environments*, chapter The Tape/PVM Monitor and the PROVE Visualization Tool, pages 291–303. Nova Science, 2000. ISBN: 1-56072-865-5.

[KDL00] P. Kacsuk, G. Dózsa, and R Lovas. *Parallel Program Development for Cluster Computing: Methodologies, Tools and Integrated Environments*, chapter The

GRADE Graphical Parallel Programming Environment, pages 231–247. Nova Science, 2000. ISBN: 1-56072-865-5.

[KGV96] D. Kranzlmuller, S. Grabner, and J. Volkert. Debugging massively parallel programs with ATTEMPT. In H. Liddell, A. Colbrook, B. Hertzberge, and P. Sloot, editors, *High-Performance Computing and Networking (HPCN'96 Europe)*, volume 1067 of *Lecture Notes in Computer Science*, pages 798–804. Springer-Verlag, New York, 1996.

[KS00] J. C. Kergommeaux and B. O. Stein. Pajé: An extensible environment for visualizing multi-threaded programs executions. In *Proc. Euro-Par 2000*, volume 1900 of *LNCS*, pages 133–140. Springer, 2000.

[KW96] H. Krawczyk and B. Wiszniewski. Interactive testing tool for parallel programs. In I. Jelly, I. Gorton, and P. Crolll, editors, *Software Engineering for Parallel and Distributed Systems*, pages 98–109, London, UK, 1996. Chapman & Hal.

[KW99] H. Krawczyk and B. Wiszniewski. *Analysis and Testing of Distributed Software Applications*. John Wiley & Sons, November 1999. ISBN 0471978027.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

[LC98a] J. Lourenço and J. C. Cunha. The PDBG process-level debugger for parallel and distributed programs. In *Proceedings of the 2nd Sigmetrics Symposium on Parallel and Distributed Tools (SPDT'98)*, page 154, Portland, Oregon, EUA, August 1998. ACM Press. (Poster).

[LC98b] J. Lourenço and J. C. Cunha. Replaying distributed applications with RPVM. In *Proceedings of the 2nd Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'98)*, pages 121–126, Budapest, Hungary, October 1998. Report Series of the Institute of Applied Computer Science and Information Systems, University of Vienna.

[LC98c] J. Lourenço and J. C. Cunha. A thread-level distributed debugger. In Faculdade de Engenharia da Universidade do Porto, editor, *Proceedings of the 3rd International Conference on Vector and Parallel Processing (VecPar'98)*, pages 359–366, Porto, Portugal, April 1998.

[LC99] J. Lourenço and J. C. Cunha. *Flexible Interface for Distributed Debugging (Library and Engine): Reference Manual*. Departamento de Informática da Universidade Nova de Lisboa, Portugal, December 1999. Under development.

[LC01] J. Lourenço and J. C. Cunha. Fiddle: a flexible distributed debugging architecture. In V.N. Alexandrov, J.J. Dongarra, B.A. Juliano, R.S. Renner, and C.J.K. Tan, editors, *Proc. ICCS 2001, International Conference on Computational Science, Part II, Special Session on "Tools and Environments for Parallel and Distributed Programming"*, volume 2074 of *Lecture Notes on Computer Science*, pages 821–830, San Francisco, CA, USA, May 2001. Springer-Verlag. ISBN 3-540-42233-1.

[LCK+97] 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)*, pages 291–298, Budapeste, Hungary, September 1997. IEEE Computer Society Press.

[LCM03] J. Lourenço, J. C. Cunha, and V. Moreira. Control and debugging of distributed programs using Fiddle. In K. De Bosschere M. Ronsse, editor, *Proc. of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, pages 143–158, Ghent, Belgium, September 2003. `http://arxiv.org/abs/cs/0309049`.

[Lex] Lectric Law Library's Lexicon. `http://www.lectlaw.com`.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, C-36(4):471–482, 1987.

