STUDIA INFORMATICA 2012
Volume 33 Number 1 (104)

Michał KOMOROWSKI
Warsaw University of Technology, Institute of Computer Science

ENHANCING AND EXTENDING THE INTELLITRACE DEBUGGING CAPABILITIES

Summary. IntelliTrace is a historic debugger for the .NET platform. It has many capabilities but also some serious limitations: it significantly affects the performance of applications or provides only basic methods for analysis of collected data. In this paper, IntelliTrace is examined and a set of tools called IntelliTrace Toolkit, which allows these problems to be overcome, is proposed.

Keywords: debuggers, historic debuggers, IntelliTrace

WSPOMAGANIE I ROZSZERZANIE FUNKCJONALNOŚCI DEBUGOWANIA INTELLITRACE

Streszczenie. IntelliTrace to debuger historyczny dla platformy .NET. Ma wiele możliwości, ale również kilka ograniczeń, na przykład wpływa negatywnie na wydajności aplikacji i udostępnia tylko podstawowe sposoby analizowania zgromadzonych danych. W tym artykule możliwości IntelliTrace zostały poddane analizie i na tej podstawie zaproponowano zestaw narzędzi IntelliTrace Toolkit, który rozwiązuje wspomniane problemy.

Słowa kluczowe: debugery, debugery historyczne, IntelliTrace

1. Introduction

Every year companies produce millions of lines of code that unfortunately contain many bugs. In the United States, according to [1], "annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion"'. In other countries, the situation is probably similar. In consequence, developers need efficient debuggers in order to be able to cope with software bugs .

Nowadays, integrated development environments like Eclipse [7], NetBeans [13], Visual Studio [18] provide programmers with sophisticated tools of this kind which make it easier to detect, localize and fix bugs. However, most of these debuggers have limited insight in the history of a program's execution. This fact compels developers to run the application many times in order to find a bug and fix it. One of the debuggers that is able to go back in the history of a program's execution is part of Visual Studio 2010 and is known as IntelliTrace. IntelliTrace is dedicated for the .NET platform and it is the only tool of this kind intended for this technology. It has many capabilities but also some limitations: it significantly affects the performance of monitored applications, provides only basic methods for analysis of collected data and cannot be used in production environments without installing Visual Studio 2010.

Having analyzed IntelliTrace, the author has proposed two new tools called *Events Manager* and *IntelliTrace Analyzer*. *Events Manager* facilitates the use of IntelliTrace within a production environment and monitoring applications with minimal overhead. *IntelliTrace Analyzer* is a tool that can load collected data into a dedicated database designed with the aim of data mining. These tools have been developed and tested.

At the beginning of this paper, in section 2, historic debuggers are described. Then, in section 3 IntelliTrace is introduced. The description of *IntelliTrace Toolkit* can be found in section 4. Finally, the experiments are described in section 5. The last section 6 contains the summary.

2. Historic debuggers

Historic debuggers also known as time travelling, reversible, post-mortem or omniscient debuggers are not a new idea. The concept of moving back and forward in the history of a program execution was already described in the 60s [9]. However, despite their many advantages, historic debuggers are not widely used.

They allow one to save a lot of time and money needed to fix bugs by reducing the number of restarts that are required to debug an application. They also make it possible to quickly reproduce rare problems or problems which only occur on the client side, but not in the development environment. A historic debugger can also be used by developers to better understand how complex programs or algorithms work.

There are two main approaches used by historic debuggers in order to provide insight into the history of a program's execution. The first one is to log methods calls and program state changes to some kind of storage (for instance a file) during the program's execution. This approach is used by [2], RDXP (*Reversible Debugger for Cross Platform*) [3], TOD (*trace-oriented debugger*) [4], Chronon [5], IntelliTrace [11], ODB (*Omniscient Debugger*) [14] or

[15]. The second one, which is less common, is to generate the reverse version of the program code [8]. In this approach moving back in the history of the program's execution is performed not by reading the recorded state of the program from the log, but by executing the reverse code.

Historic debuggers can be also classified based on how they monitor applications and collect information. It is achieved either by using the static instrumentation [4, 5, 11, 14, 15], or by running the program on a special virtual machine [2, 3]. In case of the static instrumentation, instructions responsible for collecting information about a program execution are inserted into program's code before executing it. In the second approach a virtual machine is responsible for this.

