.NET Technologies 2006

University of West Bohemia
Campus Bory

May 29 – June 1, 2006

Short communication papers proceedings

Edited by

Jens Knoop, Vienna University of Technology, Austria
Vaclav Skala, University of West Bohemia, Czech Republic

.NET Technologies – Short communication papers conference proceedings

Editor-in-Chief: Vaclav Skala
University of West Bohemia, Univerzitni 8, Box 314
306 14 Plzen
Czech Republic
skala@kiv.zcu.cz
Managing Editor: Vaclav Skala

Author Service Department & Distribution:
Vaclav Skala - UNION Agency
Na Mazinach 9
322 00 Plzen
Czech Republic
Reg.No. (ICO) 416 82 459

Hardcopy: ***ISBN 80-86943-11-9***

CONFERENCE CO-CHAIR

Knoop, Jens (Vienna University of Technology, Vienna, Austria)
Skala, Vaclav (University of West Bohemia, Plzen, Czech Republic)

PROGRAMME COMMITTEE

Aksit, Mehmet (University of Twente, The Netherlands)
Giuseppe, Attardi (University of Pisa, Italy)
Gough, John (Queensland University of Technology, Australia)
Huisman, Marieke (INRIA Sophia Antipolis, France)
Knoop, Jens (Vienna University of Technology, Austria)
Lengauer, Christian (University of Passau, Germany)
Lewis, Brian,T. (Intel Corp., USA)
Meijer, Erik (Microsoft, USA)
Ortin, Francisco (University of Oviedo, Spain)
Safonov, Vladimir (St. Petersburg University, Russia)
Scholz, Bernhard (The University of Sydney, Australia)
Siegemund, Frank (European Microsoft Innovation Center, Germany)
Skala, Vaclav (University of West Bohemia, Czech Republic)
Srisa-an, Witawas (University of Nebraska-Lincoln, USA)
Sturm, Peter (University of Trier, Germany)
Sullivan, Kevin (University of Virginia, USA)
van den Brand, Mark (Technical University of Eindhoven, The Netherlands)
Veiga, Luis (INESC-ID, Portugal)
Watkins, Damien (Microsoft Research, U.K.)

REVIEWING BOARD

Alvarez, Dario (Spain)
Attardi, Giuseppe (Italy)
Baer, Philipp (Germany)
Bilicki, Vilmos (Hungary)
Bishop, Judith (South Africa)
Buckley, Alex (U.K.)
Burgstaller,Bernd (Australia)
Cisternino, Antonio (Italy)
Colombo, Diego (Italy)
Comito, Carmela (Italy)
Ertl, Anton,M. (Austria)
Faber, Peter (Germany)
Geihs, Kurt (Germany)
Gough, John (Australia)
Groesslinger, Armin (Germany)
Huisman, Marieke (France)
Knoop, Jens (Austria)
Kratz, Hans (Germany)
Kumar,C., Sujit (India)
Latour, Louis (USA)
Lewis, Brian (USA)
Meijer, Erik (USA)
Midkiff, Sam (USA)
Ortin, Francisco (Spain)
Palmisano, Ignazio (Italy)
Pearce, David (New Zealand)
Piessens, Frank (Belgium)
Safonov, Vladimir (Russia)
Schaefer, Stefans (Australia)
Scholz, Bernhard (Australia)
Schordan, Markus (Austria)
Siegmund, Frank (USA)
Srinkant, Y.N. (India)
Srisa-an, Witawas (USA)
Strein, Dennis (Germany)
Sturm, Peter (Germany)
Sullivan, Kevin (USA)
Tobies, Stephan (USA)
van den Brand, Mark (The Netherlands)
Vaswani, Kapil (India)
Veiga, Luis (Portugal)

Contents

Handling Session Classes for Predicting ASP.NET Performance Metrics

Ágnes Bogárdi-Mészöly
BUTE, Department of Automation and Applied Informatics
Goldmann György tér 3. IV. em.
1111, Budapest, Hungary
agi@aut.bme.hu

Tihamér Levendovszky
BUTE, Department of Automation and Applied Informatics
Goldmann György tér 3. IV. em.
1111, Budapest, Hungary
tihamer@aut.bme.hu

Hassan Charaf
BUTE, Department of Automation and Applied Informatics
Goldmann György tér 3. IV. em.
1111, Budapest, Hungary
hassan@aut.bme.hu

ABSTRACT

Distributed systems and web applications play an important role in computer science nowadays. The most common consideration is performance, because these systems must provide services with low response time, high availability, and certain throughput level. With the help of performance models, the performance metrics can be determined at the early stages of the development process. The goal of our work is to predict the response time, the throughput and the tier utilization of web applications, based on queueing models handling one and multiple session classes, with MVA and approximate MVA (Mean-Value Analysis) evaluation algorithm, in addition to balanced job bounds calculation. We estimated the model parameters based on one measurement. We implemented the MVA and the approximate MVA evaluation algorithm for closed queueing networks along with the calculation of the balanced job bounds with the help of MATLAB. We have tested a web application with concurrent user sessions in order to validate the models in ASP.NET environment.

Keywords

Web application, web performance, queueing models, performance prediction, and measurements.

1. INTRODUCTION

New frameworks and programming environments have been released to aid the development of complex web-based information systems. These new languages, programming models and techniques are proliferated nowadays, thus, developing such applications is not the only issue anymore: operating, maintenance and performance questions have become of key importance. One of the most important factors is performance, because network systems face a large number of users, they must provide high-availability services with low response time, while they guarantee a certain level of throughput.

These performance metrics depend on many factors. Several papers have investigated various configurable parameters, how they affect the performance of a web-based information system. Statistical methods, hypothesis tests are used in order to retrieve factors influencing the performance. An approach [Sop05a] applies analysis of variance, another [Bog05a] performs independence test.

The performance-related problems emerge very often only at the end of the software project. With the help of properly designed performance models, the performance metrics of a system can be determined at the earlier stages of the development process [Smi90a] [Smi01c]. In the past few years several methods have been proposed to address this goal. A group of them is based on queueing networks or extended versions of queueing networks [Jai91a] [Man02a] [Urg05a]. By solving the queueing model using analytical and simulation solutions, performance metrics can be predicted. Another group uses Petri-nets or generalized stochastic Petri-nets [Ber02b] [Kin99a], which can represent blocking and synchronization aspects much more than queueing networks. The third proposed approach uses a stochastic extension of process algebras, like TIPP (Time Processes and Performability Evaluation) [Her00a], EMPA (Extended Markovian Process Algebra) [Ber98a] and PEPA (Performance Evaluation Process Algebra) [Gil94a].

.NET Technologies 2006

Today one of the most prominent technologies of web-based information systems is Microsoft .NET. Our primary goal was to predict the response time of ASP.NET web applications based on queueing models handling one and multiple session classes, because response time is the only performance metric to which the users are directly exposed. Our secondary goals were to predict the throughput and the utilization of the tiers.

The organization of this paper is as follows. Section 2 covers backgrounds and related work. Section 3 presents our demonstration and validation of the models in the ASP.NET environment, namely, Section 3.1 describes our estimation of the model parameters, Section 3.2 presents our implementation of the MVA and the approximate MVA evaluation algorithm along with the calculation of the balanced job bounds, and Section 3.3 demonstrates our experimental configuration as well as our experimental validation of the models. Finally, we draw conclusions.

2. BACKGROUNDS AND RELATED WORK

Queueing theory [Jai91a] [Kle75a] is one of the key analytical modeling techniques used for computer system performance analysis. Queueing networks and their extensions (such as queueing Petri nets [Kou03a]) are proposed to model web applications [Man02a] [Urg05a] [Urg05b] [Smi00b].

In [Urg05a] [Urg05b], a basic queueing model with some enhancements is presented for multi-tier web applications. An application is modeled as a network of M queues: $Q_1,...,Q_M$ (Figure 1). Each queue represents an application tier, and it is assumed to have a processor sharing discipline, since this discipline closely approximates the scheduling policies applied by most of the operating systems.

A request can take multiple visits to each queue during its overall execution, thus, there are transitions from each queue to its successor and its predecessor as well. Namely, a request from queue Q_m either returns to Q_{m-1} with a certain probability p_m, or proceeds to Q_{m+1} with the probability $1-p_m$. There are only two exceptions: the last queue Q_M, where all the requests return to the previous queue $(p_M = 1)$, and the first queue Q_1, where the transition to the preceding queue denotes the completion of a request. S_m denotes the service time of a request at Q_m $(1 \le m \le M)$.

Internet workloads are usually session-based. The model can handle session-based workloads as an infinite server queueing system Q_0, that feeds the network of queues and forms the closed queueing network depicted in Figure 1. Each active session is in accordance with occupying one server in Q_0. The time spent at Q_0 corresponds to the user think time *Z*. It is assumed that sessions never terminate. Because of the infinite server queueing system, the model captures the independence of the user think times and the service times of the request at the application.

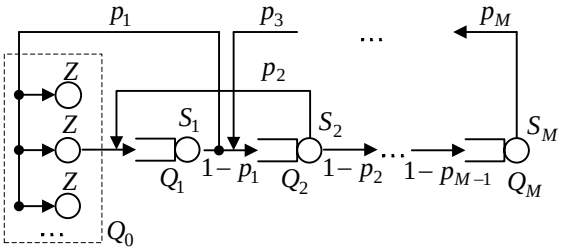

Figure 1. Modeling a multi-tier web application using a queueing network

An enhancement of the baseline model [Urg05a] can handle multiple session classes. Incoming sessions of a web application can be classified into multiple (*C*) classes. *N* is the total number of sessions, and N_c denotes the number of sessions of class *c*, thus, $N = \sum_{c=1}^{C} N_c$. A feasible population with *n* sessions means that the number of sessions within each class *c* is between *0* and N_c, and the sum of the number of sessions in all classes is *n*. In order to evaluate the model, the service times, the visit ratios, and the user think time must be measured on a per-class basis.

The model can be evaluated for a given number of concurrent sessions *N*. A session in the model corresponds to a customer in the evaluation algorithm. The MVA (Mean-Value Analysis) algorithm for closed queueing networks [Jai91a] [Rei80a] iteratively computes the average response time of a request and the throughput. The algorithm introduces the customers into the queueing network one by one, and the cycle terminates when all the customers have been entered.

In addition, the utilization of the queues can be determined from the model, using the utilization law [Jai91a] [Kle75a]. The utilization of the queue *m* is $U_m = XV_mS_m$, where *X* is the throughput and V_m is the visit number (the number of visits to Q_m made by a request during its processing).

The MVA algorithm is only applicable if the queueing network is in product form, namely, the network has to satisfy the conditions of the job flow

balance, one-step behavior, and device homogeneity. Furthermore, the queues are assumed either fixed-capacity service centers or infinite servers, and in both cases exponentially distributed service times are assumed.

MVA is a recursive algorithm. Handling one session class for large values of customers, or if the performance for smaller values is not required, MVA can be too expensive computationally. If we handle multiple session classes, the time and space complexities of MVA are proportional to the number of feasible populations, and this number rapidly grows for relatively few classes and jobs per class. Thus, it can be worth using an approximate MVA algorithm [Rai91a] [Sin05a] or a set of two-sided bounds [Rai91a] [Zah82a].

These bounds referred to as balanced job bounds are based on the issue that a balanced system has a better performance than a similar unbalanced system. A system without a bottleneck device is called a balanced system, in other words, the total service time demands are equal in all queues. The balanced job bounds are very tight, the upper and lower bounds are very close to each other as well as to the real performance. D is the sum of total service demands, $D_{avg} = D/M$ is the average service demand per queue, and $D_{\max}$ is the maximum service demand per queue.

$$\max\left\{ND_{\max} - Z, D + (N-1)D_{avg}\frac{D}{D+Z}\right\}$$
$$\le R(N) \le D + (N-1)D_{\max}\frac{(N-1)D}{(N-1)D+Z}$$

$$\frac{N}{Z + D + (N-1)D_{\max}\frac{(N-1)D}{(N-1)D+Z}}$$
$$\le X(N) \le \min\left\{\frac{1}{D_{\max}}, \frac{N}{Z + D + (N-1)D_{avg}\frac{D}{D+Z}}\right\}$$

The model validation presented in [Urg05a] was executed in a J2EE environment, while in this paper the models are demonstrated and validated in an ASP.NET environment. In order to improve the model, it must be enhanced to handle the limits of the four thread types in .NET thread pool, in addition to the global and the application queue limit [Mei04a], since in previous work [Bog05a] we have proven by statistical methods [Bra87a], that the limits of the four thread types, namely, the *maxWorkerThreads*, *maxIOThreads*, *minFreeThreads*, *minLocalRequest-FreeThreads*, along with the global and application queue limit, namely, the *requestQueueLimit*, *app-RequestQueueLimit* parameters have a considerable effect on performance, in other words, they are performance factors.

3. CONTRIBUTIONS

We have implemented a three-tier ASP.NET test web application (Figure 2). Compared to a typical web application, it has been slightly modified to suit the needs of the measurement process.

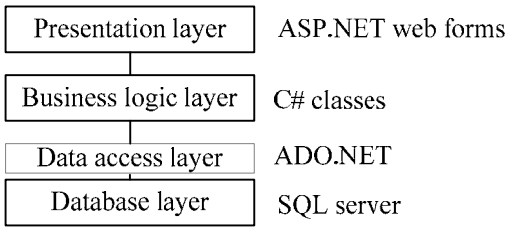

Figure 2. The test web application architecture

Thereafter, we have demonstrated and validated the models in the ASP.NET environment. Firstly, we have estimated the input values of the model parameters from one measurement. Secondly, we have implemented the MVA and the approximate MVA algorithm, along with the calculation of the balanced job bounds with the help of MATLAB, and we have evaluated the model. Finally, we have tested a typical web application with concurrent user sessions, comparing the observed and predicted values in order to validate the models in the ASP.NET environment.

We expect that the baseline model and the model handling multiple session classes can be validated in ASP.NET environment. The thread pool settings and the queue limits are common in the two environments (J2EE and .NET), but the concrete threads (four types, their partitioning in the thread pool) and queues (two types, their placement and configuration) are specific to .NET. Thus, a general model for the two environments with specific extensions is expected, which is more accurate than the baseline model or the model handling multiple session classes. The algorithms presented in [Urg05a] could not be reused directly, because they must be extended.

Estimation of the Model Parameters

The web application was designed in a way that the input values of the model parameters can be determined from the results of one measurement. Each page and class belonging to the presentation, business logic or database was measured separately.

Handling one session class, the input parameters of the model are the number of tiers, the maximum number of customers (simultaneous browser connections), the average user think time $\overline{Z}$, the visit number V_m and the average service time $\overline{S}_m$ for Q_m $(1 \le m \le M)$.

During the measurements the number of tiers was constant (three). The maximum number of customers means that the load was characterized as follows: we started form one simultaneous browser connection then we continued with two, until 52 had been reached. In order to determine the average user think time we averaged the sleep times in the user scenario. To determine V_m we summed the number of requests of each page and class belonging to the given tier in the user scenario. To estimate $\overline{S}_m$ we averaged the service times of each page and class belonging to the given tier.

Handling multiple session classes, the input model parameters are the number of tiers, the number and the maximum number of customers, respectively, on a per-class basis, the average user think time $\overline{Z}_c$, the visit number $V_{m,c}$, and the average service time $\overline{S}_{m,c}$ for Q_m $(1 \le m \le M,\ 1 \le c \le C)$.

There were two classes. The number of sessions for one class was constant 10, while the number of simultaneous browser connections for the other class was varied up to a maximum number of customers. The load was characterized as follows: we started from one simultaneous browser connection then we continued with 5, 10, until 70 had been reached. To determine $\overline{Z}_c$, the sleep times in the user scenario were averaged per class. In order to determine $V_{m,c}$, the number of requests of each page and class belonging to the given tier and class in the user scenario was summed. In order to estimate $\overline{S}_{m,c}$, the service times of each page and class belonging to the given tier and class were averaged.

Model Evaluation

The conditions described in Section 2 have been satisfied: the number of arrivals to a queue equals the number of departures from the queue, the simultaneous job moves are not observed, since the queues have processor sharing discipline, and finally, the service rate of a queue does not depend on the state of the system in any way except for the total queue length. In addition, the queues Q_1, Q_2, Q_3 are fixed-capacity centers, and the Q_0 queue is an infinite server. Therefore, the MVA algorithm can be applicable to evaluate the model (Figure 1) of the test web application (Figure 2), because the model is in a product form.

We implemented the MVA and approximate MVA algorithm for closed queueing networks, in addition the calculation of the balanced job bounds with the help of MATLAB. Our MATLAB scripts can be downloaded from [Mat06a].

When we handle one session class, the inputs of the script are the number of tiers, the maximum number of customers, the average service times, the visit numbers, and the average user think time. When we handle multiple session classes, the inputs the number of tiers, the number and the maximum number of customers, respectively, on a per-class basis the average service times, the visit numbers, and the average user think time. The scripts compute the response times, the throughputs and the tier utilizations up to a maximum number of customers. MVA provides a recursive way, approximate MVA computes these in a few steps, while balanced job bounds method completes in one step.

Model validation

Finally, our experimental configuration and experimental validation of the model in ASP.NET environment are demonstrated.

The web server of our test web application was Internet Information Services (IIS) 6.0 with ASP.NET 1.1 runtime environment, one of the most proliferated technologies among the commercial platforms. The database management system was Microsoft SQL Server 2000 with Service Pack 3. The server runs on a 2.8 GHz Intel Pentium 4 processor with Hyper-Threading technology enabled. It had 1GB of system memory; the operating system was Windows Server 2003 with Service Pack 1. The emulation of the browsing clients and measuring the response time were performed by ACT (Application Center Test), a load generator running on another PC on a Windows XP Professional computer with Service Pack 2 installed. It ran on a 3 GHz Intel Pentium 4 processor with Hyper-Threading technology enabled, and it also had 1GB system memory. The connection among the computers was provided by a 100 Mb/s network.

ACT [Ald03a] is a well-usable stress testing tool included in Visual Studio .NET Enterprise and Architect Editions. The test script can be recorded or manually created. Virtual users send a list of HTTP requests to the web server concurrently. Each test run takes 2 minutes and 10 seconds warm-up time for the load to reach a steady-state. In the user scenario, sleep times are included to simulate the realistic usage of the application.

When we handle one session class, while the number of simultaneous browser connections varied, the average response time and throughput per class were measured (Figure 3).

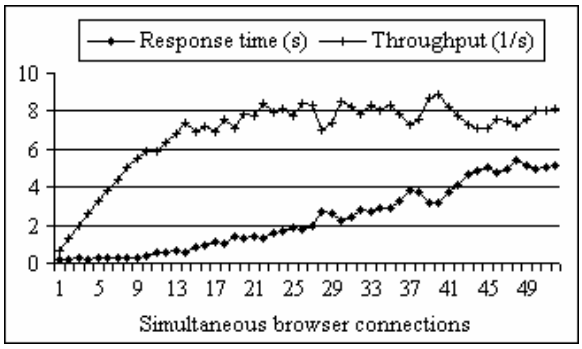

Figure 3. The observed response times and throughputs handling one session class

Handling multiple session classes, there were two classes of sessions: a database reader and a database writer. The number of simultaneous browser connections of one class was fixed at 10, while the number of simultaneous browser connections of the other class varied, and we measured the average response time and throughput per class (Figure 4).

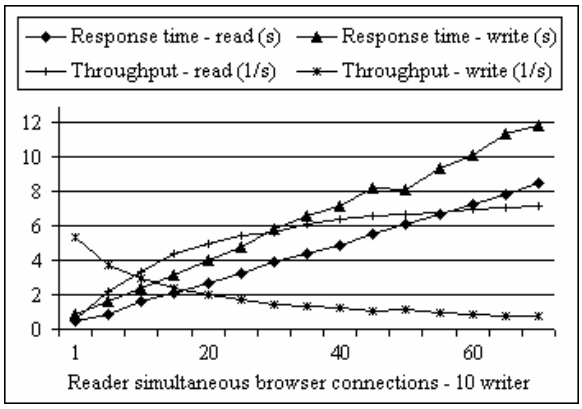

Figure 4. The observed response times and throughputs handling multiple session classes

The results presented in Figure 3 and in Figure 4 correspond to the common shape of response time and throughput performance metrics. Increasing the number of concurrent (reader) clients, the (reader) throughput (served requests per second) grows linearly, while the average (reader) response time advances barely. After the saturation the (reader) throughput remains approximately constant, and an increase in the (reader) response time can be observed. In the overloaded phase, the (reader) throughput falls, while the (reader) response time becomes unacceptably high.