[LWSB97] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS – On-line Monitoring Interface Specification (Version 2.0). Technical Report TUM-I9733, SFB-Bericht Nr. 342/22/97 A, Technische Universität München, Munich, Germany, July 1997.

[Mai95] E. Maillet. *Issues in Performance Tracing with Tape-PVM*, 1995.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editors, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Amsterdam, 1989. Elsevier Science Publishers.

[Meg00] D. Megginson. SAX 2.0: The simple API for XML. `http://www.megginson.com/SAX/index.html`, May 2000.

[MHC94] B. P. Miller, J. K. Hollingsworth, and M. D. Callaghan. The Paradyn parallel performance tools and PVM. In Jack J. Dongarra and Bernard Tourancheau, editors, *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 201–210, Townsend, PA,

USA, May 1994. Society for Industrial and Applied Mathematics. ISBN:0-89871-343-9.

[Mor02] V. Moreira. Deipa e codificação xml para o formato xdr do fiddle. Technical report, Departamento de Informática, Universidade Nova de Lisboa, July 2002.

[Mor03] V. Moreira. Deipa e codificação xml para o formato xdr do fiddle. First degree diploma project, Departamento de Informática da Universidade Nova de Lisboa, Lisboa, Portugal, July 2003. (In portuguese).

[Mos88] Dale Mosby. PDBX: A source level debugger for parallel programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 325–327, Madison WI, May 1988. [Extended abstract].

[Net93] R. H. B. Netzer. Optimal trace and replay for debugging shared-memory parallel programs. In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 28 of *ACM SIGPLAN Notices*. ACM Press, 1993.

[Net94] R. H. B. Netzer. Trace size vs. parallelism in trace-and-replay debugging of shared-memory programs. *LNCS*, 768, 1994.

[RB03] Michiel Ronsse and Koen De Bosschere, editors. *AADEBUG'03, 5th International Workshop on Automated Debugging*, September 2003.

[RBC+03] Michiel Ronsse, Koen De Bosschere, Christiaens Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmüller. Record/replay for nondeterministic program executions. *Communications of the ACM*, 46(9):62–67, September 2003.

[RK98] M. A. Ronsse and D. A. Kranzlmuller. Rolt-MP: Replay of Lamport timestamps for message passing systems. In *Proceedings of Euromicro Workshop on Parallel and Distributed Processing*, pages 87–93, 1998.

[Roy70] W. W. Royce. Managing the development of large software systems: Concepts as techniques. In *Proc. IEEE WESTCON*, Los Angels, CA, 1970. Ch. 3.

[SNC00] D. Stringhini, P. Navaux, and J. Chassin de Kergommeaux. A selection mechanism to group processes in a parallel debugger. In *PDPTA'2000*, Monte Carlo Resort, Las Vegas, Nevada, USA, June 2000.

[SP93] Richard Stallman and Roland H. Pesch. *Debugging with GDB: the GNU source-level debugger*. Free Software Foundation, 4.09 for GDB version 4.9

edition, 1993. Previous edition published under title: The GDB manual. August 1993.

[Spi95] John Michael Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice-Hall, New York, 2^{nd} edition, 1995.

[Str02] D. Stringhini. *Depuração de Programas Paralelos: Projecto de uma Interface Intuitiva*. PhD thesis, Universide Federal do Rio Grande do Sul, Brasil, August 2002.

[TG93] A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In B. P. Miller and C. McDowell, editors, *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 28 of *ACM SIGPLAN Notices*, pages 21–31, New York, NY, USA, May 1993. ACM Press.

[W3C98] W3C. `http://www.w3.org/XML/`, February 1998.

[Wit88] L. Wittie. Debugging distributed C programs by real time replay. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, volume 24 of *ACM SIGPLAN Notices*, pages 57–67. ACM Press, January 1988.

[WK94] S. Winter and P. Kacsuk. Software engineering for parallel processing. In *Proc. of the 8^{th} Symp. on Microcomputers and Microprocessor Applications*, Budapest, Hungary, 1994.

[This page was intentionally left blank]