Finally, some debuggers allow only collected data to be browsed, like changes to the value of some variable at different points of time [4, 5, 11, 14], while the others allow also the program's execution to be resumed [2, 3, 8, 15].

Another group of tools are those that monitor the execution of programs in order to detect so called invariants, for example: a variable A takes on values from the range 1 to 4. This knowledge allows abnormal situations to be detected. Some examples of this type of tools include Daikon [6], Perracotta [16], DIDUCE [17].

3. IntelliTrace

The IntelliTrace [11] debugger was introduced in Visual Studio 2010 Ultimate Edition and is dedicated for managed single/multi- threaded desktop, web or cloud applications based on the .NET platform. It uses a logging approach together with the instrumentation of a program's code and is not able to resume the execution of a program.

IntelliTrace operates as a separate process (stared from within Visual Studio) that attaches to an application and modifies its CIL (*Common Intermediate Language*) by injecting special instructions that are responsible for recording data. Collected data are stored in log files with the *iTrace* extension. These logs can be examined in Visual Studio 2010 at any time. IntelliTrace operates in two modes: basic and extended, that differ in performance and the range of information that is collected.

In the first mode, IntelliTrace monitors only so called diagnostic events. A diagnostic event is an important point in the execution of an application, for instance establishing connection to a database, executing a query or throwing an exception. Except for events that are recorded when an exception is thrown, every other event is related to some method. The range of information that is recorded for a particular event depends on the definition of this event. The number of available events types is limited to about one hundred and fifty. In this

mode, IntelliTrace collects a limited amount of information about program's execution, but on the other hand, it does not affect the performance of the monitored application so much.

In the extended mode, IntelliTrace collects information about the execution of every method or constructor. The values of primitive parameters passed on to a method or results returned by the methods are also monitored. In the case of non-primitive parameters and returned results, only the values of primitive fields are recorded. This mode allows one to investigate an application more thoroughly but at the same time, the log with collected information can be very big (of several gigabytes or even more). In this mode, IntelliTrace affects seriously the performance of the monitored application (for details see section 5). It is possible to instruct IntelliTrace to monitor only some assemblies, but the impact on the application still can be serious.

Inside the *iTrace* logs there is no difference between the both modes, except for the amount of collected data. Both diagnostic events, exceptions or method calls are represented in a similar way, by different types of low level events (it should be noted that a diagnostic event is something different than a low level event). For instance a method call is represented by the pair of low level events *MethodEnterEvent* and *MethodExitEvent*, while diagnostic events are represented by a low level event called *DiagnosticEvent*.

There are two possible ways of browsing IntelliTrace logs inside Visual Studio 2010. *IntelliTrace Events View* shows list of recorded events, while *Calls View* shows a call tree. For every executed and recorded method it also shows the values of actual parameters. If the user clicks on an event or on a method, the proper place in the code will be shown. IntelliTrace provides also *IntelliTrace API* [12] which allows one to programmatically analyze the *iTrace* logs.

4. IntelliTrace Toolkit

IntelliTrace Toolkit is a set of two applications (*Events Manger* and *IntelliTrace Analyzer*) designed and implemented by the author. These applications use IntelliTrace and *IntelliTrace API*, but provide a more convenient interface and more capabilities than the original technology does.

4.1. EventsManger

The first important problem with IntelliTrace is that it can seriously affect the performance of a monitored application, especially in the advanced mode. In the case of large applications (many assemblies with many types and methods), it can be even impractical to

use IntelliTrace. The simple mode is a partial solution to this problem, because the set of available events is limited.

Secondly, IntelliTrace cannot be used in production environments because it can only be deployed together with Visual Studio 2010. In consequence, it is not possible to run IntelliTrace on the client side without installing Visual Studio 2010, in order to record bugs which are difficult to reproduce in the development environment.

Both of these problems can be overcome by using *EventsManager*. Firstly, *EventsManager* allows one to modify the *CollectionPlan.xml* file. This file contains the definitions of diagnostic events and can be found in the Visual Studio 2010 installation directory. It is an XML file with a quite complicated structure. Manual modification of this file is possible, but this is not easy and is error prone. With the *EventsManager* events can be easily managed (created, modified, updated), so the developer is not limited to the default set of 150 events types.

IntelliTrace operates as a separated process. This means that there must exists an executable file which can be run. This file is called *IntelliTrace.exe* and, similarly to *CollectionPlan.xml*, can be found in the Visual Studio 2010 installation directory. *EventsManager* uses this executable to run IntelliTrace beyond the control of Visual Studio 2010 which makes it possible to take advantage of IntelliTrace in production environments.