Handling one session class, we experimentally validated the model to demonstrate its ability to predict the response time and the throughput of ASP.NET web applications with MVA (Figure 5), and approximate MVA algorithm. We have found that the model handling one session class predicts the response time and throughput acceptably.

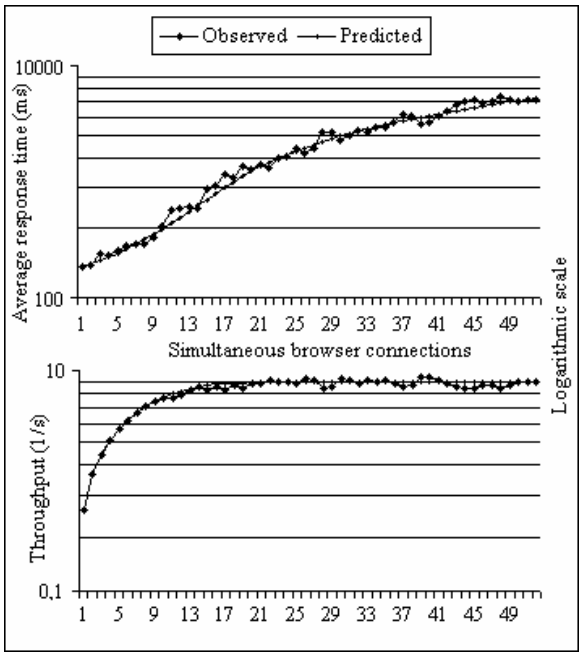

Figure 5. The observed and predicted response times and throughputs handling one session class with MVA

Moreover, from the model, the utilization of the tiers can be predicted. The results are depicted in Figure 6. The presentation tier is the first that becomes congested. The utilization of the database queue is the second (29%), and the utilization of the business logic queue is the last one (17%).

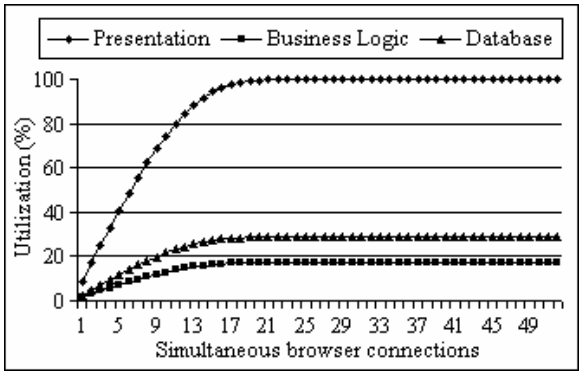

Figure 6. The tier utilization handling one session class with MVA

Thereafter, we demonstrate that the response time, the throughput and the tier utilization of ASP.NET web applications move within tight upper and lower bounds (Figure 7, Figure 8). We have found that the

response time, the throughput, and the queue utilization from the observations fell into the upper and lower bounds. Thus, the balanced job bounds handling one session class predict the response time, the throughput, and the utilization of the tiers acceptably.

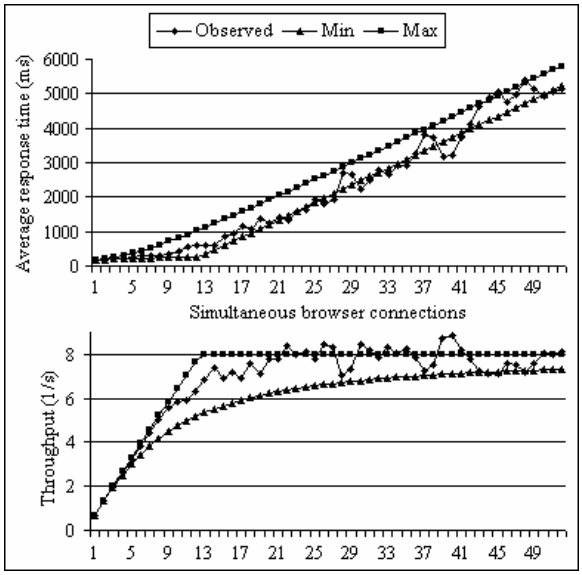

Figure 7. The observed and predicted response times and throughputs handling one session class with balanced job bounds

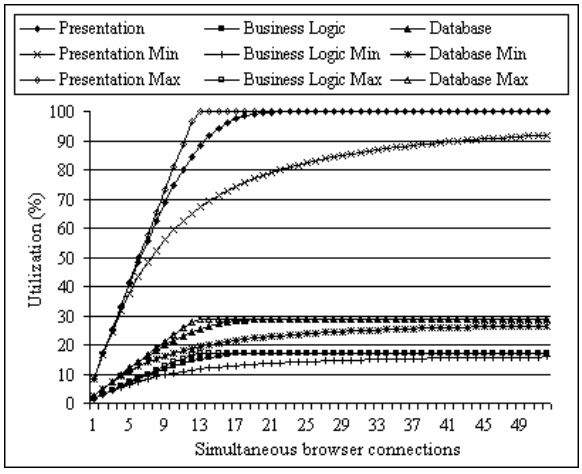

Figure 8. The tier utilization handling one session class with balanced job bounds

Finally, the model handling multiple session classes was experimentally validated. We have found that the model predicts the response time and throughput with approximate MVA acceptably (Figure 9). While the presentation tier is congested, the utilization of the database queue is about 84%, and the utilization of the business logic queue is about 16% (Figure 10). We have found that the response time, the throughput, and the utilization from the observations as well as from the approximate MVA fell into the upper and lower bounds. Hence, the balanced job bounds predict the response time, the throughput, and the utilization acceptably (Figure 11).

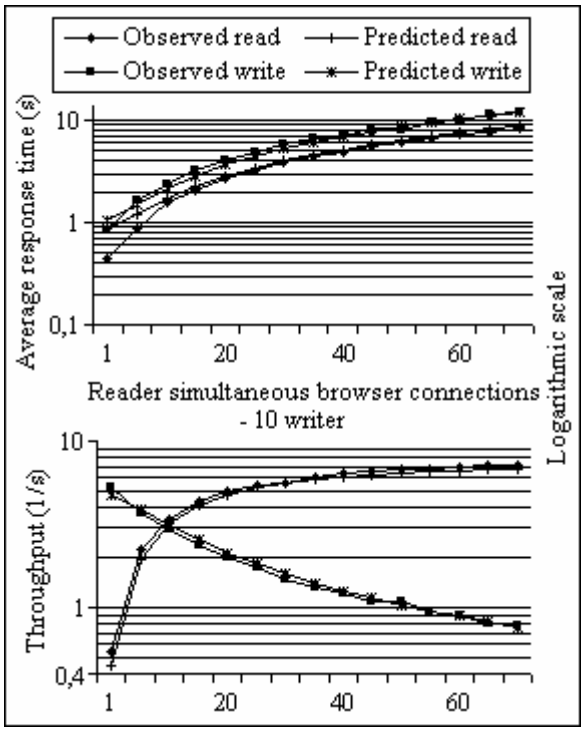

Figure 9. The observed and predicted response times and throughputs handling multiple session classes with approximate MVA

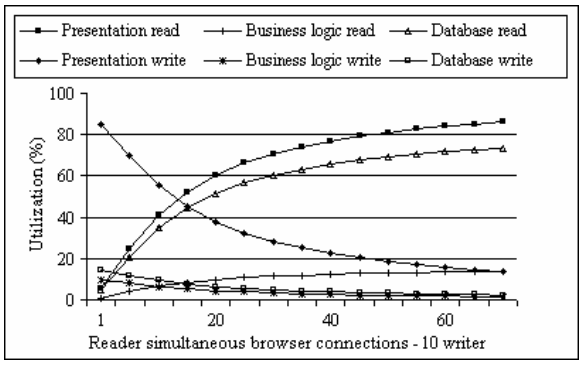

Figure 10. The tier utilization handling multiple session classes with approximate MVA

4. CONCLUSIONS AND FUTURE WORK

We have demonstrated and validated queueing models handling one and multiple session classes in ASP.NET environment, namely, the input model parameters were estimated from one measurement, the MVA and approximate MVA evaluation algorithm, in addition the calculation of the balanced job bounds were implemented with the help of MATLAB, and a measurement process was executed in order to experimentally validate the models.

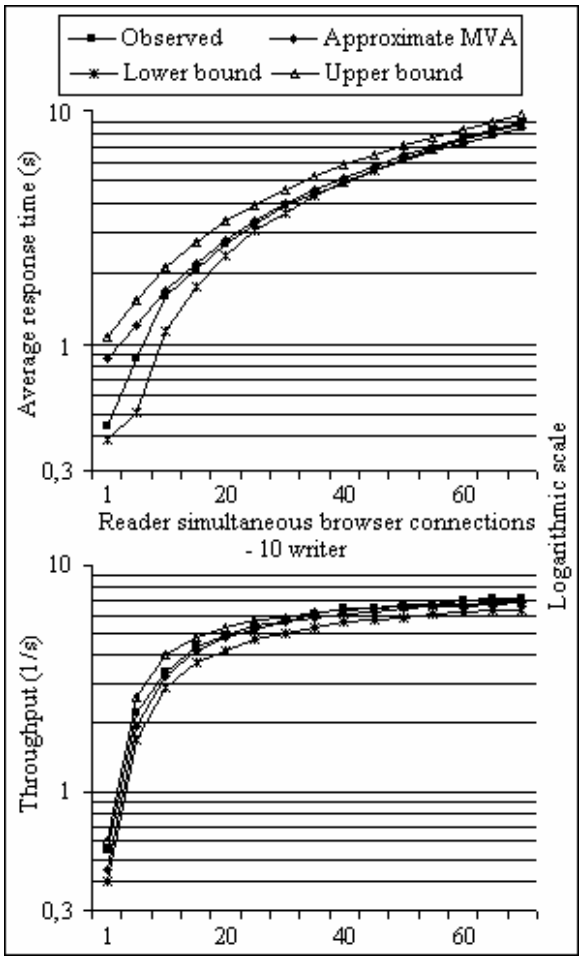

Figure 11. The observed and predicted response times and throughputs handling multiple session classes with balanced job bounds

Our results have shown that the models handling one and multiple session classes predict the response time and the throughput acceptably with MVA and approximate MVA evaluation algorithm, along with the calculation of balanced job bounds. Furthermore, the presentation tier is the first to become congested. The utilization of the database tier is the second one, and the utilization of the business logic queue is the last one.

In order to improve the model, the limits of the four thread types in .NET thread pool, the global and application queue limits must be handled along with other features. These extensions of the model and the validation of the enhanced models, as well as the validation of the models in ASP.NET 2.0 environment are subjects of future work.

5. REFERENCES

[Ald03a] Aldous, J., and Finnel, L. Performance Testing Microsoft .NET Web Applications. Microsoft Press, 2003.

[Ber98a] Bernardo, M., and Gorrieri, R. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. Journal of Theoretical Computer Science, Vol. 202, pp. 11-54, 1998.

[Ber02b] Bernardi, S., Donatelli, S., and Merseguer, J. From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. In Proceedings of ACM International Workshop Software and Performance. Rome, Italy, pp. 35-45, 2002.

[Bog05a] Bogárdi-Mészöly, Á., Szitás, Z., Levendovszky, T., Charaf, H. Investigating Factors Influencing the Response Time in ASP.NET Web Applications. Proceedings of Lecture Notes in Computer Science, 3746, pp. 223-233, 2005.

[Bra87a] Brase, C.H., and Brase, C.P. Understandable Statistics. D. C. Heath and Company, 1987.

[Gil94a] Gilmore, A.S., and Hillston, J. The PEPA Workbench: A Tool to Support a Process Algebra-Based Approach to Performance Modelling. In Proceedings of Seventh International Conference Modelling Techniques and Tools for Performance Evaluation, pp. 353-368, 1994.

[Her00a] Herzog, U., Klehmet, U., Mertsiotakis, V., and Siegle, M. Compositional Performance Modelling with the TIPPtool. Journal of Performance Evaluation, Vol. 39, pp. 5-35, 2000.

[Jai91a] Jain, R. The Art of Computer Systems Performance Analysis. John Wiley and Sons, 1991.

[Kin99a] King, P., and Pooley, R. Derivation of Petri Net Performance Models from UML Specifications of Communication Software. In Proceedings of 25th UK Performance Eng. Workshop, 1999.

[Kle75a] Kleinrock, L. Queueing Systems, Volume 1: Theory. John Wiley and Sons, 1975.

[Kou03a] Kounev, S., Buchmann, A. Performance Modelling of Distributed E-Business Applications using Queueing Petri Nets. Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, Austin, Texas, USA, 2003.

[Man02a] Manescé, D.A., and Almeida, V.A.F. Capacity Planning for Web Services. Prentice Hall, 2002.

[Mat06a] Our MATLAB scripts can be downloaded from - http://avalon.aut.bme.hu/~agi/research/

[Mei04a] Meier, J.D., Vasireddy, S., Babbar, A., and Mackman, A. Improving .NET Application Performance and Scalability (Patters & Practices). Microsoft Corporation, 2004.

[Rei80a] Reiser, M., and Lavenberg, S.S. Mean-Value Analysis of Closed Multichain Queuing Networks. Journal of Association for Computing Machinery, Vol. 27, pp. 313-322, 1980.
[Sin05a] Sinclair, B., Mean Value Analysis. Computer Systems Performance Handout, 2005.
[Smi90a] Smith, C.U. Performance Engineering of Software Systems. Addison-Wesley, 1990.
[Smi00b] Smith, C.U., Williams, L.G. Building responsive and scalable web applications. Computer Measurement Group Conference, Orlando, FL, USA, pp. 127-138, 2000.
[Smi01c] Smith, C.U., and Williams, L. G. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, 2001.
[Sop05a] Sopitkamol, M., and Menascé, D.A. A Method for Evaluating the Impact of Software Configuration Parameters on E-Commerce Sites. In Proceedings of the ACM 5th International Workshop on Software and Performance, Palma, Illes Balears, Spain, pp. 53-64, 2005.
[Urg05a] Urgaonkar, B. Dynamic Resource Management in Internet Hosting Platforms. Dissertation, Massachusetts, 2005.
[Urg05b] Urgaonkar, B., Pacifici, G., Shenoy, P., Speitzer, M., and Tantawi, A. An Analytical Model for Multi-tier Internet Services and its Applications. Journal of ACM SIGMETRICS Performance Evaluation Review, Vol. 33, No. 1, pp. 291-302, 2005.
[Zah82a] Zahorjan, J., Sevcik, K.C., Eager D.L., and Galler, B. Balanced Job Bound Analysis of Queueing Networks. Journal of Communications of the ACM, Vol. 25, No. 2, pp. 134-141, 1982.

Towards Effective Runtime Trace Generation Techniques in the .NET Framework *

Krisztián Pócza
Eötvös Loránd University
Fac. of Informatics, Dept. of Programming Lang. and Compilers
Pázmány Péter sétány 1/c.
H-1117, Budapest, Hungary
kpocza@kpocza.net

Mihály Biczó
Eötvös Loránd University
Fac. of Informatics, Dept. of Programming Lang. and Compilers
Pázmány Péter sétány 1/c.
H-1117, Budapest, Hungary
mihaly.biczo@axelero.hu

Zoltán Porkoláb
Eötvös Loránd University
Fac. of Informatics, Dept. of Programming Lang. and Compilers
Pázmány Péter sétány 1/c.
H-1117, Budapest, Hungary
gsd@elte.hu

ABSTRACT

Effective runtime trace generation is vital for understanding, analyzing, and maintaining large-scale applications. In this paper two cross-language trace generation methods are introduced for the .NET platform. The non-intrusive methods are based on the .NET Debugging and Profiling Infrastructure; consequently, neither additional development tools, nor the .NET Framework SDK is required to be installed on the target system. Both methods are applied to a test set of real-size executables and compared by performance and applicability.

Keywords

Runtime trace generation, .NET, Debugger, Profiler, program slicing

1. INTRODUCTION

Generating and analyzing runtime traces for large scale enterprise applications is a common task to investigate the cause of arising malfunctions and accidental crashes.

In order to prepare reliable applications, it is important to investigate programs using a debugger application, and examine the application log or the event log of the operating system so that erroneous instructions and variables getting incorrect values can be detected. However, there are many situations where a simple debugger fails to find the erroneous instructions and variables. One common example is when the error occurs in a production environment where we are not allowed to install a development environment to detect the bug [Mar03a]. Furthermore, multithreaded applications or applications producing incorrect behavior only under heavy load often may not be debugged correctly on the development machines. What makes things even more complicated is that incompatibility issues might also arise in the case of programs and components that run on a deployment server or a client computer. Further problematic situations include cases when the deployment servers are in a Network Load Balancing (NLB) Cluster, or the isolation level on the IIS web server is too restrictive.

The most common research area where low level runtime traces are used in the academic world is dynamic program slicing [Agr91a, Bes01a, Póc05a, Tip95a, Zha03a]. The result of program slicing can also be used in the industry. The original goal of program slicing was to map mental abstractions made by programmers during debugging to a reduced set of statements in source code. With the help of program slicing programmers are able to identify bugs more precisely and at a much earlier stage.

In this article we show two different methods for generating source code statement level runtime traces for applications hosted by the Microsoft .NET Framework 2.0. In their current form our solutions are incompatible with older versions (1.0, 1.1) of the .NET Framework but they can be ported back. None of our methods requires the modification of the original source code nor the Runtime. Consequently, these solutions do not depend on either Rotor (the Shared Source implementation of the .NET Framework), Mono, or any other open source software.

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

* Supported by GVOP-3.2.2.-2004-07-0005/3.01

None of the methods requires the installation of the development tools or the Microsoft .NET Framework SDK on the target machine, and since .NET is a cross-language programming environment, they can be used to generate trace for programs written in any .NET programming language.

The first trace generating method uses the .NET Debugger which we presented in [Póc05] in order to utilize it in our dynamic slicing algorithm, while the second approach exploits the capabilities of the .NET Profiling API and IL code rewriting [Mik03]. It will clear up that only the second method is suitable for large scale multithreaded applications, and the first method is sufficient only for toy programs.

In the next section we describe the main concepts and the architecture of the *.NET Debugging and Profiling Infrastructure*. In the 3rd section we will describe the method that uses the *.NET Debugger* to generate trace, while in the 4th section the second solution based on the *.NET Profiler* and *IL code rewriting* technique will be presented. In the 5th section we compare these methods and present performance figures with different applications. We primarily focus on tracing statements of the original source code that appear in the execution path, and will not give detailed description on how to identify variables. However, in the last section we show how the prepared solutions can be complemented to identify variables.

2. .NET DEBUGGING AND PROFILING INFRASTRUCTURE

All 20+ .NET languages compile to an intermediate language code called *Common Intermediate Language (CIL)* or simply *Intermediate Language (IL)*. The compiled code is organized into assemblies. Assemblies are portable executables - similar to dll's - with the important difference that assemblies are populated with .NET metadata and IL code instead of normal native code. The .NET metadata holds information about the defined and referenced assemblies, types, methods, class member variables and attributes [ECMA]. IL is a machine-independent, programming language-independent, low-level, assembly-like language using a stack to transfer data among IL instructions. The IL code is jitted by the .NET CLR (Common Language Runtime) to machine-dependent instructions at runtime.

With the release of .NET, a new Debugging API has also been introduced in the Microsoft world. Script engines can now compile or interpret code for the Microsoft Common Language Runtime (CLR) instead of integrating debugging capabilities directly into applications through Active Scripting [Pell]. .NET Debugging Services is not only able to debug every code compiled to IL written in any high level language, but it also provides debugging capabilities for all modern Object Oriented languages.

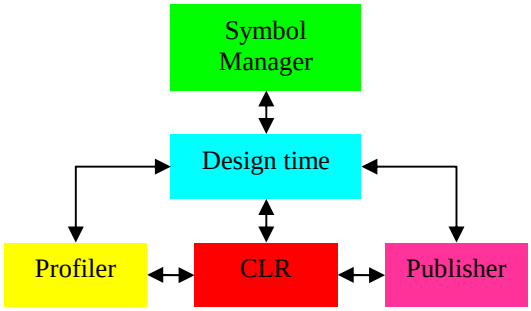

Figure 1: CLR Debugging architecture

The .NET CLR supports two types of debugging modes: out-of-process and in-process.

Out-of-process debuggers run in a separate process providing common debugger functionality.

In-process debuggers are used for inspecting the run-time state of an application and for collecting profiling information. These kinds of debuggers (profilers) do not have the ability to control the process or handle events like stepping, breakpoints, etc.