EventsManager is implemented in the C# language and uses the .NET platform in version 4.0. It is a desktop application with a graphical user interface implemented in WPF (*Windows Presentation Foundation*).

The user interface consists of two main parts. On the left-hand side there is a panel with the configuration parameters of an experiment. It allows one to choose an application to monitor, choose the simple or the advanced mode of IntelliTrace or to define the output directory. Every configuration can be saved for future use.

The central part of the application contains a list with events, which can be grouped into categories. *EventsManager* uses also a reflection mechanism to retrieve a list of types and methods from the assemblies used by monitored applications. This makes the application more user friendly. In order to define an event, the user does not have to know the precise signature of a method, because he or she can choose it from a combo box.

It is also possible to define a so called *short description* and *long description* for an event. A description can be a static text, or it can contain the values of actual parameters or a result returned by a method. For instance, in order to refer to the value of the first argument, the following syntax should be used: {0}. Descriptions of recorded events are stored in the *iTrace* logs.

4.2. IntelliTrace Analyzer

Another major disadvantage of Visual Studio 2010 is that it does not provide more sophisticated methods for the analysis of the collected data, for instance, a statistical analysis. At the same time, the IntelliTrace log file is a binary file, so it is impossible to browse it without a dedicated application or *IntelliTrace API*. If the content of the *iTrace* file is loaded into the database, it will be much easier to analyze it in a different way, for instance, by using data mining algorithms to detect execution invariants.

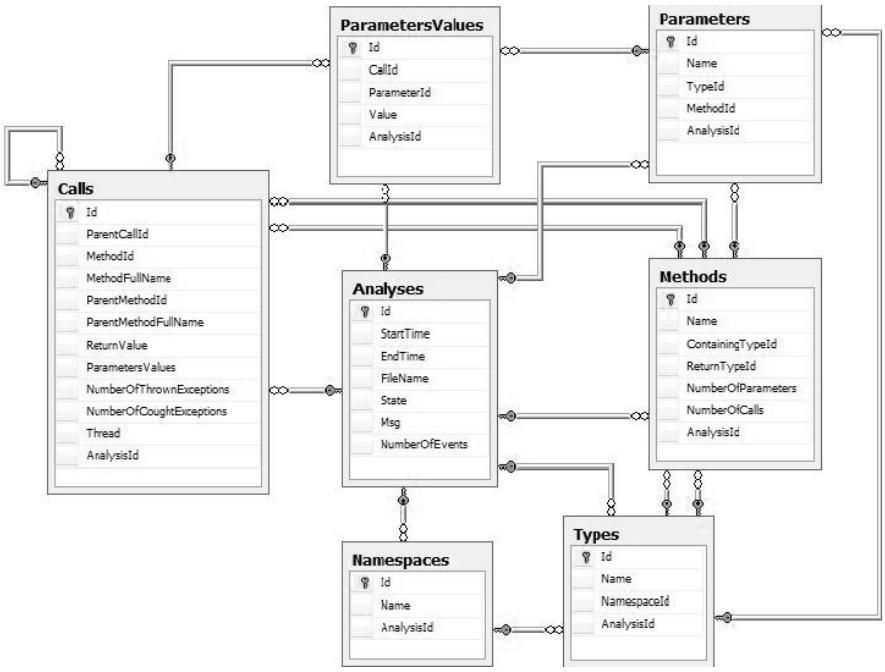

Fig. 1. Database schema diagram
Rys. 1. Schemat bazy danych

This is possible thanks to *IntelliTrace Analyzer*. This application uses *IntelliTrace API* in order to read information about the calls of methods from the *iTrace* files and loads data into the dedicated database. Later on in this article, I will use the term *IntelliTrace Database* for describing a database that is able to store information from *iTrace* logs. The schema diagram of this database is shown in Figure 1.

The central table is called *Analysis* and contains information about the log files that have been loaded into the database. The four following tables: *Namespaces*, *Types*, *Methods* and *Parameters* are used to store information about the structure of monitored applications. The values of parameters that have been passed on to methods can be found in the *ParametersValues* table.

The most valuable pieces of information are stored in the *Calls* table. Every row in this table corresponds to the execution of some method. The table contains cyclic references to itself, which allow a call tree to be recreated. The other information that can be read from the *Calls* table are: the number of exceptions thrown/caught during the execution of a method,

the string representation of a value returned by a method, and the identifier of the thread in which the method was executed. *IntelliTrace Database* is designed with the aim of data mining, so this table stores redundant information that can be found in other tables: the name of a method, the name of a parent method, or the string representation of the values of parameters.