The CLR Debugging Services are implemented as a set of some 70+ COM interfaces, which include the *design-time application*, the *symbol manager*, the *publisher* and the *profiler*.

The *design-time interface* is responsible for handling debugging events. It is implemented separated from the CLR while the host application must reside in a different process. The application has a separate thread for receiving debugger events that run in the context of the debugged application. When a debug event occurs (assembly loaded, thread started, breakpoint reached, etc.) the application halts and the debugger thread notifies the debugging service through callback functions.

The *symbol manager* is responsible for interpreting the program database (PDB) files that contain data used to describe code for the modules being executed. The debugger also uses assembly metadata that also holds useful information described earlier. The PDB files contain debugging information and are generated only when the compiler is explicitly forced to do so. Besides enabling the unique identification of program elements like classes, methods, variables and statements, the metadata and the program database can also be used to retrieve their original position in the source code.

The *publisher* is responsible for enumerating all running managed processes in the system.

The *profiler* tracks application performance and resources used by running managed processes. The profiler runs in-process of the inspected application and can be used to handle events like module and class loading/unloading, jitting, method calls, events related to exceptions and garbage collection performance.

3. .NET DEBUGGER WAY TO INSTRUMENT APPLICATIONS

To employ the *Debugger* first we set a breakpoint to the entry point of our application and we step along each executing statement until the end. The step (or step-in) debugging operation goes along sequence points in the original source code. Sequence points which can be identified using metadata and the program database divide the statements in high-level programming languages.

The CLR Debugger API called *ICorDebug* [Stall] is implemented by native COM interfaces. It can be directly reached from managed or unmanaged code but there are also higher level managed wrapper classes used by MDbg [Stall], the managed debugger part of the Microsoft .NET Framework 2.0 SDK with full source code.

Using these interfaces we can start a process for debugging and register our managed or unmanaged callback functions. As mentioned earlier, querying run-time information of program variables is another important application.

The structure of our solution:

1. Low level managed COM Wrapper
2. High level managed API of the previous
3. Application employing the previous to generate runtime execution trace

The 1st and the 2nd layer of our solution is not implemented by us rather we borrowed it from MDbg that is freely usable and provided by Microsoft.

The low level managed COM Wrapper (1st layer) represents a COM marshaling code that is used to call native Debugging API functions and is written in IL. It resides in the corapi2 folder in MDbg's source tree.

The high level managed API (2nd layer) provides an easy-to-use higher level managed wrapper to the underlying layer and it is written in C# 2.0. Sometimes it uses properties instead of methods, and dispatches native debugging events as managed events. It resides in the corapi folder of MDbg's source tree.

Our solution based on these APIs can be downloaded from http://avalon.inf.elte.hu/src/netdebug/ .

In the implementation first we create the process to be run but do not start it. A Debugger event is raised at every module load. When the module containing the user entry point (Main method) is loaded we set a breakpoint at this entry point. After loading the process and setting the breakpoint we let the application run. At this point the process is actually created and the *OnCreateProcess* event is raised by the Debugger. In the handler of this event we set the state of the application being debugged to running and start a while loop which is allowed to run while the application is alive. When the breakpoint previously set is encountered the *OnBreakPoint* debug event is raised. In the handler of this debug event an *AutoResetEvent* called *eventComplete* is set and we wait for *eventModState* to be set. The handler of *OnStepComplete* Debugger event does exactly the same.

Afterwards the while loop does the following three things:

1. Waits for the *eventComplete* event which is set by the Debugger event handlers
2. *doStepIn* operation is called as described later
3. Sets the *eventModState* event

Between setting the *eventComplete* event and waiting for the *eventModState* event the *doStepIn* method runs which requires/sets the following information at every step:

1. The IL instruction pointer
2. The current function token and module
3. Which sequence point belongs to the current IL instruction
4. The target of the next step

The IL instruction pointer, the function token and the module can be easily queried from the *CorFrame* object which can be queried from the current thread. The sequence points are required to output the actual source line and source column to the trace and to define the next step using the *StepRange* method of *CorStepper*. The sequence points and the target of the next step are static properties, therefore we cache them so that they can be queried by the *GetSequencePoints* and *GetRanges* method of the current *ISymbolMethod* interface accordingly. At the first and last sequence point of each function we log a function enter and leave event in the trace.

Unfortunately, this approach is not able to correctly handle multithreaded application because it is not possible to step from one thread to another and the debugger does not notify us about thread switches.

4. .NET PROFILER WAY TO INSTRUMENT APPLICATIONS

Basically, this approach explores all sequence points in all methods of all classes and all modules of the application being profiled and inserts trace method calls defined in an outer assembly at every sequence point at IL code level [Mik03].

The .NET Profiler provides a COM interface called *ICorProfilerCallback2* exposing a set of callbacks which can be implemented as a COM class. The implementer is not allowed to use any managed programming language, otherwise the Profiler would profile itself. Consequently we have chosen the C++ language to demonstrate this approach.

We have used some other COM interfaces also like *ISymUnmanagedReader*, *ISymUnmanagedMethod*, *IMetaDataImport* and *ICorProfilerInfo2* while the standard classes implementing these interfaces were instantiated using Microsoft's ATL (Active Template Library).

From the 70+ Profiler events provided by the *ICorProfilerCallback2* interface we have used only two: *ModuleLoadFinished* and *ClassLoadFinished*.

4.1. Tracing Methods: Implementation and Referencing

In this section we discuss the tracing methods we are using, how they log and the way we reference them.

We created a module (assembly) called *TracerModule* and placed a static class called *Tracer* in it containing only static methods.

```
public static void DoFunc(uint startLine,
   uint startColumn, uint endLine, uint endColumn,
   uint functionID, uint action)
{
  try
  {
    lock (lockObj)
    {
      char act = 'E';
      if (action == 2)
        act = 'L';
      sw.WriteLine("{6}T{5}{4}{0}:{1}-{2}:{3}",
         startLine, startColumn, endLine,
         endColumn, act, functionID,
         Thread.CurrentThread.ManagedThreadId);
    }
  }
  catch { }
}
```

Listing 1: Trace method

Listing 1 illustrates the trace method executed at every method entry (first sequence point executed) and leave (last sequence point, which is always executed unless exception has been thrown).

The first four parameters represent the position of the sequence point in the source code, the fifth parameter represents the unique function identifier and the action code (1 for E(nter), 2 for L(eave)). Since the tracer is prepared for multithreaded applications, we *lock* on a static object and output the unique managed thread identifier at every step. At intra-function sequence points the trace method gets only the first four parameters and does not output any function identifier or action code.

If we intend to call a method placed in an outer module we have to reference the assembly containing that method, the class and the method itself. We decided not to modify the original program in any way so we have to add these references to the in-memory metadata of every assembly at runtime. The best place to do this is the *ModuleLoadFinished* Profiler event.

Through the *DefineAssemblyRef* method of the *IMetaDataAssemblyEmit* interface, the *DefineTypeRefByName* and the *DefineMemberRef* methods of *IMetaDataEmit2* interface we are able to add these references to the in-memory metadata of assemblies and receive their *token* values. When adding these references they are specified simply by their names, the function token is used to call the belonging function at the corresponding sequence points.

4.2. Internal Representation of Native .NET Primitives

In this section we will give a general overview of the internal representation of .NET methods, IL instructions and Exception Handling Clauses [Mik03].

4.2.1. Internal representation of .NET methods

Every .NET method has a header, IL code and may have extra padding bytes to maintain DWORD alignment. Optionally, it may have an SEH (Structured Exception Handling) header and Exception Handling Clause.

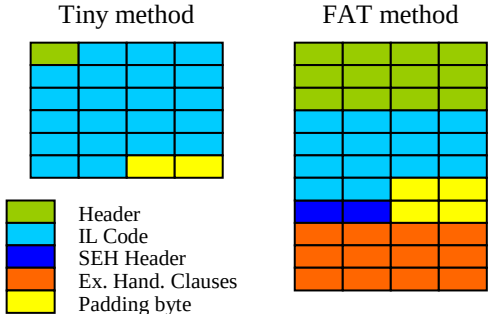

Figure 2: Method formats

A .NET method can be in *Tiny* and in *Fat* format. A Tiny method is smaller than 64 bytes, its stack depth does not exceed 8 slots, contains no local variables, SEH header and exception handlers. Fat methods

overrun one or more of these criterions.

4.2.2. IL instruction types

IL instructions can be divided into several categories based on the number and type of parameters they use:

- have no parameter (dup: duplicates the element on top of the stack; ldc.i4.-1,…ldc.i4.8: load an integer on stack (-1,…8))
- has one integer (8, 16, 32, 64 bits long) parameter (ldc.i4 <int>: load the integer specified by <int> on stack; br <param>, br.s <reloff>: long or short jump to the relative address specified by <reloff>)
- has one token parameter (call <token>: calls the method specified by <token>; box <token>: box a value type with type <token> into an object; ldfld <token>: load the field specified by <token> of the stack-top class on stack)
- multi-parameter instructions (switch <count> <reloff1>…<reloffcount>: based on the stack-top value representing the relative offset parameter index jumps to the chosen relative offset)

4.2.3. Exception Handling Clauses

Every *Fat* method can have one or more exception handlers. Every EHC (Exception Handling Clause) has a header and specifies its *try* and *handler* starting (absolute) offset and length. An EHC can be also in *Tiny* and *Fat* format based on the number of bytes the offset and length properties are used to describe. Obviously each EH offset and length specifies a sequence point beginning and ending position in the IL code-flow.

4.3. Let the Game Begin: IL Code Rewriting

Our goal is to change the IL Code of methods before they are jitted to native code. We have chosen the *ClassLoadFinished* Profiler event to perform this operation because in this early stage we are able to enumerate all methods (with the *EnumMethods* method of *IMetaDataImport interface*) of the class just loaded and rewrite the IL code of a whole bunch of methods. The binary data of a method can be retrieved by the *GetILFunctionBody* method of *ICorProfilerInfo2*. After IL code rewriting, necessary space for the new binary data can be allocated using the *Alloc* method of *IMethodMalloc* and the binary data can be set with the *SetILFunctionBody* method of *ICorProfilerInfo2*.

Single-method binary data operations and IL code rewriting can be divided into five steps:

1. Parsing binary data and storing it in custom data structures
2. Upgrading method and instruction format
3. Insertion of instrumentation code to the IL code-flow
4. Recalculating offsets and lengths
5. Storing new representation in binary format

4.3.1. Parsing binary method data

At first we determine the sequence points of the method being parsed using the *GetSequencePoints* method of *ISymUnmanagedMethod*. This procedure determines the IL- and original source code-level start and end offsets for every sequence point. The first byte of the header describes whether the method is tiny or fat, the function is parsed using this information.

The IL-level offsets of sequence points were determined previously, now the binary data has to be assigned to them and the IL instructions have to be identified based on the binary data at every sequence point. Every category of IL instructions featured in 4.2.2 is able to parse itself and determine its parameters (integer value, token value, multiple parameters). Furthermore it can also generate both a human readable and a binary representation (along with its length) of it.

```
static bool IsFirstLess(int value1, int value2)
{
  if (value1 < value2)
  {
    Console.WriteLine("Yes, first is less");
    return true;
  }
  return false;
}
```

Listing 2: Simple C# Method

Consider the simple method in Listing 2. In Table 1 the corresponding sequence points are shown identified by their IL offset, the start and end offsets by line and column numbers.

Index	IL offset	Start offset	End offset
0	0	25,1	25,2
1	1	26,3	26,23
2	9	0xfeefee,0	0xfeefee,0
3	12	27,3	27,4
4	13	28,7	28,47
5	24	29,7	29,19
6	28	31,3	31,16
7	32	32,1	32,2

Table 1: Sequence Point Offsets

Sequence point at index 2 petted FeeFee does not have a real source code level offset just helps us to jump out if the predicate fails.

The IL code in Listing 4 illustrates the internal representation of method in Listing 2. The numbering on the left indicates the IL offsets while the numbers right to the branch instructions (*brtrue.s*, *br.s*)

represents absolute target offset, relative target offset, target sequence point and target instruction index at the target sequence point. Parameters of *ldstr* and *call* instructions are of type string and functions tokens respectively. The absolute target offset of branch instructions identified by target IL instruction has to be calculated from the instruction offset and the relative target offset.

If exist, the EHCs are also parsed [Mik03].

```
0: nop
1: ldarg 0
2: ldarg 1
3: clt
5: ldc.i4 0
6: ceq
8: stloc 1
9: ldloc 1
10: brtrue.s 28 (16) [tsp: 6, til: 0]
12: nop
13: ldstr 1879048193
18: call 167772181
23: nop
24: ldc.i4 1
25: stloc 0
26: br.s 32 (4) [tsp: 7, til: 0]
28: ldc.i4 0
29: stloc 0
30: br.s 32 (0) [tsp: 7, til: 0]
32: ldloc 0
33: ret
```

Listing 4: Human Readable Output of Internal Method Representation

4.3.2. Upgrading method and instruction format

In case of *Tiny* method format the header is upgraded to represent a *Fat* format because we can easily overrun the limitations of *Tiny* format.

The short branch instructions (brtrue.s, br.s, bge.un.s, etc.) are converted to their long pairs (brtrue, br, bge.un, etc.) because we cannot guarantee that the relative branch lengths will remain within the numeric representation barriers after inserting some instrumentation instructions between the branch instructions and their targets.

Tiny Exception Handling Clauses are also upgraded to store offset and length values in DWORD format because the limitation of original WORD (offset) and BYTE (length) can be easily overrun after instrumentation code insertion.

4.3.3. Instrumentation code insertion

Now we have the *Token* IDs of Trace methods, queried the IL and source code level offsets and lengths of sequence points and converted the binary data to upgraded IL instruction flow. Now we examine how the methods called *DoFunc* (in Listing 1) and its pair called *DoTrace* can be parameterized and called. While DoFunc is intended to use at method enter and leave, DoTrace handles intra-function sequence points.

As we have mentioned earlier, IL instructions are able to parse themselves therefore we create a BYTE array to store binary data which can be easily parsed and stored in the same type of container where the original instructions are stored.

The parameters of the method to be called are loaded on the stack using the *ldc.i4* instruction (opcode 0x20) in order of parameters and the *Token ID* of method is given as the parameter of *call* instruction (opcode 0x28). The possible instruction (ldc.i4.1, or ldc.i4.2) at index 25 surely having a one byte opcode (0x17 or 0x18) loads 1 for enter or 2 for leave on stack respectively.

```
BYTE insertFuncInst[31];
insertFuncInst[0] = 0x20; //ldc.i4, start line
insertFuncInst[5] = 0x20; //ldc.i4, start column
insertFuncInst[10] = 0x20; //ldc.i4, end line
insertFuncInst[15] = 0x20; //ldc.i4, end column
insertFuncInst[20] = 0x20; // ldc.i4, func. id
insertFuncInst[25] = 0x0; // ldc.i4.1 or ldc.i4.2
insertFuncInst[26] = 0x28; // call
*((DWORD *)(insertFuncInst+27)) =
                    tracerDoFuncMethodTokenID;
```

Listing 3: Binary representation of trace method call

The above parameters are dynamically substituted depending on the data of the current sequence point and a unique function ID (generated by an own counter) while the function token can be preset since it is module (and not function) dependent.

In the intra-function sequence points only the data of sequence points is substituted and the thread ID is queried at each step, the function ID and other information are irrelevant here. The substituted binary data is parsed and converted to IL instructions and inserted into the beginning of the IL code container of every sequence point.

4.3.4. Recalculating offsets and lengths

Since the IL instruction flow is altered by inserting extra instructions the target offsets of branch instructions and the start offset and length properties of Exception Handling Clauses have to be recalculated.

A target offset of a branch instruction can point to the first instruction of a sequence point and can point to other than the first instruction. If the original branch target offset pointed to the first instruction of a sequence point then we change the target offset to the newly created first instruction in order to run instrumentation after jumps also. If the original branch target pointed to other then the first instruction then we leave it to target to the same instruction as before.

Any IL instruction in our representation can calculate its length, so we can easily recalculate the new offsets of IL instructions and sequence points for the branch targets also.

The offset and length properties of Exception Handling Clauses can be calculated similarly.

```
0: ldc.i4 25
5: ldc.i4 1
10: ldc.i4 25
15: ldc.i4 2
20: ldc.i4 3
25: ldc.i4 1
26: call 167772195
31: nop
32: ldc.i4 26
37: ldc.i4 3
42: ldc.i4 26
47: ldc.i4 23
52: call 167772194
57: ldarg 0
58: ldarg 1
59: clt
61: ldc.i4 0
62: ceq
64: stloc 1
65: ldloc 1
66: brtrue 165 (94)
71: ldc.i4 27
76: ldc.i4 3
81: ldc.i4 27
86: ldc.i4 4
91: call 167772194
96: nop
97: ldc.i4 28
102: ldc.i4 7
107: ldc.i4 28
112: ldc.i4 47
117: call 167772194
122: ldstr 1879048193
127: call 167772181
132: nop
133: ldc.i4 29
138: ldc.i4 7
143: ldc.i4 29
148: ldc.i4 19
153: call 167772194
158: ldc.i4 1
159: stloc 0
160: br 197 (32)
165: ldc.i4 31
170: ldc.i4 3
175: ldc.i4 31
180: ldc.i4 16
185: call 167772194
190: ldc.i4 0
191: stloc 0
192: br 197 (0)
197: ldc.i4 32
202: ldc.i4 1
207: ldc.i4 32
212: ldc.i4 2
217: ldc.i4 3
222: ldc.i4 2
223: call 167772195
228: ldloc 0
229: ret
```

Listing 5: Altered IL code of IsFirstLess method

4.3.5. Storing the instrumented method

Now we have the instrumented method represented in our data structures. The job is to store the data and IL code back in binary format following the specification. The binary data can be restored to the CLR by using the method described in 4.3.

5. COMPARISON OF METHODS AND TEST RESULT

In the previous sections we have presented two different methods for generating runtime execution trace of .NET-based applications.

None of the methods require us to modify the applications being tested. Both methods can be accomplished to produce trace information about the value of accessed variables of any type, and identify reference variables. With the help of the Debugger, reference variables can be identified by their Object Id, but obtaining this Id requires many time consuming operations [Stall]. Using the Profiler's IL code rewriting capabilities it is also possible to identify reference variables, and much faster than with the Debugger. A value type variable is always identifiable by the sequence point occurrence it was created in.

The Debugger is unable to notify us about thread switches and the step-in operation is unable to jump through threads therefore it is not possible to handle multithreaded applications. To the contrary, using the Profiler we are able to log the thread's ID at every sequence point of the application.

In order to make the Debugger work we have to attach it to the process we intend to instrument. To use the Profiler, it is required to register it as a COM component using the *regsvr32* command and set two environment variables in the process, user or system context to enable the Profiler in that context. Set *Cor_Enable_Profiling* to *0x1* and *Cor_Profiler* to the GUID or ProgID of our object implementing the *ICorProfilerCallback2* interface.

We demonstrate the performance of the methods through four applications. The first two use only few class library calls so they are intended to measure the pure performance. The third application uses much more but very short, while the last one uses many and long class library calls.

The character of the four applications:

1. Counter is a simple application that calculates the sum of numbers from 1 to 10000 and prints a dot at each step on the screen by implementing the addition in a separate function and uses only few class library calls, but a lot of integer operations which are implemented by native IL instructions.
2. ITextSharp is an open source PDF library. In our test we created a basic PDF document. It uses very few class library calls and a lot of string operations which are implemented by native IL instructions.
3. DiskReporter recursively walks the directory tree from a previously specified path and creates an XML report. In our test 3141 directories and 12257 files were enumerated. It uses more, but short library calls (xml node and attribute operations, file property query).
4. Mohican is a small HTTP server using multiple threads for serving requests. In our test Mohican served a 1.3MB HTML document referencing 20 different pictures. It uses many and long class library calls (mainly network and file access).

App. name	Normal run	Debugger trace	Profiler trace	No. of SPs
Counter	00:00.17	01:53:92	00:01.34	110,034
ITextSharp	00:01:02	98:11.32	02:33:50	2,825,242
Disk-Reporter	00:05.46	24:04.42	00:11.76	316,196
Mohican	00:01.37	n/a	00:01.89	175,434

Table 2: Test results

Table 2 shows the performance comparison of the normal application run, the run under the control of the Debugger and the Profiler in mm:ss.ii format. The last column contains the number of source code statements executed.

It can be seen that applications containing few class library calls perform poor under the control of both the Debugger and the Profiler, while applications containing many class library calls perform better.

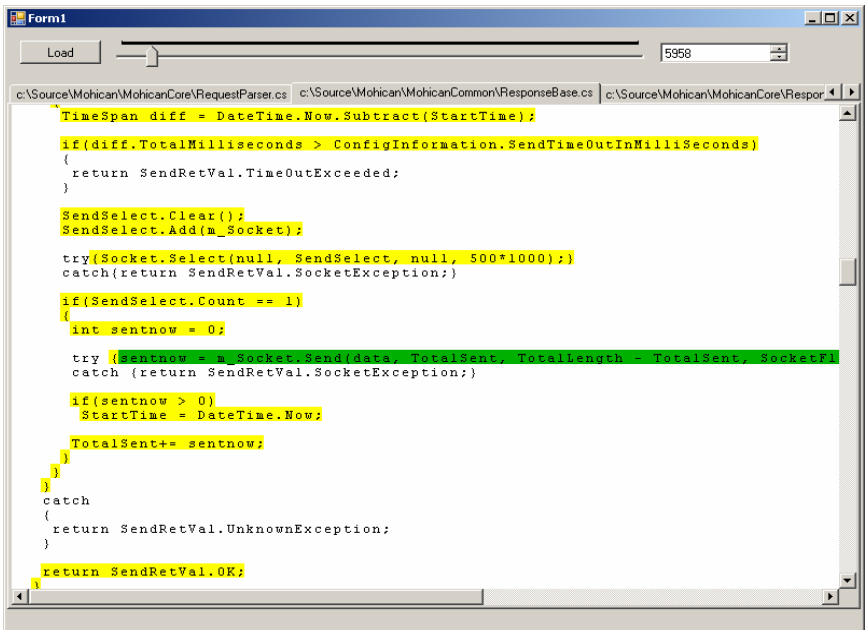

Figure 3: Visualizing the trace

Applications containing long class library calls (like any real world enterprise application) perform well under the control of the Profiler. Unfortunately the Debugger could not be tested (because of multithreading).

The runtime trace generated by the Profiler can be visualized using a Winform application as shown in Figure 3 (the trace of Mohican). The code fragment in green (darker) shows the statement executed at an arbitrary step of the application. Statements in yellow (lighter) have already been executed, while white statements have not yet been traversed.

6. CONCLUSION AND FURTHER WORK

In this paper we have shown how to utilize the .NET Debugging and Profiling Infrastructure to generate runtime execution trace of large applications and analyzed both method using programs of different characteristic. We can conclude that although the method based on the Debugger is easier to implement, the Profiler is much more suitable for tracing large scale, multithreaded applications.

Therefore, we plan to advance on the Profiler way. The first and most important thing is to extend our framework to identify variables in the order as local variables, method arguments and class variables appear. We can insert instrumentation code after any variable load and before any variable store operation. The on-stack-top variables can be duplicated by the *dup* IL instruction in order to consume them in the parameter of a trace method call.

There are some language elements and CLR features which we currently do not support like exceptions, nested classes, anonymous methods, generic types and methods, application domains.

7. REFERENCES

[Agr91a] H. Agrawal and J. R. Horgan. Dynamic program slicing. In SIGPLAN Notices No. 6, pages 246-256, 1990.

[Bes01a] Á. Beszédes, T. Gergely, Zs. M. Szabó, J. Csirik, T. Gyimóthy. Dynamic slicing method for maintenance of large C programs, CSMR 2001, pages 105-113.

[ECMA] ECMA C# and Common Language Infrastructure Standards http://msdn.microsoft.com/netframework/ecma/

[Mar03a] K. Maruyama, M. Terada, Timestamp Based Execution Control for C and Java Programs, AADEBUG, 2003

[Mik03] A. Mikunov, Rewrite MSIL Code on the Fly with the .NET Framework Profiling API, MSDN magazine, issue September 2003, http://msdn.microsoft.com/msdnmag/issues/03/09/NETProfilingAPI/

[Póc05] K. Pócza, M. Biczó, Z. Porkoláb. Cross-language Program Slicing in the .NET Framework, Journal of .NET Technologies, 2005

[Stall] Mike Stall's .NET Debugging Blog, http://blogs.msdn.com/jmstall/, 2004-2006

[Tip95a] F. Tip, A survey of program slicing techniques. Journal of Programming Languages, 3(3):121-189, Sept. 1995.

[Zha03a] X. Zhang, R. Gupta, Y. Zhang. Precise dynamic slicing algorithms. Proc. International Conference on Software Engineering, pages 319-329, 2003

A Microsoft .NET Front-End for GCC

Martin v. Löwis
Hasso-Plattner-Institut
für Softwaresystemtechnik GmbH
Postfach 900460
+49 331 5509 239
Martin.vonLoewis@hpi.uni-potsdam.de

Jan Möller
Hasso-Plattner-Institut
für Softwaresystemtechnik GmbH
Postfach 900460
Jan.Moeller@hpi.uni-potsdam.de

ABSTRACT

In the past, embedded systems developers have been severely constrained in their choice of programming languages. Recent advancements in processing power and memory availability allow for new techniques. We present an extension to the GNU Compiler Collection (GCC) that offers the expressiveness of all Microsoft .NET languages to embedded systems.

Keywords

Common Intermediate Language, GNU Compiler Collection, GCC.

1. INTRODUCTION

Embedded systems are known for the severe resource constraints in terms of memory size and clock speed. For that reason, developers traditionally use assembler language and C for such systems [Bar99]. Compared to current desktop and server programming languages such as Java, C#, Python, Visual Basic, and others, the typical development environment is tedious to use, and the development is less productive.

There are two primary aspects of the "desktop" programming languages that we consider interesting for embedded developers as well: object-orientation and safety. With object-orientation, the software may become more maintainable, as the encapsulation mechanisms allow for better modularization and abstraction.

By "safety", we refer to the reliability aspects that are typically associated with interpreters: the run-time system of the language will make sure that invalid operations (such as out-of-bounds accesses to arrays) cause a well-defined program termination (typically through an exception), instead of causing undefined behavior (such as memory corruption). Safe programming languages reduce the number of bugs that remain in the software after testing, as errors are reliably detected. They also simplify the process of locating the source of a bug, as the error is often detected right after it occurred.

Unfortunately, both object-orientation and safety come at significant run-time cost. Interpreters execute program code much slower than similar compiled programs. Alternatively, just-in-time compilation is used to speed-up execution [Kra98]. Unfortunately, just-in-time compilation is itself expensive and causes unpredictable run-time behavior. Furthermore, a just-in-time compiler needs to be developed for each new target architecture.

As an alternative, we present an approach which allows static compilation of .NET programs for embedded targets. We briefly discuss different aspects of this solution in the remainder of this paper.

2. GCC

The GNU Compiler Collection integrates different programming languages (C, C++, Java, Ada, …) for various microprocessor architectures [GS04]. Among the supported targets are many desktop and embedded processors; GCC is known for relatively easy extensibility to new architectures [Sta95]. While it originally focused on the C language only, it has recently been extended to object-oriented and safe languages, such as Java [Bot97].

In GCC, the source code of the input language is transformed into an intermediate representation[1], which is then processed in optimization passes. The result of the compilation is then output as an assembler source code file for the target machine. This assembler file is processed with assemblers, loaders, etc. for the target system to produce an executable program.

The design of GCC is engineered towards extensibility. Support for new microprocessors can

[1] More precisely, there are two internal representations: the *tree* structure, and the Register Transfer Language (RTL).

 ISBN 80-86943-11-9

be added relatively easy by describing the processor in a *machine definition*. Using this machine definition, the compiler can convert the internal representation (RTL) into assembler code of the target system. This assembler code is then further processed in an assembler to object files, and eventually combined with a linker into executable files and libraries.

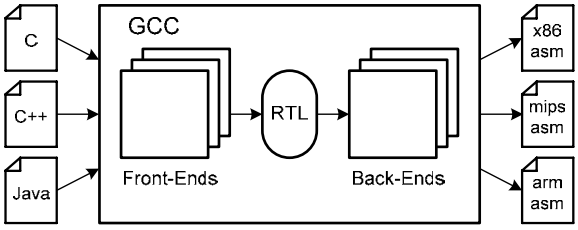

Figure 1. GCC Architecture

In the last few years, the focus in extensibility moved towards integration of new languages into GCC, and into integration of new optimization algorithms. To support a new front-end, several aspects have to be considered in the compiler framework:

- Integration of the front-end into the build process,
- Integration of input and output file handling,
- Management of symbol tables,
- Representation of the actual code of the program,
- Debugger support, and
- Optimization.

For each of these aspects, GCC defines interfaces which a new front-end must use. For example, to add a new front-end to the build process of the compiler itself, one must create a subdirectory in the source tree, and add files such as Make-lang.in and config-lang.in. This will automatically result in another option for the GCC --enable-languages switch, so that an administrator can enable or disable the build of this front-end. Likewise, by adding a file lang.opt to the source directory, the GCC command line option processing framework will automatically support language-specific compiler options.

To integrate a front-end into the actual processing flow in the compiler, the compiler framework defines certain hook functions which might be filled out by the front-end. For example, the compiler framework will invoke a parser call-back, which then should process all input files for the source language.

To support symbol tables and code representation uniformly across languages, GCC defines a set of data structures and utility functions. In the parser, the front-end will use the utility functions to build a program representation, which is then passed to the back-end passes of the compiler. As an example, the function build_decl is used to create a function declaration object. This object is enriched, through further function calls, with the actual body of the function. Eventually, rest_of_compilation must be called, which performs the optimization (if requested), and output the assembler code.

Both optimization and debugger support in the compiler need the help from both the front-end and the back-end. The front-end needs to annotate the tree with programming-level knowledge (e.g. whether the address of an object was ever taken), and the back-end needs to specify how many cycles each instruction consumes, so that the instruction scheduler can pick the most efficient of several alternative instruction sequences.

3. The CIL front-end

The Common Intermediate Language (CIL) [ECM02a] is a platform-independent representation of object-oriented programs. It was designed to support a wide range of languages. It focuses on the C# language [ECM02b], but also supports variants of Java, C++, Visual Basic, Eiffel, and other languages. CIL builds the core of the Microsoft .NET environment.

Our front-end transforms CIL code into the internal representation, which GCC then optimizes and outputs for the target system. Similar to the Java front-end, we use symbolic execution to convert the stack machine that CIL assumes into the tree structures of GCC.

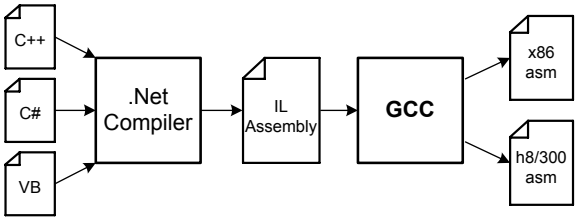

Figure 2. Integration of the Common Intermediate Language into GCC

Unlike the Java front-end, we have no plans to process source code directly. Instead, we use the IL library from the DotGNU Portable.NET framework [Dot05] to load IL assembly files into memory, and traverse the meta-data structures in the assembly. As a result, we do not have a traditional parser in our front-end. Instead, we define our own traversal algorithm, which processes all classes in the assembly in sequence. For each class, we build the layout of the class and the structure of the virtual method table, and emit code for each method.

The IL front-end can, in principle, support all aspects of the semantics of .NET programs, except for the dynamic loading of additional assemblies which had

not been compiled through this front-end. In the current implementation, only a subset of the .NET concepts is available; see section 5 for details.

4. Target Systems

In principle, it is possible to support all features of the .NET platform that don't require dynamic insertion of behavior. That is, all instructions of the intermediate language can be converted into sequences of assembler instructions of the target system. Through generation of data structures into the resulting assembler code, introspection of objects is possible, using the standard APIs. Even dynamic loading of assemblies is possible, as long as the assembly to be loaded was compiled using GCC in advance.

For the remaining features, we plan to support interoperability with the Mono software [DB04]. To achieve an integration of Mono, we need to use the same application binary interface (ABI) that mono uses, with respect to calling conventions, and representation of meta-data in memory.

At the same time, we also like to target embedded systems. At the moment, our primary target is the Lego Mindstorms hardware [Sat02], which uses the Renesas H8/300 processor [Ren03]. On this system, memory is limited. For our .NET implementation, this means primarily that we have to be very selective in the subset of the .NET library that we can support – the entire platform library just won't fit into 32k of main memory. In this environment, we may also have to accept further limitations. However, depending on the application's needs, we believe that all features of the virtual machine can be supported. The more challenging features are floating point computations (which require emulation in software, as the chip has no hardware floating point support), exception handling, and garbage collection. At this point, we cannot yet predict what costs in terms of memory and processor cycles these features will require.

In addition to the Lego Mindstorms, we also target Windows CE; in particular CE PC.

5. Current Status

Currently, only a small fraction of the CIL features are supported, namely

- primitive data types (bool, byte, short, int, float, double)
- classes, including static and instance attributes and properties, as well as inheritance,
- static and instance methods, including parameters, local variables, and constructors,
- arrays and strings,
- delegates
- arithmetic operations, and
- control flow operations (conditional and unconditional branch instructions).

Using this subset, we have been able to develop small control programs for the Lego Mindstorms platform.

On the Windows CE system, we were able to create control programs which meet hard real-time constraints.

Work to provide additional features, such as interfaces, and exception handling, is in progress. Our current implementation is available from http://www.dcl.hpi.uni-potsdam.de/research/lego.NET/release.htm.

6. Conformance

This implementations of the CLI aims to comply with the Kernel Profile of the ECMA specification 335. Support for the Compact Profile would be largely possible through integration of library implementations, such as the ones provided with Mono. To support the Compact Profile, the biggest challenge is the support for reflection, in particular, for the dynamic loading of assemblies. For that to work, a byte code interpreter or just-in-time compiler is needed in addition to the statically-compiled code.

With respect to the Kernel Profile as specified in [ECM02c], section 4.1 (Features Excluded From Kernel Profile), our implementation has the following properties:

- Floating Point is supported if the target processor supports it or an emulation library is available.
- Non-vector Arrays are not currently supported; adding support would be straight-forward, though.
- Reflection is currently not supported, but work to add support for reflection is in progress. Due to the overhead of reflection, support for reflection will be selectable on a per-application basis. See above for a discussion of dynamic assembly loading.
- Application domains are currently not supported; however, concepts needed to support them (e.g. per-appdomain static class variables) are already implemented.
- Remoting is not supported; no support is planned.
- Varargs functions, frame growth, and filtered exceptions are currently not supported; no support is planned. Code that tries to use these features is rejected in the compiler

As shown in section 5, many features of the CLI are currently unimplemented. Most notably, there is no support for verification: We assume that all assemblies passed to the compiler are verifiable. However, at this point, we don't foresee any aspects of the CLI metadata or instruction semantics that are unsuitable for our implementation approach. For example, verification would be implemented most naturally in the compiler itself, causing no run-time overhead.

7. Related Work

Cygnus Solutions (now Redhat) has developed a Java front-end [GCJ05], supporting both Java source code and byte code. The CIL front-end has taken much inspiration from the latter.

Microsoft currently develops the Phoenix framework [Lef04], which appears to be similar in architecture to GCC, and also appears to contain a .NET front-end. Very little information about Phoenix has been published so far.

8. REFERENCES

[Bar99] M. Barr. Programming Embedded Systems in C and C++. O'Reilly, 1999.

[Bot97] P. Bothner. A Gcc-based Java implementation. IEEE Compcon'97.

[DB04] E. Dumbill, N.M. Bornstein. Mono: A Developers Notebook. O'Reilly, 2004.

[Dot05] DotGNU Portable.NET. http://www.dotgnu.org

[ECM02a] ECMA-335. Common Language Infrastructure, Partition III: CLI Instruction Set. Dec. 2002.

[ECM02b] ECMA-334. C# Language Specification. Dec. 2002.

[ECM02c] ECMA-335. Common Language Infrastructure, Partition IV: Library. Dec. 2002.

[GCJ05] The GNU Compiler for the Java™ Programming Language. http://gcc.gnu.org/java

[GS04] B.J. Gough, R.M. Stallman(Forword). An Introduction to GCC. Network Theory Ltd, 2004.

[Kra98] A. Krall. Efficient JavaVM Just-in-Time Compilation. PACT, 1998.

[Lef04] J. Lefor. Phoenix as a Tool in Research and Instruction. July, 2004.

[Ren03] Renesas Technology Corp..H8/300 Programming Manual. 2003.

[Sat05] J. Sato. Jin Sato's Lego Mindstorms. No Starch Press, San Francisco, 2002.

[Sta95] R.M. Stallman. Using and Porting GNU CC. Free Software Foundation, 1995.

Architecture and Design of Customer Support System using Microsoft .NET technologies

Nikolay Pavlov
PU Paisii Hilendarski
236 Bulgaria Blvd.
Bulgaria, Plovdiv 4003
npavlov@kodar.net

Asen Rahnev
PU Paisii Hilendarski
236 Bulgaria Blvd.
Bulgaria, Plovdiv 4003
assen@pu.acad.bg

ABSTRACT

This paper describes the four-tiered architecture, technologies, functionality and electronic services for the participants in the process of customer support with a software system for customer support – Integrated Help-Desk Center (IHDC), based on an object-oriented framework for development of distributed applications.

There exist multiple solutions for customer support management. Many of them do not provide services to end-clients, do not support vertical organizational structure or lack relevant multi-language support for international clients.

The participants in the process of customer support in IHDC are: clients, local partners, local branches, central office and development department. Multilingual support is provided to enable operation over different counties. IHDC consists of: Customer Relationship Management; System for registering and management of tickets; Management of application known issues; Management of application updates.

The system is developed with Microsoft .NET Framework. Its infrastructure is build upon Microsoft Windows Server 2003, Microsoft SQL Server 2000 and Microsoft Information Server. The four tiers are: database server, application server, functional objects and thin client interface – Windows-based and browser-based.

Keywords

Four-tiered architecture, object-oriented framework, customer support, Microsoft .NET Framework.

1. INTRODUCTION

High-quality customer support services have been always identified as an important element of the overall package of software services for customers, crucial for the mission of every software company. Therefore, successful software companies strive to provide increasingly higher quality services to their customers, and seek ways to achieve this by automating and optimizing their processes. One of the necessary elements is to have a centralized repository of all customer-related issues, and to have an established process for handling those issues, to ensure that no problem is neglected or processed inadequately. The dynamic of the modern world creates new economic and cultural environments, thus putting further difficulties before companies, which operate across country boundaries. Some of those problems include multi-level company hierarchical structures and multi-national fields of operation.

There exists a wide range of software solutions for customer support. Many of them are not suited for the latest requirements for quality customer support, because they lack support of certain features. Common disadvantages are:

- Support of horizontal company structure only.
- No multi-language support.
- Insufficient integration with existing office packages.
- Insufficient functionality for Customer Relation Management (CRM), or integration with third-party CRM systems.
- Cannot operate in a distributed environment, for example - over the Internet.
- Do not support built-in declarations of hours and costs.

.NET Technologies 2006

This paper describes a Customer-Support System (CSS), which is aimed to enable multi-national companies provide high-quality support services to their customers by offering an affordable and flexible solution, which overcomes the limitations stated above.

A major advantage of the proposed CSS is the automated translation-request system. This system monitors and assigns translation tasks to appointed personnel, when information is crossing language boundaries within the organization. For example, a call from the client needs attention from a higher level of support, where responsible employees do not speak the client's language. In this case the system assigns all the information, as provided by the client, to a translator. Also, all information, which is to be communicated back to the client, is translated before being made available to the client.

The architecture of the Customer Support System is four-tiered and is based on an object-oriented framework for development of distributed applications. The system infrastructure and is build upon Microsoft Windows Server 2003, Microsoft SQL Server 2000 and Microsoft Information Server. The four tiers are: database server, application server, functional objects and thin client. The system is implemented using technologies, based on Microsoft .NET Framework.

2. ROLES IN CSS

The design of CSS identifies the following roles for participants in the process of customer support:

- Central office (management).
- Branches
- Partners
- Development
- Clients

Central office represents the management of the company, providing the services. This role performs the highest level of support and supervision of the performance of all other levels. Central office is the only instance, which communicates with development, thus providing a centralized and controlled information flow towards development.

Branches are head offices for countries. There is only one branch per country. Branches provide support for all customers from the corresponding country. Branches also serve as a bridge between clients and the Central Office, and enhance communication flow to and from the Central Office by providing translations whether necessary.