Each of the above-mentioned tables has an additional field called *Analysis* which is redundant, but allows data to be easily filtered by the source log file without having to use complicated join operations. In order to conveniently browse information in the *IntelliTrace Database*, three views have been prepared:

- *TypesView* allows one to browse the full names of types (the name of a type + the name of a namespace);
- *ParametersView* allows one to browse parameters together with their types;
- *MethodsView* allows one to browse the full signatures of methods.

IntelliTrace Analyzer is a console application. Currently, it loads extracted data into the selected instance of *SQL Server 2008*, but another database management system can also be used. *IntelliTrace Analyzer* is implemented in the C# language and uses the .NET platform in version 4.0.

5. Experiments

In order to verify the developed tools, four functionally completely different applications were used. The first one is a console application that check if there is an Euler path in a graph (356 lines of the C# code). The second one is an application developed by the author, called *LanguageTrainer*, that is used to maintain a set of words in foreign languages and help the users in repeating and learning of these words (2825 lines of the C# code). *Farseer Physics Engine* [10] is a collision detection system with realistic physics responses (38 486 lines of the C# code). Finally, a commercial transfer agent platform was used (126 478 lines of the VB.NET code). The diversity of selected programs allowed the author to test *IntelliTrace Toolkit* thoroughly. All experiments were performed on a machine with 4 GB of memory and an Intel Core 2 Duo 2.53 GHz CPU processor .

The first experiments involved preparing, in *EventsManager*, different sets of events, for example: loading the configuration of an application, finding a word, showing a dialog box. Then the applications were run by *EventsManager* under the control of IntelliTrace configured in the simple mode, and the following checks were made: if all of the defined events were recorded, and if the created log could be opened by Visual Studio 2010 and did not contain any errors. The applications were also monitored in the advanced mode, and the

results were loaded into *IntelliTrace Database* by *IntelliTrace Analyzer*, and were also verified.

Table 1

Results of monitoring of the console application

Mode	Events	Log size	Calculation time	Slowdown factor
Without IntelliTrace	-	-	25 ms	1
Simple 1	105	5 MB	111 ms	4.44
Simple 2	113148	28 MB	2419 ms	96.76
Advanced	1003266	0.34 GB	7431 ms	297.24
Simple 1 + Advanced	1003371	0.34 GB	7705 ms	308.2
Simple 2 + Advanced	1116414	0.39 GB	9132 ms	365.28

The Table 1 shows the results of tests conducted based on the application that checks if there is an Euler path in the graph. The input graph had about 100 thousand nodes. The first column shows the configurations of IntelliTrace. *Simple 1* means that IntelliTrace was configured in the simple mode with monitoring of five rarely used methods (the five types of diagnostic events). While *Simple 2* means that diagnostic events were defined for frequently used methods. The second columns shows the number of registered low-level events. The *Log size* column contains the size of produced logs. The next column shows how much time does it take to finish the computations.

The last column shows the value of the slowdown factor calculated by dividing the computation time of the application running under the control of IntelliTrace, by the computation time without IntelliTrace. The results varied from about 5 for the simple mode to more than 300 for the advance mode + the simple mode. This indicates that it is very important to be able to define the custom events for the simple mode in order to work efficiently with IntelliTrace. However, it should be kept in mind that if we choose a frequently called method to monitor, the slowdown factor for the simple mode can be also high.

This experiment showed also that the simple and the advance modes are additive. If the both modes were enabled, the number of collected events was equal to the sum of events collected for each mode separately. The computation time was also correspondingly higher, similarly as the size of logs.

Table 2

Results of monitoring of the application with GUI

Mode	Events	Log size	Frames per second	Slowdown factor
Without IntelliTrace	-	-	60	1
Simple with 12 custom diagnostic events	9975	15 MB	60	1
Advanced	88 million	2.5 GB	3	20

In the next experiment, *Farseer Physics Engine* was used. The aim of this experiment was to check how IntelliTrace affects the performance of an application that requires the smooth display of animations. The author set himself the task of finding in which order and how often the methods of the *Screen Manager* class, that are responsible for refreshing the screen, are called. The author defined a diagnostic event for each of the 13 methods in this class. Then, a demo application using *Farseer Physics Engine* was run under the control of IntelliTrace operating in the advance mode and in the simple mode with custom events.