Partners are agents within one country, and subordinates of the corresponding branch. Partners provide first level of support, education and other services like installations, configuration on-site, demonstrations. They communicate most actively with existing clients and potential clients.

Development is a department, which provides software services like bug fixing, product extensions, new versions, etc. Issues, which cannot be solved otherwise, are ultimately sent to development by the Central Office. Development never has a direct contact with clients and other levels.

The following diagram represents the structure of the roles, as defined by CSS.

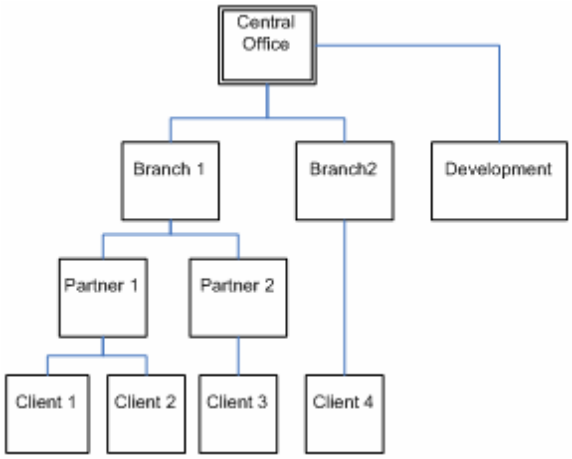

Figure1. Roles Structure in CSS

Figure 1 gives an example of the relations between different roles in CSS. The direct link between Branch 2 and Client 4 means that the role of a Partner is not required and can be skipped in certain cases. For example, in a relatively small market partners may not contribute to the efficiency of the organization and, therefore, will not be established.

Clients are the end-customers of the organization. They contact with the organization always via a Partner or a Local Branch.

3. SERVICES FOR PARTICIPANTS IN CSS

Services for Clients

Clients in CSS are provided with a browser-based Internet application – thin client, with which they can do the following:

- Enter new issues (tickets).
- Attach documents and files to tickets.
- Review status of tickets.
- Provide additional information on existing tickets on request.
- Close tickets when solved.
- Review existing known issues.
- Review new releases, fixes and release notes.

Tickets are entered under pre-defined categories, and with different urgency level. Clients enter a description of their problem / question, and can attach an external file – office document, screenshot, etc to a ticket. Clients can check existing known issues and their solutions to determine if there is a ready answer to their problem. Clients can monitor the progress on each of their tickets, and provide additional information if the customer support asks for it. Certain actions the customer support produce an e-mail to the client to emphasize on issues that would need quick attention. Clients can also see all new releases, patches and fixes, to determine if they should upgrade their software.

Services for Partners

Partners use the desktop-based client of CSS. Partners can see all tickets, entered by their clients, and take actions on them. Partners can provide solutions, request additional information, or send tickets to branches for higher level of support if the solution is beyond their competency.

Partners are provided with access to the database with all their clients, including contact and address information, plus all application releases, fixes, and special consultancy documentation, which is provided by the central office.

Services for Branches

Branches use the desktop-based client of CSS. They see all tickets, entered by clients in the corresponding country. In this way branches can monitor the performance of all their subordinate partners and take necessary measures. Branches see tickets, received directly from their clients, and tickets, for which partners cannot provide a timely solution. Branches can provide solutions, request additional information, or send tickets to the central office, if the solution is beyond their competency. Branches have tools to provide translated information to the central office, if necessary.

Just like partners, branches are provided with access to the database with all their clients, including contact and address information, plus all application releases, fixes, and special consultancy documentation, which is provided by the central office.

Additionally, branches have access to a database with all their partners.

Services for Central Office

The Central Office uses the desktop-based client of CSS. They see all tickets, and thus can monitor the performance at any sub-level. There is an automated system, which automatically escalates tickets, which are not processed timely by the responsible sub-level.

The Central office has all necessary facilities to provide solutions and request additional information. It can also assign tickets to development department, if the problem cannot be solved with other means. There are tools for on-line discussions and solution design. Other tools exist for authoring of release notes, and creating known issues from similar tickets. The Central office can also monitor the performance of the development department.

There is a special tool, which provides summarized information about the status at any level within the organization, with special emphasis on overdue work, or work, approaching its designated deadline. This tool enables management to quickly spot problematic nodes in the company structure and take the necessary measures to resolve the issues.

When a client is updated to a new version of the application, and the update is actually a new application, not simply an upgrade of the old one, all its existing information has to be converted. This includes all tickets, installation information, etc. There is a special tool, which facilitates this process. It archives all data, relevant to the previous version, and creates the necessary structures for the new application.

Services for Developers

Development department uses the desktop-based client of CSS. They see only tickets, assigned to development by the central office. Developers can use the tool for on-line discussions to receive logged additional information. Developers report their work to the Central Office, which is responsible for testing their work before delivering the solution to the customer, and for authoring the necessary release notes for both clients and other levels of support. Development department is isolated from clients, and direct communication between these two parties is not allowed.

Other Services

CSS contains a system for automatic escalation of tickets. It monitors if a ticket is not processed timely at any level below the central office. In such a case, the ticket is escalated automatically to the higher level. This system also monitors ticket deadlines. If a deadline is approaching, the system sends notifications by e-mail to the responsible employees at the current level of support and the central office.

There is and integrated Customer Relation Management system, available to partners, branches and the central office, is. It provides an extensible

data structure for storing client-related commercial information. This system focuses on development of new clients, and management of sales.

Another integral part of CSS is the system for registration of visit reports. Visit reports summarize all agreements and arrangements, negotiated during meetings between clients, company personnel and representatives, and external consultants. Information includes all participants in meetings. The system prepares report documents for each meeting and sends those by e-mail to all participants.

4. PROCESSES IN CSS

CSS defines a schema of sequential processes for handling tickets. It goes in following steps:

1. Ticket is entered by client.
2. The appropriate partner sees the ticket. The partner can solve the problem, and notify the client to approve the solution, or, escalate the ticket to the branch. If no partner is available, this step is skipped.
3. The branch sees the ticket. They can provide a solution, or escalate the ticket to the central office, if they cannot handle the ticket. If a solution is found, the branch can directly implement the solution, or send the ticket back to the partner for implementation. Before escalating the ticket, the branch should translate the ticket into the language of the central office, if necessary.
4. Ticket is received at the central office. If the central office can provide a solution, the ticket is sent back to the branch for localization and implementation. If the problem requires programming, the ticket is assigned to development with additional description and translation, where necessary.
5. Development department receives a ticket with a detailed description, and scheduled dates for start and completion of work on every ticket. When work is completed, the ticket is sent back to the central office for approval.
6. A completed ticket is sent back to the central office. They test the solution and depending on the result, can sent it to the branch for implementation, or revert it back to development.

At every level, except for development, support can request client to send more information, suspending the ticket. The ticket is reopened automatically, when the client gives an answer.

When a ticket is solved, the client is notified and the ticket is suspended. It is the client who does close the ticket.

Suspended tickets are automatically reopened if no action is taken on them for a specified period of time. This logic prevents tickets from being forgotten and stalled.

5. ARCHITECTURE

CSS is developed through the use of an object-oriented framework for distributed business applications. This section describes the key features of the framework.

Four-tier Application Architecture

The architecture of the framework is presented on figure 2.

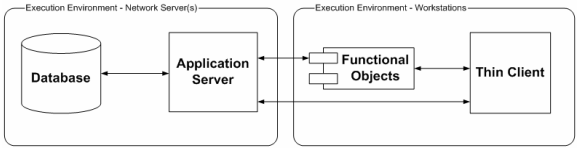

Figure 2: Four-tier architecture

The four-tiers (Figure 2) are: database server, application server, functional objects, and client.

The database is responsible for storing the application data, as well as the internal framework data - application dictionary, security, and customer preferences. Application and framework data are stored within one logical database, which improves encapsulation.

The application server is an intermediate layer between the database and the functional objects. It does not implement business logic; in stead, it provides a number of services to the other layers – services for data access and modification, security services, communication services and system services for initialization, multi-language support and maintenance. It is the only layer that communicates directly with the database server, which enables development of applications for various database platforms by changing only the application server. Access to the database is realized with ADO.NET. The application server is multithreaded – each client is served by a separate thread, which improves the performance on SMP and Hyper-threading systems.

The functional objects are specially designed program modules, which are integrated at run-time within the process of the client application. They provide the functional core of the client application. The functional objects software components, which accomplish a certain task. They realize the business logic of the application and a part of the user interface.

The client application is an environment for execution of functional objects under a common user interface. It provides a number of services for:

- Common graphical user interface
- Load, execute and release functional objects
- Translate all user interface text items (commands, menus, and static texts) towards the active functional object
- Data exchange between running functional objects

The choice of desktop-based architecture for the client application is motivated by the significantly richer features for building of the graphical user interface (GUI), which desktop-based GUI technologies present. The strong support of various infrastructures for distributed applications in Microsoft .NET Framework enables desktop-based applications to access resources and components over local networks and over the Internet - .NET Remoting, SOAP (Simple Object Access Protocol) based web-services, TCP/IP sockets, robust COM (Component Object Model) integration. These are strong foundations for easily building applications with fully-featured, convenient and aesthetic user-interface, while not being limited to client-server application architecture.

Another significant benefit of desktop-based applications is that they make it possible to implement "push" callbacks – events. Such events may be triggered when a user modifies a record in the database, thus notifying all the users, working in the same logical sub-domain, that a relevant data modification has taken place.

The application framework, employed to develop CSS, is using .NET Remoting to realize the communication between the client and the server application. .NET Remoting may be used over the Internet, though its callback features are not suitable for wide-area networks, due to technical and security restrictions. Therefore the application framework implements callbacks via proprietary TCP/IP connections, established by the client application to the application server. This is necessary to resolve issues with firewalls and NAT (network address translation), which "hide' the clients from the server.

Scaling

The architecture of the framework defines two execution environments – servers and workstations. It is possible to run the database and application server on a single server, or scale the environment and have database(s) and the application server running on different server computers. For example, one way to scale up is to run the application server and the database with system data on one box, and the application user data on a separate box. Further scaling is possible by using more than one application servers, each running on a separate computer. This, however, has its drawback – callbacks (events) are not possible across multiple application servers. Proper organization of work may overcome the negative effect, because in large organizations each department has its own area of operation and it is less critical for immediate view of all data modifications, made across the organization. Additionally, in stead of using callbacks, the client is able to use "pull" technology to acquire events from the application server.

Standard Functional Objects

Targeting enhanced code reuse, the framework includes a set of functional objects, which implement security, data browsing and searching, data modifications, reporting, document integration with Microsoft Office and other external documents, and database integrity administration. They are versatile and function according to the specific definitions in the application dictionary.

Data overview: browsing, and sorting data, search for data, filtering data on user-selected criteria. Data is displayed in table format, with options for additional information in addition to the table. Searching and filter can be performed on fields from the table, as well as on other related data – both one-to-many and many-to-many relationships are supported.

Data entry and modification: entry of data, with built-in facilities for client-level data validation. Additional services include copy of data, and edit of multiple records with a single operation.

Reporting services: preview and print of reports. Custom reports can be created on any level, with a What-You-See-Is-What-You-Get (WYSIWYG) editor. Reports can be exported into popular formats like Abode Portable Document Format (PFD), Microsoft Excel (XLS), and HTML.

Integration with Microsoft Office (Microsoft Word): creation, storage and retrieval of Microsoft Word documents, which contain data from the database of the client application. Active links between the documents and the data is maintained. Microsoft Word document files are stored on an especially designated storage folder, which allows access to them even in case of system failure, and provides an organized depository of office's files.

Attachment of external documents (files) to existing application data. An essential part is the descriptive definitions of relevant application data, which

external files can be attached to. Links between application data and the files attached are created. Attached files are stored on an especially designated storage folder, which allows access to them even in case of system failure, and provides an organized depository of office's files.

Security management: application administrators can assign access policies to application users and user groups. There are two types of access policies: on application level, and on data level. Application level security policies determine which screens and functions are available to a user, while data level access policies determine which data is accessible by a user.

Data Integrity management: application administrators can overview all modifications, made by users, and take necessary actions to sustain the logical integrity of the application data.

Those functional objects allow development with minimum, even no code pre-compilation by using the application dictionary. The application dictionary is the "heart" of the framework – it is a centralized repository of logical, functional and business definitions. It describes the hierarchical structure of the application, the user interface – icons, menus, toolbars and forms, and the access security roles on both application and data level. It contains all parameter definitions for the functional objects and thus determines their behavior in every part of the application.

Application dictionary is created in a special descriptive language, based on Extensible Markup Language (XML). The application dictionary is stored in the database of the application and is always interpreted on application startup. The client application parses only the structure, to build the menus and screens, and the security policies for the current user. Functional object parse their designated parameters on loading.

An integral part of the framework is the special tool for authoring of application dictionary contents. It features a schematic presentation of the application structure, plus syntax-highlight editor for the parameters. Authoring of application dictionaries requires understanding of the client database structure, Structured Query Language (SQL), and of the specifics of parameter definition schemas for every functional object used. As a result, application development and support can be performed by non-programmers.

Multi-language support is handled with a tool, which enables the application administrator to translate all text items in the application into virtually any number of languages. It provides convenience facilities: incremental searching, filters for not-translated items, searching for similar translations, etc

CSS is developed on Microsoft .NET Framework. The communication between the database server and the application server is done via ADO.NET. The communication between the application server and the functional objects and the thin client is done via .NET Remoting over TCP/IP. Client front-end is realized as a ASP.NET, installed on Microsoft Information Server; it uses .NET Remoting to communicate with the application server.

Microsoft SQL Server 2000 is used as a relational database server for CSS.

6. REFERENCES

[1] Object-Oriented Application Frameworks, Fayad M., Schmidt D., Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997

[2] Ralph Johnson and Brian Foote, "Designing Reusable Classes", Journal of Object-Oriented Programming. SIGS, 1, 5 (June/July. 1988), 22-35

[3] Pavlov N., Rahnev A., Framework for Application Development, Scientific works of Plovdiv University, Bulgaria, vol. 35, book 3 - Mathematics, 2006

Design and Implementation of a Grid Architecture over an Agent-Based Framework

Christian Vecchiola, Alberto Grosso, Roberto Podestà, Antonio Boccalatte
DIST – University of Genoa
Via Opera Pia 13
16142, Genova, Italy
{christian, agrosso, ropode, nino}@dist.unige.it

ABSTRACT

Agent based programming presents several features appearing to be interesting for Grid and distributed computing needs. The typical environment required by Grid computing is complex, heterogeneous, and highly dynamic. The autonomous and flexible behavior provided by software agents meets various Grid requirements. In this paper we present the design and the implementation of a Grid architecture built over an agent based framework called AgentService. In this work we highlight the advantages in using the services of an agent oriented framework to develop a Grid application.

Keywords

Agent Mobility, Load Balancing Policy, Agent Framework, Grid computing

1. INTRODUCTION

Resource sharing through the Internet has become in the last years a paramount instrument for scientists, not only because it offers great advantages in distributed computing, but also because data sharing is becoming more and more useful in many scientific fields. Resources can be classified in three different groups: data, services, and computational power. By following this classification we can distinguish three types of grids [Fos01a]. Data Grids manage huge collections of geographically distributed data, which can be generated in many different ways, for example data streams are daily sent from satellites for weather forecast and climatic changes analysis; large collections of data generated from scientific experiments allow geographically distributed researchers to collaborate to the same research project. Service Grids provide services that could not be obtained from a single platform: for example streaming multimedia services or collaborative applications. Computational Grids provide the aggregate power of a collection of processors spread over the network as a unique, meta-computer.

In general, grid computing system are intended to replicate in the computing world the notion of a distribution grid fostered by utility networks such as the electrical power grid. In the vision of grid computing, computational power, memory, and disk space should be obtained “on demand” from a network of “suppliers”, potentially belonging to the entire Internet.

In the last decade, distributed high performance computing has been built mainly on cluster computing systems where the communication among the different components of an application is performed using the message passing model implemented by systems such as MPI [MPI] and PVM [Gei94a]. Current trends in the grid community aim at providing frameworks not more strictly tied to the classical parallel computing programming model. However, is very hard to migrate this model to a dynamically changing environment such as the Internet, thus, in order to cope with the new challenges, a more structured, service and object oriented approach has to be adopted. The evolution toward a heterogeneous, dynamic, distributed over multiple domains environment has brought to the definition of the Open Grid Service Architecture [Fos02a] (OGSA), which proposes the convergence between grid computing and Web Services technologies in order to get over the classical parallel programming paradigm. The main, world-wide known Grid project, namely the Globus Toolkit [Fos05a], in its latest release implements the OGSA

.NET Technologies 2006

specification, and leverages on Web Services technologies and on the most widely known Internet standards. With this choice Globus is able to make available cluster-based high performance computing services to simple clients and end users without a specific parallel programming know-how too.

Moreover, starting from analogue considerations was conceived the Alchemi project [Luth05a]. Its authors aim at involve into the grid community the unused computational power provided by the almost ubiquitous Windows-based desktops. The Alchemi platform is a .Net-based framework providing the runtime machinery and the programming environment required to construct enterprise/desktop grids and to develop grid applications. Alchemi is able to interface with a Globus grid too, leveraging on a Web Services interface. Another effort trying to port the grid computing towards an object oriented programming model is the H20 [H20] project. This project provides a platform independent Java-based framework able to build meta-computing application leveraging on various remote method invocation protocol, such as SOAP, Java RMI, and TCP-based RPC [Kur03a].

It is our opinion that moving from the parallel computing programming model to a services and object oriented model, built on top of widely known technologies, can not be considered the last step of the Grid programming paradigm evolution. For example, Agent technology [Jen99a] and the agent programming model could be very useful to build virtual, highly dynamic, and distributed environment such as the context where typically operates a Grid framework. Agents are autonomous software entities with some level of intelligence; agents work better if they belong to a community such as a multi-agent system (MAS) [Wei99a]. Agents act in a distributed manner, cooperate, compete, and negotiate to solve a problem or to perform a task. These features make the agents an interesting technology to implement Grid infrastructures.

In this paper we present the design and the implementation of a Grid Computing architecture over an agent based framework. The Grid infrastructure has been built leveraging on the capabilities of the .Net based AgentService programming platform [Boc04a].

The paper is structured as follows: in section 2 we provide a brief overview on agent technology and multi-agent systems, and we describe the related synergy with grid computing; section 3 includes the description of AgentService programming platform; in section 4 we provide a detailed description of our agent-based Grid Computing architecture; section 5 shows a case study where our framework has been adopted; finally in section 6 we provide some concluding remarks, and we depict possible future works.

2. AGENT TECHNOLOGY AND GRID

A software agent is an autonomous software entity able to expose a flexible behavior. Flexibility is obtained by means of reactivity, pro-activity and social ability [Wei99a]. Reactivity is the ability to react to environmental changes in a timely fashion while pro-activity is the ability to show a goal directed behavior by taking the initiative. Social ability, that is the ability to interact with peers by means of cooperation, negotiation, and competition, is one of the most important features of agent oriented programming: agents do their best when they interoperate. Interaction is obtained by arranging agents in communities called multi-agent systems (MAS). MAS are generally decentralized open systems with distributed control and asynchronous computation: they provide a context for agents' activity with the definition of interaction and communication protocols. In addition they are scalable, fault-tolerant, reliable, and designed for reuse.

An abstract architecture specification of a generic multi-agent system has been proposed by the Foundation of Intelligent Physical Agents (FIPA), an international organization that promotes standards for agent technologies. The proposed architecture [FIP01a] is implemented by different multi-agent systems and has been taken as reference model in the comparison of different implementations of MAS.

Agents are reliable components to build flexible and fail safe systems, since autonomy and reactivity allow recovering from fault conditions. Agent and multi-agent technologies provide a promising approach to make Grid technologies smarter, more flexible, and adaptable. To support Grid computing, agents can offer different roles, be organized into dynamic groups, and be able to migrate between groups to support load balancing. Therefore, agents could play an important role in Grid computing, and Grid computing can offer useful test-beds for investigating Agent services. The social ability, the autonomous and flexible behavior could play an important role for the communication and the interaction with different nodes, for example, in exchanging information about the resources available on each node. The intrinsic nature of Agent technology, explicitly oriented to model high dynamic and complex systems [Woo99a], seems to be well suited to meet the Grid computing requirements. Moreover, the adoption agent technology could bring to Grid users and administrators more friendly and understandable

interfaces to interact with the system. Some projects have already proved that the agent oriented approach could be adopted for Grid computing. The Agile Architecture and Autonomous Agent [Cao02a, Cao01a] (A4) is an agent based methodology for grid resource management. The computational power of the Grid is managed with a hierarchy of identical agents used to provide an abstraction of the system architecture. Each agent is able to cooperate with other agents to provide service advertisement and discovery to schedule applications that need to use grid resources. The Bond Agent System [BOND] is a FIPA project on top of which is possible to build agent based applications able to manage the state of the nodes and the coordination of a distributed system such a Grid [Kha03a].