The results are shown in Table 2, which is built in a similar way as Table 1. In this case, the slowdown factor was calculated by dividing the number of *FPS* (Frames per second) as displayed by the application running under the control of IntelliTrace, by the number of *FPS* as displayed without IntelliTrace.

According to the results, for the simple mode it was possible to work normally with the application. The average number of *FPS* was the same as for the application running without IntelliTrace. In the case of the advance mode, it was impossible to work with the application because of the very low number of displayed *FPS.* IntelliTrace generated also a very big log file and collected a large number of events. Such an amount of data could allow one to analyze the execution of the application more thoroughly, however, working with such an amount of data is difficult, especially if we only need a small part of it.

Table 3

Performance of IntelliTrace Analyzer

Program	Log Size	Types	Methods	Param.	Calls	Events	Time(s)	Events/s
LanguageTrainer	0.25 GB	96	290	178	2174512	6.6 million	1224	4339+
Physics Engine	70 MB	129	440	446	450862	2.3 million	370	6334
Transfer Agent	75 MB	329	1792	1241	239710	1.2 million	267	4437

The table 3 shows the results of the performance tests of *IntelliTrace Analyzer*. The three columns: *Types*, *Methods*, *Param* contain information about the structure of a program, i.e. the number of types (classes), the number of distinct methods, and the total number of the parameters of methods that were found in the log. The number of recorded method calls and low-level events can be found in the columns: *Calls* and *Events*. The average time (in seconds) of loading data into *IntelliTrace Database* is given in the column *Time*. The last column contains the average number of low-level events processed per second.

The experiments have shown that *IntelliTrace Analyzer* is able to load from about 4000 to about 6000 low-level events per second. The reason for this is difficult to establish. If we compare *Farseer Physics Engine* and *Transfer Agent*, it seems that the results depend on how complex an application is. The more types and methods are used by an application, the smaller number of low-level events can be processed per second. On the other hand, the log for *LanguageTrainer* contains a considerably smaller number of distinct methods, but the

average speed of loading events was the worst. The author thinks that it could be dependent on how *IntelliTrace API* operates internally and how the data are stored in the *iTrace* files.

The performed experiments revealed also more limitations of IntelliTrace. Two of them have already been elevated by *IntelliTrace Toolkit*:

- IntelliTrace diagnostic events do not support interfaces. The support for abstract methods is also limited. For instance, it is not possible to define an event for an abstract method definition. However, an event can be defined for the implementation of this method in a subclass derived from an abstract class. *EventsManager* takes this into account and does not allow the definition of events that will not operate properly;
- It is not possible to refer, within the definition of an event, to the values of arguments and to the return value, at the same time. Trying to do this will cause an error. In order to solve this problem, *EventsManager* allows the user to choose which values are interesting for him in the context of a particular diagnostic event.

The following identified limitations have not been overcome yet and should be taken into account when planning experiments:

- IntelliTrace does not record the execution time of methods;
- It is only possible to refer within the definition of an event to the values of primitive types and *String*;
- It is not possible to determine the value of *out* parameters (it compels the called method to initialize it) or *ref* parameters (it allows the called method to change the object referenced by a parameter). It is always undefined. The author did not manage to determine the reason for this behavior. It might be due to a bug in IntelliTrace or *IntelliTrace API*;
- For some methods, it is not possible to extract the values of parameters from the log. If we make an attempt to do this, an exception is thrown. The author did not manage to determine the reason for this error. This is probably due to a bug in *IntelliTrace API*;
- The values of arguments passed on to the constructors are not recorded.

6. Summary

In this paper, the capabilities and limitations of IntelliTrace were investigated. In order to overcome these limitations, two new applications, namely *EventsManager* and *IntelliTrace Analyzer*, were proposed and developed. Experiments confirmed the efficiency of these tools.

Nonetheless, more work is needed regarding *IntelliTrace Toolkit*. Firstly, the *EventsManager* and *IntelliTrace Analyzer* applications should be fully integrated in order to provide a comfortable environment to work with IntelliTrace. Secondly, although *Events-Manager* allows one to define the descriptions for events, the user has to know the proper

syntax. This is not very convenient and should be improved. Moreover, *EventsManager* should support more complex scenarios of working with events. For instance, the possibility to refer (within the definition of an event) to the field or property of an object for which the method was called, or allowing the users to provide their own code that will be executed when an event is recorded (so called *Data Queries*).