3. THE AGENTSERVICE PROGRAMMING PLATFORM

AgentService [Boc04a] is a framework designed to develop multi-agent systems. It provides a class library to implement agents, an agent platform hosting multi-agent systems and a set of monitoring and design tools supporting either the development or the management of the MAS. The framework does not enforce particular agent architectures, but provides developers with a flexible agent model based on the concepts of knowledge and behavior. An agent is modeled, and implemented, as a software entity whose state is defined a set of knowledge objects, and whose activity is carried out by a set of concurrent tasks known as behavior objects. A knowledge object is a shared object containing related items which together define a unit of information. Knowledge objects can be shared among behaviors objects which model the different capabilities of an agent. AgentService comes with a set of extensions to the C# programming language that simplifies the development of agent applications. The AgentService object-oriented model is hidden by the APX [Vec03a], so that a clear agent-oriented interface is offered to the developers with slight changes to the C# syntax.

The platform provides a complete environment to execute agent instances which rely on the advanced services of the platform: repository, communication, and directory services. Some of these services become strategic when platform instances constitute the nodes of a computing grid. In particular directory and communication services are discussed in detail.

Directory Service

AgentService has been designed following the architectural specifications provided by FIPA which states that a set of basic services are required on each agent platform. These are implemented as agents and are:

- Agent Management Service (AMS) - supervisor and controller of the platform services;
- Directory Facilitator (DF) - providing yellow pages service;
- Message Transport Service (MTS) - managing communication service.

Directory services are fundamental in dynamic and distributed environments due to the fact that a single entity needs to know if, when, and where a specific service is available. For these reasons DF is a compulsory component for an agent platform. In AgentService each platform provides a directory service to agents. By registering to the DF agents can specify the services they offer and their communication profile. Directory Facilitator agents scattered on AgentService platforms can join together to form a federation, hence if an agent registers to the local DF, it becomes visible, and advertises its services, to all the platforms of the federation. When deregistration occurs the information is spread on all the nodes of the federation. DF agents maintain a distributed database of all the services available on the federation: each agent by interacting with the local DF gets access to an entire net of services. DF agents according to the service profile advertise it on all the nodes of the federation or just to a subset of it. The ability of controlling the advertising policy allows a better use of the network resource.

Communication Infrastructure

A dedicated agent, the MTS (Message Transport Service) is responsible of managing the platform messaging subsystem. The messaging subsystem is implemented within a module and by default AgentService provides a communication service based on message exchange and conversations (connected communication between two agents). The ability of changing the implementation and the communication channel among platform nodes in a transparent manner for agents is remarkable advantage of this architecture: the implementation of the module is hidden to the MTS, and then to the agents, which interacts with the module through the IMessagingModule interface. The messaging module creates and maintains a specific message queue for each agent hosted in the platform and can choose the best technology solution to store this information (a database, a file system, or a message queuing service). Messages exchanged among agents are compliant to the FIPA specifications and need to contain only serializable items, since messages may

trespass the boundary of the single machine. The default messaging module provided with the AgentService installation comes with two fundamental services: conversations and inter-platform message dispatching. Conversations are connected message exchange services and provide a useful abstraction to model interaction protocols. Inter-platform message dispatching allows the community of agents to extend beyond the single platform instance boundaries. The communication among different AgentService installations is based on the Web Service infrastructure provided by .NET framework. Hence soap messages are exchanged among platforms and a specific format of the xml content is defined by AgentService to ensure the secure and correct delivery of the messages. The .NET automatic serialization process for the soap messages has been customized to allow the serialization of agent messages and to decrease the amount of the transferred data without loss of information.

Additional Services

The platform has been designed to be an extensible software environment: the community of agents hosted in the federation of platforms evolves and additional features may be required when the system is installed. Hence the ability to extend the proper capabilities becomes a requirement. The architecture of the platform allows third party modules to be integrated into the platform core and to offer services to either the other modules or the agents. By using this technique the platform has been extended by an FTP service available to all the other platform components. Agents and other modules can require a folder space or just send files by using the service as a simple FTP client.

The directory service, the communication infrastructure and additional services, together with a set of dedicated agents constitute the core of the grid infrastructure provided with AgentService.

4. DESIGN OF A GRID INFRASTRUCTURE OVER THE AGENTSERVICE PLATFORM

The elements defining the grid infrastructure are agents, platform components, and additional services. Agents encapsulate the logic of the system while platform components and additional services maintain its structure. This organization is replicated on each installation of the platforms participating in the grid.

Figure 1 gives an overview of the entire system. The federation of the platforms defines the boundary of the grid. The structure of the systems is dynamic since AgentService instances can dynamically join the federation by sending a message to the agent managing one of the nodes. In the same way nodes can detach from the system. This is a fundamental feature for grid systems that are dynamic by nature. According to the configuration of the node each platform can act as a computational node, provide access to the system, or perform both the two roles.

The System's Logic: the Agents

The logic of the system is composed by a community of specific agents deployed on each installation of the AgentService platform. In this section we will describe the tasks delivered to each agent and how they take advantage of the services offered by the platform to deploy and to manage the infrastructure of the computational Grid.

Each node which is part of the grid infrastructure is equipped with an installation of the AgentService platform. On each node the platform hosts the following agents:

- NodeManager: the NodeManager is the maintainer of the node, it coordinates all the activities required to implement the grid service. The NodeManager maintains a registry of the platforms which constitute the computational grid and manages the dynamic registration of platform instances. The NodeManager is responsible of assigning a task to a specific node by looking at the topology of the grid, at the computational load of each node, and at the services offered by that node.

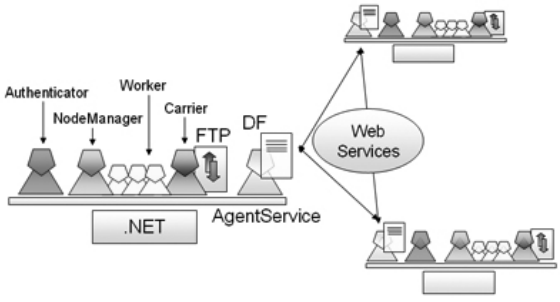

Figure 1. A graphical overview of the Grid Architecture based over the AgentService Framework

- Carrier: the Carrier agent is responsible of transferring on the selected node of the grid all the resources required to perform the task. The Carrier relies on the file transfer service offered by the platform, by which it delivers to the selected node the object code containing the task and all the related input or data files.

- Authenticator: an instance of the Authenticator agent manages the security of the node; it maintains a registry of user profiles, checks the user credentials when a task is submitted to the grid, and applies the security policies defined in the user management module.
- Worker: multiple instances of the Worker agent are hosted on each node and take care of tasks execution. On the selected node, the NodeManager contacts the worker agent every time a new task needs to be executed; the worker agent sets up the computing environment required for the task, executes the task, and eventually communicated the results. The NodeManager agent can limit the maximum number of concurrent Worker agent instances in order to control the computational load of the node. Worker agents can perform many tasks concurrently thanks to agent model adopted by AgentService. The tasks partition criteria among worker agents can be defined as configuration parameters of the node or dynamically decided by the NodeManager; a simple selection criterion could be dividing the tasks according to the permission of the users they belong to.

Tasks are submitted to the grid and NodeManager agents cooperate to identify the candidate node on which the task will be executed. Since cooperation, negotiation, and competition are natural activities in multi-agent systems this functionality is naturally obtained by using the agent oriented approach. In the same way localization of services and coordination within a single node are obtained with less effort.

The Grid Structure

The community of agents that is distributed on the nodes constituting the grid gives a high level view of the entire grid. The implementation of the infrastructure strongly relies on the core services of the platform. In particular, communication services, file transfer, and localization. These features are respectively implemented by using the messaging subsystem, the FTP service, and the DF agents spread on each node.

The messaging subsystem is one of the core elements of both the multi-agent system and the grid infrastructure implemented on it. Software agents interact with peers by exchanging messages; hence the coordination of the elements defined in the logical layer is based on the messaging subsystem. The ability to communicate with peers hosted on other nodes is a requirement to distribute computation; hence, the installation of AgentService has been customized with a messaging module that uses the web services technology to deliver messages on other platforms. The use of web services provides a solid, well known standard allowing interoperability and integration with other applications. Agent messages are required to be serializable but not to be represented by using a SOAP message. The platform replaces the default XML serialization provided by the .NET framework with a custom technique that reduces the body of the SOAP message and allows the transport of any serializable managed type. The messaging module attaches the description of the type to the binary serialization of each item in the agent message; the binary instance is encoded into a base64 string and transmitted as an attribute of the XML element representing the item. On the target node each item is reconstructed according to the type information attached to the item: the full name of the type, its assembly name, and the public key token of the assembly are used to de-serialize the instance into the original object. This solution speeds up the transmission of any complex object via web services, avoids type mismatch, and is completely transparent to developers which are not required to provide an XML serializer for every type they define.

The ability to transfer objects among platform nodes is a requirement for distributing the computation. The messaging subsystem provides a simple way to transport messages but it cannot handle efficiently the transfer of large amount of data. Moreover, the communication infrastructure has been designed to send .NET instances and not for large files. For this reason, the installation of AgentService has been enriched with an additional module that handles the FTP protocol. The module integrates into the platform and provides this feature as service. The FTP service can be exploited either by software agents or platform modules and it is mainly used to move on the target node all the assemblies containing the code executing the task and the required data files. Modules and software agents can dynamically check the availability of the service and eventually require a personal folder or just submit a file to transfer. When files are uploaded to the server the owner of the folder is notified about the transfer. In this case the FTP service is mainly used by the Carrier agent who is responsible of transferring the assemblies containing the task to be executed on the target node. Carrier agents ask for a personal folder to the FTP service and the FTP service creates the corresponding directory in the root folder of the FTP server. When a task is moved to a node of the grid the Carrier agent on the source platform instruct the FTP service to upload the file on the target platform. When the upload is finished the FTP service of the target platform notifies the Carrier agent about the

transferred files. The same interaction pattern is used by modules if they need to send or receive files.

Localization and discovery of services play an important role in distributed systems. The ability to discover agents and the services they offer is a requirement for agent communities which are dynamic by definition. These are requirements for Grid systems too: nodes should be able to obtain information about other nodes in order to distribute the load. Within AgentService a distributed directory service is responsible of advertising and retrieving services available in the multi-agent system. Directory Facilitator agents constitute a federation sharing all the information about published services. DF agents provide information to NodeManager and Carrier agents: the first ones query the local DF in order to know all the other NodeManager agents and set up the topology of the grid; the second ones look for Carrier agents when they need to transfer files on a selected node. DF agents are also useful for connecting agents within a single node: each of the previously defined agents register its service profile to the local DF. Directory Facilitator agents can be instructed for a local search: in this way the agents defining the logical layer of the grid connect each other.

Many of the elements constituting the infrastructure of the grid are provided by the environment hosting the agent. These elements are commonly required by the agents to perform their activities; hence the use of a multi-agent system for grid computing can strongly simplify the development of grid system. In addition, the modular architecture of the AgentService platform and its natural extensibility allows the simple implementation of the missing features as in the case of the FTP service.

5. CASE STUDY

A common computing task submitted to the grid can be taken as a case study since it is useful to describe the interaction among the agents modeling the logical layer of the grid and their connection with system components.

Users that want to submit a job to the grid have to contact those nodes which are configured as access points to the grid. These nodes are the starting point of the entire process. The user authenticates by sending a message containing tis credentials to the Authenticator agent of the access point. Installations of the AgentService platforma provide a communication channel that can be used by GUIs or web applications for remote management and access: the common scenario involves a web application connecting to the access point through a web browser submitting a task by uploading all the required files. The web application connects to the platform with the credentials provided by the user and queries the DF for the Authenticator agent which checks the user permissions and validates the request of the user. The Authenticator agent sends a message to the NodeManager agent of the same platform which selects the best node of the grid according to:

- the user profile;
- the type of task to execute;
- the availability of processor cycles on each node.

In order to select the best node NodeManager interacts with the other NodeManager agents hosted on the other platforms. The NodeManager agents maintain updated the state of the entire grid by exchanging messages when interesting events occur (a task is finished, a task is started, a task has been aborted); hence, each NodeManager agent is always aware of the status of the grid.

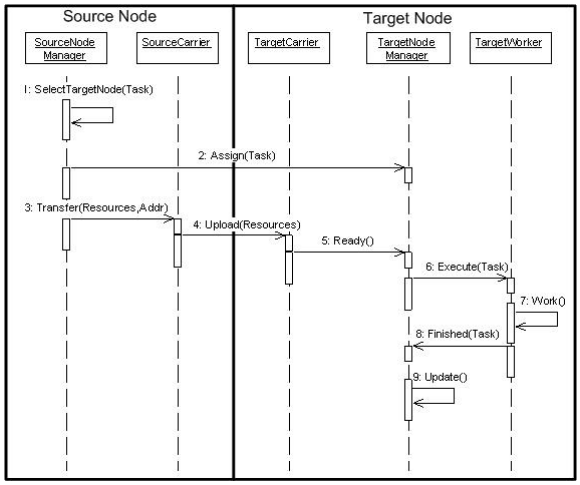

Figure 2. Sequence diagram describing the protocol for task execution

Once the node has been selected the local NodeManager is contacted to start the task. The target node could require additional resources to perform the task and in that case the NodeManager agent instructs the local Carrier agent to accomplish the transfer on the target site. The local Carrier agent by querying the DF looks for the remote Carrier agent and then sets up the transfer by using the local FTP service. On transfer completion the Carrier agent on the selected node notifies the local NodeManager that all the resources required to perform the task are available. This is the final step of the activation process: the NodeManager agent according to the computational load of the node requires a new Worker agent or submits the work request to an active Worker agent. The number of

active Worker agent can change on each node and the NodeManager itself can dynamically decide the best policy to apply. Figure 2 depicts the sequence diagram describing task execution after the credential of the user have validated.

The Worker agents picks up a new work request inspects the information describing the task to execute and by means of reflection creates a new instance of the type defining the tasks, starts its activity by using a configuration files transmitted along with the resources. Assemblies containing the tasks can be cached on the nodes in the platform storage and useless transfers can be avoided. The types must implement the following interface:

```
interface ITask
{
  bool IsReusable { get; }
  Exception Error { get; }
  bool Prepare(string configFile);
  void Execute();
  bool Abort();
  bool Dispose();
}
```

In order to execute a task the Worker agent creates an instance of the required type and invokes the Prepare method that configures the task to execute. If the method returns true the task will be executed by invoking the Execute method and upon completion a call to Dispose finalizes the execution and eventually communicates the results. Exceptions occurred during execution are obtained by looking at the Error property while, while IsReusable is true if the same instance can be used to perform many tasks of the same type in sequence. Two additional interfaces are provided to make tasks execution more flexible: IControllableTask and IIterativeTask. The first one adds facilities to control task execution with a pause-resume pattern while the second one allows the execution of tasks one step at time.

When the task is finished, the Worker agent notifies the NodeManager about completion which update the status of the grid.

6. CONCLUDING REMARKS AND FUTURE WORKS

Agent technology seems an interesting solution to implement distributed and dynamic computational environments: agents confer a certain degree of autonomy to the system components and simplify the creation of dynamic relations among them. Hence, the use of such technology in the field of grid computing is a reasonable and interesting approach. This paper has presented the design and the implementation of an infrastructure for grid computing which relies on agent technology and takes advantage of the AgentService framework. The community of agents defines the logic of the system while the extensible core of the agent platform implements the low level services required by a Grid architecture. This approach has two main advantages:

- the coordination and task distribution policies can rely on the interaction capabilities of agents: they are high level system components which naturally embed negotiation, competition and cooperation capabilities;
- the default services provided by multi-agent system meet typical grid computing requirements; hence the use of a modular and extensible multi-agent system, like AgentService, as a backbone simplifies and improves the efficiency in the Grid architecture development.

The structure of the system is based on a net of platform instances connected together by using the web services technology. Web services are used only for communication and AgentService implements custom technique which allows the transfer of any .NET serializable and complex object, keeps the SOAP packet small, and speeds up the transfer. The use of web services could lead to possible performance bottlenecks but message exchange among agents should have a small cost if compared to the time required to perform tasks submitted to the grid. In addition, AgentService uses web services only for communication and has been enriched with an FTP service that is used to move object code and data files among node.

The architecture described in this paper is specifically designed for computational Grids, but the underlying model can be applied also to other types of grids. A possible extension of the presented architecture could be the ability to move agents which are performing a task in order to apply load balancing policies. This service could be provided by adding a mobility module in order to provide a task migration service. This module allows agent instances to cross the platform boundaries and move among AgentService platform instances.

7. REFERENCES

[Fos01a] Foster, I., Kesselman, C., and Tuecke, S. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. International Journal of Supercomputer Applications, 2001

[MPI] Message Passing Interface Forum. Message Passing Interface, documentation available on line at www.mpi-forum.org

[Gei94a] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, B., and Sunderam, V. PVM: Parallel Virtual Machine a User's Guide and

Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, 1994

[Fos02a] Foster, I., Kesselman, C., Nick J., Tuecke S., The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, Global Grid Forum, June 22, 2002

[Fos05a] Foster, I., Globus Toolkit Version 4: Software for Service-Oriented Systems, IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2005

[Lut05a] Luther, A., Buyya, R., Ranjan, and R.,Venugopal, S., Alchemi: A .NET-Based Enterprise Grid Computing System, Proceedings of the 6th International Conference on Internet Computing (ICOMP'05), June 27-30, 2005, Las Vegas, USA.

[H20] H2O Project, http://www.mathcs.emory.edu/dcl/h2o/

[Kur03a] Kurzyniec, D., Wrzosek, T., Sunderam, V., and Slominski, A.. RMIX: A Multiprotocol RMI Framework for Java. In Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'03), pages 140-146, Nice, France, 2003

[Jen99a] Jennings, N.R., and Wooldridge, M., Agent-Oriented Software Engineering, Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99), 1999

[Wei99a] Weiss, G., Multi-agent Systems – A Modern Approach to Distributed Artificial Intelligence, G. Weiss Ed., Cambridge, MA, 1999

[Boc04a] Boccalatte, A., Gozzi, A., and Grosso, A., Una Piattaforma per lo Sviluppo di Applicazioni Multi-Agente, WOA 2003: dagli oggetti agli agenti – sistemi intelligenti e computazione pervasiva, Villa Simius, Italy, September 2003

[FIP01a] FIPA Abstract Architecture Specification, http://www.fipa.org/specs/fipa00001/

[Woo99a] Wooldridge, M., Intelligent Agents, in Multi-agent Systems – A Modern Approach to Distributed Artificial Intelligence, G. Weiss Ed., Cambridge, MA, 1999, pp. 27-78

[Cao02a] Cao, J., Spooner, D. P., Turner, J. D., Jarvis, S. A., Kerbyson, D. J., Saini, S., and Nudd, G. R., Agent-Based Resource Management for Grid Computing, Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)

[Cao01a] Cao, J., Kerbyson, D. J., and Nudd, G. R., Performance Evaluation of an Agent-Based Resource Management Infrastructure for Grid Computing, Proceedings of 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '01), Brisbane, Australia, May 2001

[BON] BOND Project, http://bond.cs.ucf.edu/

[Kha03a] Khan, M.A., Vaithianathan, S.K., Sivoncic, K., and Boloni, L. Towards an Agent Framework For Grid Computing, CIPC-03 Second International Advanced Research Workshop on Concurrent Information Processing and Computing, Sinaia, Romania, 2003

[Boc04a] Boccalatte, A., Gozzi, A., Grosso, A., and Vecchiola, C. AgentService. The Sixteenth International Conference on Software Engineering and Knowledge Engeneering (SEKE'04), Banff Centre, Banff, Alberta, Canada 20-24 June 2004

[Vec03a] Vecchiola, C., Coccoli, M., and Boccalatte, A. Agent Programming Extensions relying on a component oriented infrastructure, Proceedings of the 2003 IEEE International Conference on Information Reuse and Integration (IRI - 2003), Oct. 26-29, Las Vegas, NV, 2003.