More work is also needed regarding the newly detected limitations of IntelliTrace. For example, it is a quite serious problem that IntelliTrace does not record the execution time of methods. It should be checked to see, if this can be overcome by preparing some special types of diagnostic events.

It is also necessary to carry out more work regarding the analysis of collected data. In particular, the performance of *IntelliTrace Analyzer* should be improved and *IntelliTrace Database* should be extended to store not only a call tree but also diagnostic events. It will be also valuable to test the data mining algorithms that are provided by *SQL Server Analysis Services* in order to detect some interesting dependencies and the application execution invariant. The association rules or clustering could be useful in this case.

BIBLIOGRAPHY

1. Gallaher M. P., Kropp, B. M.: Economic Impacts of Inadequate Infrastructure for Software Testing. 2002.
2. Koju T., Takada S., Doi N.: An efficient and generic reversible debugger using the virtual machine based approach. 1st ACM/USENIX international conference on Virtual execution environments, 2005, p. 79÷88.
3. Wang L., Liu X., Song A., Xu L., Liu T.: An Effective Reversible Debugger of Cross Platform Based on Virtualization. International Conference on Embedded Software and Systems, 2009, p. 448÷453.
4. Pothier G., Tanter E.: Back to the Future: Omniscient Debugging. Software, IEEE, 2009, p. 78÷85.
5. Chronon: http://www.chrononsystems.com.
6. Daikon: http://groups.csail.mit.edu/pag/daikon.
7. Eclispe: http://www.eclipse.org.
8. Lee J.: Dynamic Reverse Code Generation for Backward Execution. In: Proceedings of the Workshop on Verification and Debugging, Vol. 174, 2006, p. 37÷54.
9. Balzer R. M.: EXDAMS: extendable debugging and monitoring system. Proceedings of the May 14-16, 1969, spring joint computer conference, 1969, p. 567÷580.
10. Farseer Physics Engine: http://farseerphysics.codeplex.com.
11. IntelliTrace: http://msdn.microsoft.com/en-us/library/dd264915.aspx.

12. IntelliTrace API: http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.intellitrace.aspx.
13. NetBeans: http://netbeans.org.
14. Omniscient Debugging, http://www.lambdacs.com/debugger/debugger.html.
15. Chen S., Fuchs W. K., Chung J.: Reversible Debugging Using Program Instrumentation. IEEE Transactions on Software Engineering, Vol. 27, 2001, p. 715÷727.
16. Perracotta: http://www.cs.virginia.edu/perracotta/.
17. Hangal S., Lam M. S.: Tracking Down Software Bugs Using Automatic Anomaly Detection. 24rd International Conference on Software Engineering, 2002, p. 291÷301.
18. Visual Studio 2010: http://www.microsoft.com/visualstudio/en-us.

Wpłynęło do Redakcji 2 grudnia 2011 r.

Omówienie

Debuger historyczny to narzędzie, które pozwala cofnąć się w historii wykonania programu. Dzięki temu można sprawdzić, kiedy i gdzie została wywołana dana metoda albo jak zmieniał się stan jakiegoś obiektu w czasie działania programu. Takie podejście pozwala ograniczyć liczbę ponownych uruchomień aplikacji potrzebnych do odtworzenia i zlokalizowania błędu, ułatwia również zrozumienie działania programu lub algorytmu.

Debugery historyczne są przeważnie projektowane z myślą o konkretnej technologii, na przykład dla platformy Java lub kodu natywnego. IntelliTrace to debuger historyczny, który został wprowadzony w Visual Studio 2010 i jest dedykowany dla platformy .NET. Jest to jedyne narzędzie tego rodzaju dla tej technologii. Ma wiele możliwości, ale również kilka istotnych ograniczeń, na przykład wpływa negatywnie na wydajności monitorowanych aplikacji i udostępnia tylko podstawowe sposoby analizowania zgromadzonych danych.

W tym artykule możliwości IntelliTrace zostały poddane gruntownej analizie i na tej podstawie zaproponowano nowy zestaw narzędzi *IntelliTrace Toolkit*, który rozwiązuje wspomniane problemy.

Address

Michał KOMOROWSKI: Warsaw University of Technology, Institute of Computer Science, ul. Nowowiejska 15/19, 00-665 Warszawa, Polska,
M.Komorowski@ii.pw.edu.pl/michalkomorowski@tlen.pl.