A lightweight infrastructure to support experimenting with heterogeneous Transformations

Wolfgang Lohmann
Rostock University
Albert-Einstein-Str. 21
18051 Rostock, Germany
wlohmann@informatik.uni-rostock.de

Günter Riedewald
Rostock University
Albert-Einstein-Str. 21
18051 Rostock, Germany
gri@informatik.uni-rostock.de

Thomas Zühlke
Rostock University
Albert-Einstein-Str. 21
18051 Rostock, Germany
thomas.zuehlke@uni-rostock.de

ABSTRACT

We report on a class library called Trane, which provides an infrastructure to support experimenting with transformations interactively. Transformations here mean algorithms, which take software artifacts as input and output manipulated artifacts. Trane supports easy combination of transformations available in different languages, libraries and tools. Several combinations can be presented at the same time, parameters can be visually changed, and results can be compared. New transformations can be easily added. Generated transformations from experiments can be integrated into the experiments at run-time.

The paper presents the general model of the class library. We show how the class library profits by the features provided by .NET, such as language interoperability, foreign language interface, shell access, reflection, and web services by demonstrating five variants to integrate new transformations.

Keywords

Transformations, .NET, Language interoperability, cross-language inheritance, visual programming, component-based transformation systems, platform independence

1. INTRODUCTION

We report on a lightweight infrastructure developed to support experimenting with transformations interactively. Here, transformations mean algorithms, which take software artifacts as input and output manipulated artifacts or results of an analysis. We use .NET, as it facilitates integration and combination of heterogeneous transformations, i.e. transformations available as programs in different languages, existing command line tools, web services, libraries through a foreign language interface, and dynamic compilation and loading of DLLs resulting from a transformation.

Experiments with Transformation Nets

Some kinds of complex transformation are developed in an explorative way, where they are extended after a test with representative examples shows that the development might be on the desired way. Examples vary from combinations of UNIX command line tools such as sed, awk, grep to extract and manipulate information in text files to more sophisticated examples, such as refactoring, where there are many ways to achieve an improvement of the source code, or to achieve software evolution by transformations [Läm04, Set04]. Another example is the collection of individually changes for maintenance in batch files for later reuse in [Klu05].

We intend to use Trane to experiment with transformations on language components, e.g. grammars, semantic descriptions, and language processors, though it is not restricted to those applications. We want to extend languages stepwise during their development, explore several possibilities, how a grammar could be changed, compare the variants, extract parts of existing grammars and adapt them to form a sublanguage DSL, and directly connect the generated output to front end generators to test example programs. There are tools, but they are available in different formats, e.g. command like tools like yacc and GDK [Kor02], left-recursion removal for attributed grammars in Prolog and TXL [Loh04], grammar representations in XML, BNF etc.

.NET Technologies 2006

However, to the user, it should not matter, whether a transformation is a command line tool like yacc, or an analysis written in Prolog, and should be represented uniformly modulo their parameters.

Using .NET

We were interested in an implementation on .NET mainly because it comes with the promise of language interoperability and cross-language inheritance. With C# as main implementation language, we could make use of properties, generics, delegates, reflection, and web services. The implementation was also an experiment in platform independence wrt. the availability of .NET on Linux as well as Gtk# on Windows.

Resulting Prototype

We designed a simple class model. Transformations are represented by automatically generated or self-designed boxes to be placed on a workspace, which is itself part of a box. The boxes have typed input and output ports, which can be connected using converters to describe dataflow. Boxes can provide facilities to control transformation parameters. Several sequences of transformations can be presented simultaneously, parameters are visually changeable, and results can be compared.

Trane can be extended easily with new transformations. New boxes can be any program, a web service, an encapsulated command on shell level, etc., written in any .NET language, as long as the box interface is implemented. Thus, the user creates transformation nets without paying attention to the implementation of a transformation. Due to reflection, no extra configuration files are necessary. Trane can also be seen as a wrapper architecture or an interpreter for call graphs of complex functions. It is a lightweight implementation, because .NET already encapsulates much work for the integration of transformations.

Remainder of the Paper

In Section 2 we present the concept of Trane. In Section 3 we discuss the model and the computation strategy. In Section 4 we show five categories of transformation and how they are integrated. Section 5 discusses some related work. Finally, the paper finishes with concluding remarks.

2. TRANE CONCEPT

Trane provides facilities to model **Tra**nsformation **ne**ts with heterogeneous transformations. In Figure 1, for example, an attribute grammar of a robot move language is sent to the Lisa web-service, which generates a compiler for that language. Using LisaJavaCompile (wrapper for Java at command line), the Lisa generated code is compiled. In the second sequence, a description of a maze in XML is converted to Prolog by an XSLT based transformation. A Prolog-based transformation now analyses the inherent graph and generates a program for robot moves to control its way through it. The program is saved, the filename is delivered to the generated compiler RunLisaCode for the robot language. The result of the execution, the final position of the robot relative to start position (0, 0), is delivered to the TextOutput.

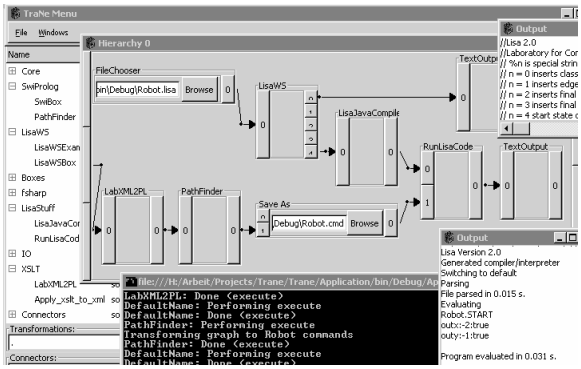

Figure 1. Trane in action

The underlying structure is a directed graph with nodes representing transformations. Nodes have input and output ports, which possess types, and correspond to input and output positions of the transformations. Output ports can be connected to input ports of other nodes by directed edges, assumed the types associated to the ports are equal. This way, the call graph of a composite transformation is modelled.

Connections between ports of different types can be obtained indirectly by converters. These are special transformations, which map values of a given type onto values of a related type. In the graphical representation, they are hidden behind connections to allow a simplified view on the net. For example, it should not matter that the result of a transformation is a grammar in XML format, but the next transformation expects it in a BNF style. An XML2BNF connection can transport the grammar and hide the necessary format conversion. The user simply chooses the connector with the desired type combination. Data transported can be text as in UNIX-pipes, structured data such as grammars, or file names for results in files.

Transformations can be added at run time, e.g. transformations created with Trane. Providing a new transformation means to embed a transformation into a node such that input and output ports are provided with data. To create a new converter means to provide a new transformation, which implements the desired type mapping. This requires knowledge about the structure of data.

The order of computations is determined by the dependencies between transformations in the graph. Cycles are not considered, as their role is not clear in

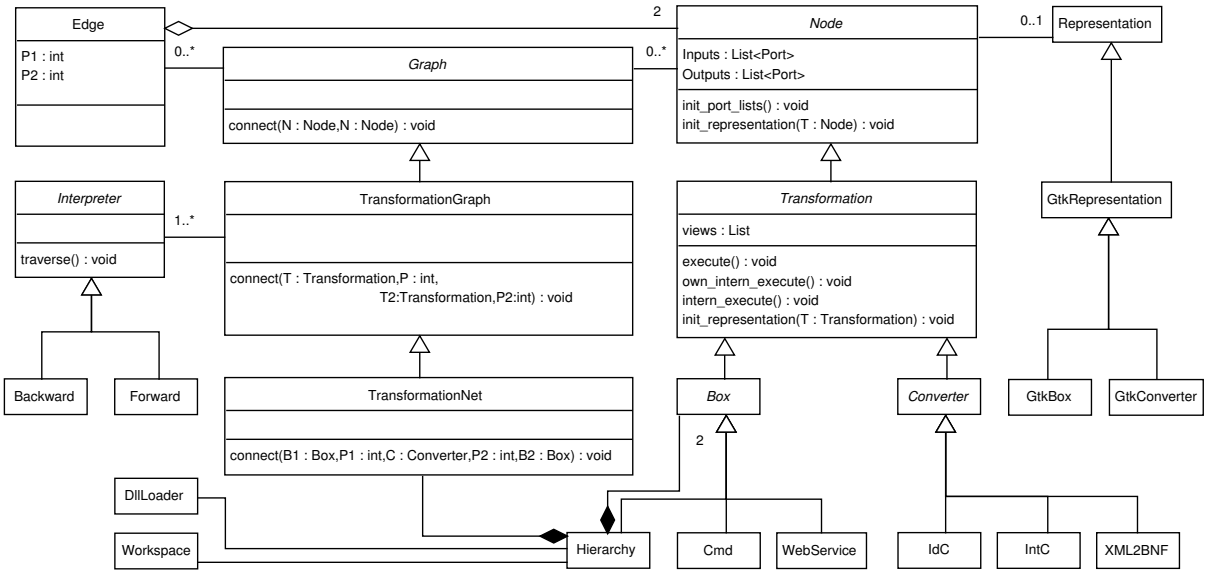

Figure 2. Class model of Trane

this setting. The computations are performed always once, when a result is demanded and the required input data for the transformation is available. Results can be queried at any output port at any transformation, thus, comparing the values of different transformations is possible. The intermediate results can be investigated, which is helpful, if the result of a transformation delivers unexpected values.

3. OBJECT-ORIENTED MODEL

Figure 2 shows the UML class diagram of the infrastructure, which largely mirrors the concept.

First Level: Combination Infrastructure

The class *Transformation* defines minimal requirements of transformation nodes. As can be seen in the class diagram, it provides lists for input and output ports. These ports manage edges connected to ports of other transformations, data, and a type annotation, which constrains data accepted. Data is packed in a separate object, which provides its value and a type. This allows for a subtype concept, i.e. the value has to be a subtype of the type of the port. The values are used as input and output values for a transformation and the object representing the transformation. To define the port lists of a special transformation, it has to override method *init_port_lists* to configure the ports (e.g. with type annotations). Port lists are extendable dynamically at run-time. Ports of transformation objects are connected using the method *connect/4* of *TransformationGraph*, which tests on type conformance, creates an edge between the ports, and keeps track of transformation objects and their connections. Edges store the nodes and indices of the ports connected.

A subclass has to override *execute*, where the actual mapping from values of input ports to values of output ports is defined or the embedded transformation is called. The computation can depend on several conditions, such as the actual computation strategy, or lazy computation (do not compute if input values have not changed). To save the user from uninteresting management work, *execute* is wrapped by methods *intern_execute* and *own_intern_execute*, which take care of the conditions, and at a suitable point in the computation call *execute*. *init_representation* associates a representation to an instance of *Transformation*.

The difference between common kinds of transformation nodes and converters is expressed by classes *Box*, which box a desired transformation, and *Converter*, whose main task is to provide some kind of type conversion. The provider of a converter will find it nice to implement it like any other transformation. They only differ from boxes through their representation and arity. This enables converters of all kinds, simple converters or arbitrary complex computations, from which the user would like to abstract in a model.

The *TransformationNet* provides a method *connect/5* to connect two objects of type *Box* using a *Converter* at the ports specified with the port index each.

We decided for overriding of some *init*-methods over configuration inside of a constructor, because in the chosen implementation language C# constructors of super classes are evaluated first before that of the actual class. For some tasks provided in the super class, e.g. for the generation of graphic representations, it is necessary that the actual class is configured already at least partially.

Second Level: Interactivity and Views

The second level provides graphical representations for transformations. In the standard representation, rectangular boxes are generated for transformations (e.g. most representations in Fig. 1 are generated.). Lists of buttons, which also activate the execution of the associated box, represent input and output ports. Converters are represented as a line, which connects two boxes. This simplifies the view on the transformation net.

If desired, the provider of the transformation can create own representations for boxes and converters by inheriting *GtkBox* and *GtkConverter* respectively. Their instances are associated to the specific transformation class by overriding *init_representation*. Objects of class *GtkBox* can be provided with additional buttons, fields, sliders, and other kinds of input/ output support for users to control the transformation.

Objects of transformation nodes can provide several views at them. The first level can already be considered as the most basic view. The main view used is the graphic representation on a workspace to combine them. In addition, more information and controlling facilities are possible, e.g. a description of the transformation represented by the object, a description of its input/output, complex tables for the user to describe or influence the way the transformation is working, status messages, and logs. Note, the workspace in Fig. 1 is just another view on a special box, allowing to create a hierarchical subnet interactively.

Providing a New Box

To create a new box, the following steps are followed: 1) Choose a box to inherit from. 2) If desired, override *init_port_lists* to redefine input and output ports by simply adding new ports to a generic list. 3) Override *execute* to describe, how values of input ports are used by the transformation to compute values and copy them into output ports. 4) If a new representation is desired, create a new subclass of *GtkBox* and redefine components or add new features to the inner frame, e.g. a button to show a new view, which can be any graphical object. Override *init_representation* in the box to assign it to the box.

Computation Strategy

There are several variants to initiate computation of the transformation net: backward and forward computation (similarly to demand-driven vs. data-driven) and direct vs. indirect data transport. The choice is realised through an instance of *Interpreter*, who performs/initiates the traversal.

With direct data transport, a transformation itself informs its successors / predecessors about results/ required results and calls their *own_intern_execute*. With indirect data transport a separate object of class *TransformationNet* controls the traversal process, e.g. calls *intern_execute*. Note, that by *connect*/5 the object keeps book about created transformations and connections. This allows intercepting and changing values for experimenting.

Backward computation is initiated by requesting the output port of the last transformation of a chain by initiating *own_intern_execute /intern_execute*, which then determine missing input values for the computation of the embedded transformation, and activate the preceding transformations. When all values are available, the wrapped *execute* is called. This strategy will be used mostly to compare several transformations at the end of a common sequence.

The forward computation strategy is thought for experiments to investigate the effect of a changed input. E.g. a composite transformation can be attached to a text editor, and show the results of a transformation chain immediately while typing e.g. a new part of a grammar (or delay start until a save-command is fired). Forward computation is simulated on top of the backward computation by calling the output ports of following transformations. This can be very expensive, though. Cycles are not allowed in the computation though we have not included a check to avoid them yet (we could think of a graph analysis based on a term generated from the net).

4. VARIANTS OF BOXES

Many transformations will only inherit from the common box type, configure the input and output ports, and define a mapping between them to create different kinds of boxes. However, using .NET, several different kinds of special box categories are viable, e.g. hierarchy boxes (to provide subnets and workspaces), web service boxes, command line tool wrappers, compilers, foreign libraries wrappers, or DLL loaders. Here we show five variants to integrate different transformations in boxes.

Web Services

As an example for a web service transformation we show in Fig. 3, how to implement the compiler generator box LisaWS used in Fig. 1. Lisa [Mer99] is a compiler generator system also available as web service. When sending an attribute grammar, it generates and delivers Java code of a compiler. The code can be compiled and the resulting compiler can be used for the programs of that language.

LisaWS gets an input port for a string value, the attribute grammar. An output port is configured to provide a string for a path (to store the generated files), and further ports, where the generated lexer, scanner, parser, and evaluator can be requested separately.

```
public class LisaWSBox : Box {
  public override void init_port_lists(){
    Inputs.Add(new Port("String"));
    Inputs[0].data =
           new ValueData(null, "String");
    Outputs.Add(new Port("String"));
    Outputs[0].data =
           new ValueData(null, "String");
       … // some more output ports
  }

  public override void  execute(){
    CServiceBeanService lisaService =
              new CServiceBeanService();
    System.Net.CookieContainer container=
         new System.Net.CookieContainer();
   lisaService.CookieContainer=container;
   lisaService.mkdir("wlohmann");

   // read file with lisa specifications
   String path = Inputs[0].data.value;
   FileStream fs = File.OpenRead(path);
   StreamReader r = new StreamReader(fs);
   String Spec = r.ReadToEnd();
   lisaService.clearError();

    // compile and save specifications
    bool OK = lisaService.compile(Spec);
    if (!OK) { … /* error */ } else {
        String scanner =
               lisaService.getScanner();
        Outputs[0].data.value = scanner;
        …  }
 }
}
```

Figure 3. A web service box

We find it especially charming to integrate remote applications into transformation nets from locally existent algorithms. Problems might be that connections are unavailable, or slow. Depending on the kind of service boxed, the transformation could require to re-compute always, even if no input values have changed.

Hierarchical Transformations

Hierarchy in transformation nets means to hide a transformation subnet *TSN* behind a box B_H, which looks and behaves like other boxes with input and output ports. Note, there are different types of hierarchy boxes. They can differ in the number of input/ output ports, or in the way they are to be used. Hiding requires mapping inputs and outputs of B_H to inputs and outputs necessary for *TSN*. This can be easily done by providing two identity boxes B_I and B_O as interface for inputs and outputs, between which *TSN* is constructed. Since transformations use properties to connect to ports, .NET helps to redirect port access to the input ports of B_H to input ports of B_I as well as output ports of B_H to those of B_O by simply overriding the definition of the properties (see Fig. 4). The graphical representation is extended by a button, which when pressed provides a second view, namely the workspace of the hierarchy box. Figure 1 shows the inner view of a hierarchical box. We additionally added a transformation browser for choosing boxes and converters. This browser makes use of reflection to analyse DLLs in a chosen directory and to create instances of provided classes.

```
public class HierarchyBox  : Box {
  public IdBox InputBox  = new IdBox();
  public IdBox OutputBox = new IdBox();

  // Hide Inputs of this box by pointing
  // to corresponding interface box
  public override  List<Port> Inputs {
      set { InputBox.Inputs = value; }
      get { return InputBox.Inputs;  }
  }

  public override List<Port> Outputs{ … }

  public override void init_port_lists(){
      base.init_port_lists();
      InputBox.Double_PortLists();
      OutputBox.Double_PortLists();
  }

  public override void execute() {
      OutputBox.ownInternExecute();
      // Input execute not necessary
  }
  // save hierarchy in a separate subnet
  private TransformationNet _TraNe =
              new TransformationNet();
  public TransformationNet TraNe {
     get { return _TraNet; }
  }
  public override void
        init_representation() {
     this.Representation = new
   Gtk_HierarchyBox_Representation(this);
 }
}
```

Figure 4. A plain hierarchy box

Use of Native Libraries

As an example for the use of existing DLLs outside of .NET we choose SWI-Prolog [Wie06], mainly because we want to use Prolog for experiments with transformation tasks similar to [Loh04, Loh03]. In Fig. 1, the PathFinder-box is based on Prolog. It determines a path through a labyrinth and generates a control program in the Robot language for it.

```
[DllImport(DllFileName)]
internal static extern uint
                      PL_new_term_ref();
     …
// make a PlTerm from a C# string
public PlTerm(string text) {
  m_term_ref = libpl.PL_new_term_ref();
  libpl.PL_put_atom_chars
                      (m_term_ref,text);
} // SwiCs.cs by Uwe Lesta
```

Figure 5. Snippet from SwiCs.cs

.NET offers the attribute *DllImport* to define access to foreign libraries. We created a DLL based on *SwiCs.cs* (cf. [Les03]) where for each exported func-

tion in the library its name is declared after the attribute (Fig. 5). The DLL provides .NET programs with methods and types to model Prolog terms and to query a SWI-Prolog engine; and is used by the box.

```
public override void execute(){
  String[] param = { @"H:\\ Projects" +
              …  "\\Application.exe"};
  PlEngine e = new PlEngine(1, param);

  // Get query as Text, call it, e.g.
  // (tell('log'),write('HiWorld'),told);
  string goal = (string)
        (Inputs[0].data.copy().value);
  PlQuery q = new PlQuery("call",
      new PlTermv(new PlCompound(goal)));
  bool b = q.next_solution();  q.free();
}
```

Figure 6. Providing direct Prolog access

Figure 6 shows how to interpret a string input as Prolog term directly and to call it. Combined with text boxes it can serve as interactive Prolog interpreter. Also, a Prolog box can provide programs that are more complex or initiate loading of a rule base.

A problem is, in our opinion, that the attribute *DllImport* expects a static string, which has to be known at compile-time. This makes replacing different versions of the Prolog DLL impossible without recompilation of the interface DLL *SwiCs.cs*, thus, reducing platform independence (the name of the dynamic libraries differ between e.g. Windows and UNIX systems).

XSLT Boxes

.NET comes with good XML and XSLT support. This offers a good basis to provide boxes to transform XML documents. Fig. 7 gives an example for the contents of *execute*.

```
  String xml_input = (String)
     ((Inputs[0].data.copy()).value);
  StringReader xml_reader =
      new StringReader(xml_input);
  XPathDocument xpath_document   =
      new XPathDocument(xml_reader);
  XslCompiledTransform transformation =
      new XslCompiledTransform();
  StringReader xsl_script_reader =
      new StringReader(Xslt_Script());

  XmlTextReader xsl_script =
    new XmlTextReader(xsl_script_reader);
  transformation.Load(xsl_script);
  StringWriter xml_output_writer = …
  XPathNavigator document_navigator =
       xpath_document.CreateNavigator();

  transformation.Transform(
       document_navigator, null,
       xml_output_writer);
  Outputs[0].data.value =
        xml_output_writer.ToString();
```

Figure 7. Apply XSLT script to input

The example takes some XML data from an input port and delivers transformed data to the output port. Note, that the XSLT script in this case is provided by a return value of *Xslt_Scipt*, a method to be overridden by subclasses to specify a concrete transformation. Other variants of XSLT boxes might expect the script itself, or a filename for the script as input at a port, or configured in another box view. A subclass of this box is used in Fig. 1 to transform the description of a labyrinth into Prolog notation.

Command Line Tools

Many transformations are available as command line tools. Examples are compilers, but also yacc, lex, awk. Additionally, there are tools like grammar deployment kit [Kor02], which could be made available through the integration in Trane. Figure 8 shows how to use the Java-compiler for Lisa-generated code (cf. Fig. 1). Here, the tool represented is hard coded into the box, but could also be provided through extra views with input fields or from input strings as part of the transformation.

```
System.Diagnostics.Process p =
                              new Process();
p.StartInfo.UseShellExecute = false;
p.StartInfo.CreateNoWindow = true;
p.StartInfo.RedirectStandardOutput=true;
p.StartInfo.RedirectStandardInput= true;
p.StartInfo.FileName = "cmd";
p.Start();
StreamWriter sw = p.StandardInput;
StreamReader sr = p.StandardOutput;
sw.AutoFlush = true;
//sw.WriteLine("dir /AD");or any cmd/tool
sw.WriteLine(@"javac –classpath lisa.jar"
                        +path+"*.java");
sw.Close();  p.WaitForExit();
Outputs[0].Data.Value=TextBuffer.Text;
```

Figure 8. Wrapping command line tools

The problem with this kind of boxes is that platform independence is restricted to the availability of the integrated tools on the platform.

Dynamic Compilation and Integration

The command line tool approach can be used to compile a transformation for Trane and make it usable at run-time. Depending on given options, the resulting executable can be started as command (maybe again wrapped in a box, as in Fig. 8), or the DLL can be examined/loaded and classes instantiated using reflection, if it is written in a .NET language. If the compiler generates .NET code itself, the resulting class can be directly instantiated instead of generating a DLL first.

F# and Other Languages

Though the above examples can use transformations written in other languages, the boxes themselves have been specified using C#. It is better to use the language of choice itself to define a box. This requires it is implemented on .NET. The resulting DLL can be

used in Trane, as if C# had been used due to cross-language inheritance. Only then *the real benefit* of .NET occurs in our opinion, as the still existing problems of data conversion in approaches like command line tools or foreign libraries could be avoided.

With F# [Fsh06] we were able to inherit from C# classes of Trane (the box), to create a new box (written in F#) and to instantiate from it in Trane again. F# is functional and thus, similar to Prolog, suitable to describe transformations.

Several languages on .NET are differently suitable. We had not the expected success with P#, but this might be our fault. With Eiffel# it is necessary to take care of the naming scheme during compilation. J# is not portable on Linux as it requires DLLs available on Windows only. We would be interested in a smooth integration of Haskell. There are some attempts, but there is still a way to go.

5. RELATED WORK

Several tools provide a plugin structure and interactive placement of components. They are either large, or provide a proprietary language to extend them with new objects. Trane has mainly been inspired by Cantata, the graphical user interface for the Khoros system to analyse and manipulate graphics [You95]. Cantata allows to interactively construct such filter pipelines.

[Spi02] considers UNIX tools as components. A GUI builder is used to create the visual programming environment. The placing relation of the components describes dataflow, which is text. UNIX tools have to encapsulate as ActiveX components with much manual work. Connectors are simply a visual encapsulation of the operating system pipe abstraction. Connector and glue-type components still need to be written by hand. Trane is not restricted to one kind of data, though it is intended to be applied mainly to artifacts of language processors, i.e. data are grammars, specifications, rewrite rules, parts of parsers, etc. We provide among others a system call box, which can take the command call directly as string. A new wrapper box for a special command can be easily written on top of the system box, which can take even the options at input ports. Our converters can transport structured data of any kind, they just have to inherit from a general converter class and implement additional treatment.

Stratego/XT [Vis04] uses mainly ATerms [Bra00] to provide input and output for terms in Stratego, and to exchange terms between transformation tools. New created transformations are wrapped into stand-alone components, which can be called from the command-line or from other tools. Those tools can be used similarly to Unix pipes, but can additionally work on structured data. For compositions of complex transformations they provide the XTC model. A repository registers locations of tools. An abstraction layer implemented in Stratego supports transparent access, allowing to call and use a tool like a basic transformation step in Stratego programs. Additionally, Stratego provides a foreign language interface to call C functions. Trane is designed mainly to reuse and to combine transformations for experiments. The XT tools could be wrapped in boxes, and used for experiments. We cannot generate stand-alone tools from composite transformations.

The Meta-Environment [Bra01] also allows the combination of different tools, but separates strictly between coordination and computation. Basis is the TOOLBUS coordination architecture, a programmable software bus based on process algebra. Coordination is expressed by a formal description of the cooperation protocol between components, while computation can be expressed in any language. Meta-Environment is used to produce real life products, on the other hand, it is complex, and difficult to adapt a new tool to the tool bus.

In Trane, coordination and computation are tangled. Evaluation of a transformation net is just traversing to each node and computing as given by the inherent dependencies between transformation nodes. Transformations can be added easily by providing a wrapper, where only two methods have to be overridden.

In Eclipse, GEF allows to create similar models and associate semantics to them. However, for new parts of the model (e.g. similarly to a new box in Trane) it requires a new compilation, while Trane nets are open. We do not need to compile the net. It is directly executable. New transformations can be added dynamically. Like other plugin systems, in Eclipse a plugin needs configuration files to add a new component, while we use reflection to extract necessary information. The language plugins for Eclipse are Java classes in a JAR archive. Transformations in Trane do not need to be written in one specific configuration language, as long it is supported by .NET.

[San99] also try to spread transformation system technology over a set of reusable heterogeneous components. Using Java, CORBA and HTTP, they have instantiated a communication layer. To configure components, a description in a hybrid architecture description language is necessary.

Calling functionality from foreign DLLs is not new. However, usually the calls are determined at compile time. We offer to combine functionality, which might come from different DLLs without recompilation.

Using Trane is similar to programming in dataflow languages. We refer to [Whi94] for further reading.

6. CONCLUDING REMARKS

SUMMARY

We have presented a lightweight infrastructure, which allows to provide heterogeneous transformations with a uniform façade to combine and interact with them. The model has been given and the essential classes have been explained. We presented five categories of transformations such as integration of web services, or command line tools. Integration of new transformations is simple. Due to reflection, no extra configuration files are necessary. Trane is lightweight as a large part of the work for integration is encapsulated in .NET. The biggest advantages have languages that are implemented on .NET directly, but we still wait for more pure .NET languages, without name scheme or inheritance problems.

FUTURE WORK

We are aware that Trane is rather a proof of concept than a tool yet. The type system is currently very ad hoc. There are still conceptual. It is still matter of research, what types mean in our context. For example, for some transformations grammars of different languages are of the same type, if they are in the same format such as BNF. On the other hand, grammars can be considered as different types despite their format, if the algorithm using it is language specific. We want to design an extensible type hierarchy.

The Visitor pattern might help with flexible computations; also, to generate command line tools from a net as well as terms describing nets for analysis. As another way to integrate transformations sockets should be examined. The usability has to be increased vastly. It might be interesting to initiate the evaluation of transformations in separate threads. A classification of boxes would be nice. We need more transformations with grammar typical support to perform the experiments. We are new to F# and need more experiments with it and with other .NET languages.

7. ACKNOWLEDGMENTS

We thank the reviewers for their comments, which provided answers, literature, and suggestions for future directions of the work, though we were not able to implement most of them in this paper. We thank Damijan Rebernak for help with the Lisa web service.

8. REFERENCES

[Bra01] v. d. Brand , M.G.J., and v. Deursen, A., and Heering,J, and de Jong, H.A., and de Jonge, M., and Kuipers,T., and Klint,P., and Moonen,L., and Olivier, P.A., and Scheerder,.J., and Vinju,J.J., and Visser, E, and Visser,J. The ASF+SDF Meta-environment: A Component-Based Language Development Environment, Procs. of the 10th International Conference on Compiler Construction, p.365-370, April 02-06, 2001

[Bra00] v. d. Brand , M.G.J., and de Jong, H. A., and Klint, P., and Olivier, P. A., Efficient annotated terms, Software- Practice & Experience, 30, pp. 259-291, 2000

[Fsh06] F# Home Page (Feb.2006) http://research.microsoft.com/fsharp

[Klu05] Klusener, S., and Lämmel, R., and Verhoef, C.: Architectural Modifications to Deployed Software. Science of Computer Programming 54, pp.143-211, 2005

[Kor02] Kort, J. and Lämmel, R., and Verhoef, C. The Grammar Deployment Kit, ENTCS 65, 3, Elsevier Science Publ., 2002

[Läm04] Lämmel, R.: Evolution of Rule-Based Programs. Journal of Logic and Algebraic Programming, Special Issue on Structural Operational Semantics, 2004

[Les03] Lesta, U.: C# Interface to SWI-Prolog. http://gollem.science.uva.nl/twiki/pl/bin/view/Foreign/CSharpInterface, Version Aug. 2003

[Loh03] Lohmann, W., and Riedewald, G. Towards automatical migration of transformation rules after grammar extension. In Proc. 7th European Conference on Software Maintenance and Reengineering (CSMR'03), Benevento, Italy, March, 2003

[Loh04] Lohmann, W., and Riedewald, R. and Stoy, M. Semantics-preserving migration of semantic rules during left recursion removal in attribute grammars, ENTCS 110 C, Elsevier, 2004

[Mer99] Mernik, M., and Zumer, V., and Lenic, M., Avdicausevic, E. Implementation of multiple attribute grammar inheritance in the tool LISA. ACM SIGPLAN not., June 1999, Vol. 34, No. 6, pp. 68-75.

[San99] Sant'Anna, M., do Prado Leite, J.C.S., An Architectural Framework for Software Transformation, Proceedings of the International Workshop on Software Transformations; STS'99', ICSE'99, 1999 http://www.dur.ac.uk/CSM/STS/

[Set04] Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004), ENTCS 127 (3), April 2005

[Spi02] Spinellis, D. Unix tools as visual programming components in a gui-builder environment. Software - Practice & Experience. 32, pp.57-71, 2002

[Vis04] Visser, E., Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9., in C. Lengauer et al., editors, Domain-Specific Program Generation, LNCS 3016, pp. 216--238. Spinger-Verlag, June 2004.

[Whi94] Whiting, P. G., and Pascoe, R. S. V. A History of Data-Flow Languages, IEEE Annals of the History of Computing, Vol.16(4), pp.38-59, 1994

[Wie06] Wielemaker, J. SWI-Prolog Home Page http://www.Swi-Prolog.org

[You95] Young, M., and Argiro, D., and Kubica, S. Cantata: Visual programming environment for the Khoros system. Computer Graphics 29, 1995

Sampling profiler for Rotor as part of optimizing compilation system

Sofia Chilingarova
St-Petersburg State University
28, Universitetskiy pr.,
Petrodvorets
Russia 198504, St-Petersburg
sofie-chil@hotmail.ru

Vladimir O. Safonov
St-Petersburg State University
28, Universitetskiy pr.,
Petrodvorets
Russia 198504, St-Petersburg
v_o_safonov@mail.ru

ABSTRACT

This paper describes a low-overhead self-tuning sampling-based runtime profiler integrated into SSCLI virtual machine. Our profiler estimates how "hot" a method is and builds a call context graph based on managed stack samples analysis. The frequency of sampling is tuned dynamically at runtime, based on the information of how often the same activation record appears on top of the stack. The call graph is presented as a novel Call Context Map (CC-Map) structure that combines compact representation and accurate information about the context. It enables fast extraction of data helpful in making compilation decisions, as well as fast placing data into the map. Sampling mechanism is integrated with intrinsic Rotor mechanisms of thread preemption and stack walk. A separate system thread is responsible for organizing data in the CC-Map. This thread gathers and stores samples quickly queued by managed threads, thus decreasing the time they must hold up their user-scheduled job.

Keywords

SSCLI / Rotor, Just-in-time compilation, sampling-based profiling, de-virtualization, inlining.

1. INTRODUCTION

Optimization techniques based on profile data obtained at run time form the essential part of optimization strategy in modern dynamic compilation frameworks.[Arn02][Sug01][Jav02] Static analysis alone cannot provide sufficiently full information by sufficiently low cost to make optimizations pay for themselves in dynamic compilers. Managed environments have the distinguishing capability to provide feedback and use it in compilation at the very time the program executes, and runtime profilers are designed to utilize this capability. With profile data enabling selective optimization of the "hot" pieces of code we gain much more.

There are two main types of profile data optimizing compiler may be interested in: individual methods "hot counts", i.e. precise or approximate estimation of method execution frequency, and some kind of "call graph" which can provide information about the frequency of calls from one method to another. The former is used to pick up the individual "hot" methods for recompilation, the later helps to plan optimizations in the broader context taking into account the hot paths through the whole application.

Many techniques have been developed to collect and store runtime profile data. But the key point has always been a balance between the accuracy of profile data and low overhead of profiling facilities, which have to do their job at run time thus adding to compilation overhead. Experiment results show that strictly accurate profile is not necessary to make a good recompilation decision, so sampling profilers turned out an excellent tool to get rather complex information about program behavior with low overhead.

Typical sampling profiler working as a part of a dynamic compilation framework acts as follows: periodically it launches a task that looks up a stack for managed methods frames, then forms collected data into some structure reflecting dynamic call context and stores it for the subsequent use. [Arn02] [Wha00] Our profiler developed for SSCLI (Rotor) also utilizes this classical schema. It uses the mechanism for exploring stack that Rotor already has (we will cover it later) and stores data in Call Context

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

Map structure that contains counters for individual methods calls, total count for every call done by one method to the other, and detailed information about the context in which the call occurs.

Contributions

This paper makes the following contributions:

- **Data structures.** It describes a Call Context Map (CC-Map) data structure used for encoding runtime profile information. It shows the advantages of the Call Context Map: its capability to provide the full information needed for recompilation decisions quickly and with minimum effort remaining at the same time a rather compact structure. It describes the algorithm for filling CC-Map from a raw stack samples containing references to managed methods metadata and offsets in code.
- **Profiling Techniques.** The paper presents a profiling technique based on profiler and managed threads cooperation and background processing of raw samples data, which allows maintaining a complex structure of profile data storage not causing the managed threads to postpone their jobs for a long time. The bunch processing of samples helps to minimize synchronization on the global samples cache.
- **Experience using SSCLI features.** The paper shows how the SSCLI core functions and structures were used to help collecting stack samples and organizing profile data. It also describes the utilization of core SSCLI mechanism for threads cooperation and synchronization to provide cooperative behavior in gathering samples.
- **Evaluation of overhead and accuracy of profiling.** The paper presents evaluation of accuracy and overhead of the profiler ran on the SSCLI quality test suit using simple execution counters statistical correlation and Arnold & Ryder overlap percentage measure[Arn02].

2. RELATED WORK

Many papers published in the last years show the benefits of profile-driven optimizations and the ways profile data may be used in compilation decisions. Arnold[Arn02] in his PhD thesis paper describes in detail several kinds of profile-driven optimizations implemented in Jikes JVM. Suganuma et. al. [Sug01] in their review of IBM DK optimizing JIT-compilation framework give a full picture of how instrumentation and sampling based profiling is used to collect profile data from interpreted and compiled code, respectively. They provide experiment results showing the evident advantages of profile-based selective optimizing compilation over both optimizing non-selective and fast non-optimizing non-selective compilation.

Several studies show the practical use of dynamic profile data in such optimizations as inlining [Sug02] and devirtualization[Ish00]. These two types of optimization are very important for managed environments with intrinsic support of object-oriented languages where most method calls are virtual and many levels of indirection often present. Suganuma et. al. [Sug03] introduce an interesting optimization technique, Region-Based Compilation, that allows more effective use of profile data.

Whaley[Wha00] describes several different approaches to profile data organization: Dynamic Call Graph (DCG), Calling Context Tree (CCT), Partial Calling Context Tree (PCCT). Arnold et. al. [Arn00] shows in more detail how the DCG is constructed. We'll look closer at these structures in the next section where we describe our data representation choice, Call Context Map (CC-Map), and compare it with the other options. CC-Map is in many respects similar to CCT and PCCT, but provides easier ways to retrieve full context information. Also we don't place such restrictions on the length of a sample, as PCCT-based approach described by Whaley. In our profiling framework we allow sample buffers to grow when needed, although we define some rather high limit for the cases of incredibly deep stack, which are rare.

Arnold and Grove [Arn05] propose an interesting variation of samples collection technique. Instead of taking one sample at a time, their profiler takes a bunch of samples: when profiling is requested, stack walk is performed several times over a short interval. Authors show how this approach helps eliminate inaccuracy in some situations.

3. PROFILER DESIGN

In this section we describe an overall structure of the profiler: how the sample data storage is organized and how the samples gathering mechanism works. We introduce a Call Context Map (CC-Map) that allows easy retrieving of many kinds of data needed for compilation/recompilation decisions. We present a sampling strategy that helps to maintain a rather complex CC-Map structure and at the same time not cause the user threads job to be postponed for long intervals. In the next section we'll take a closer look at the Rotor-specific issues and show how the profiler uses intrinsic mechanisms of the SSCLI virtual machine to do its job.

Call Context Map

3.1.1 Previous approaches

The common way to represent the sequences of calls with their relative frequency in runtime profile data is using some kind of call context tree. Call context tree consists of nodes correspondent to the method calls and directed edges, which denote caller-callee relations. The examples are Dynamic Call Graphs (DCG, DCG-E) described by Arnold et. al. [Arn00] and Calling Context Tree/Partial Calling Context Tree (CCT, PCCT) described by Whaley[Wha00] . Dynamic Call Graph is shown on the Figure 1b. Nodes represent method calls, edges mark associations between caller and callee, and weights assigned to edges mean the number of calls from the specified caller to the specified callee encountered in samples. This is rather compact representation but the information we can retrieve from it is limited. We can estimate how often one method calls the other, but with DCG alone we cannot determine, for example, that call chain ACD has never been encountered in samples, ABC has been encountered 2 times, and BCD – only once. Thus DCG can effectively represent only one-level-depth profile.

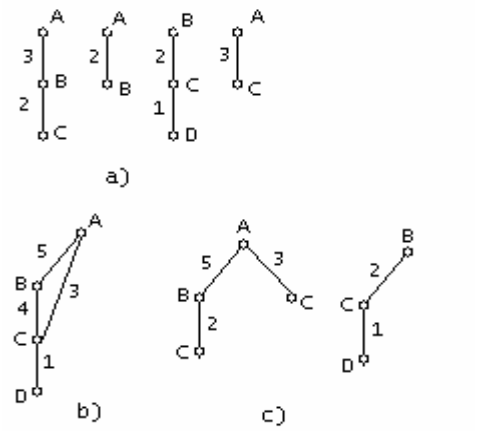

Figure 1. DCG and PCCT structures: a) samples collected from stack; b) correspondent DCG; c) correspondent PCCT

Partial Calling Context Trees (CCT) shown on Figure 1c provides more context information. Details of PCCT construction are covered in [Wha00]. They build PCCT using the fixed length buffer for samples, so that a delay does not be very long when the stack is extremely deep. When a sample is got and a PCC-Tree with the outer caller as a root is found, profiler updates counters for edges in this tree, otherwise a new tree is created. Here we can point out longer call sequences, but still cannot know, without additional analysis, that calls from B to C have been encountered 4 times, totally. To retrieve this information we should examine all the trees looking for edges BC and adding the counters to the total sum.

One more problem is illustrated by Figure 2a. Let we have a call graph shown at the left side of the figure. A and E call B and in both cases B calls C. Then C calls D or F. Also the samples with B as the outer frame are found, as shown on the figure. Let we build the Call Context Trees form these samples. We get three of them, with A, E, and B as roots.

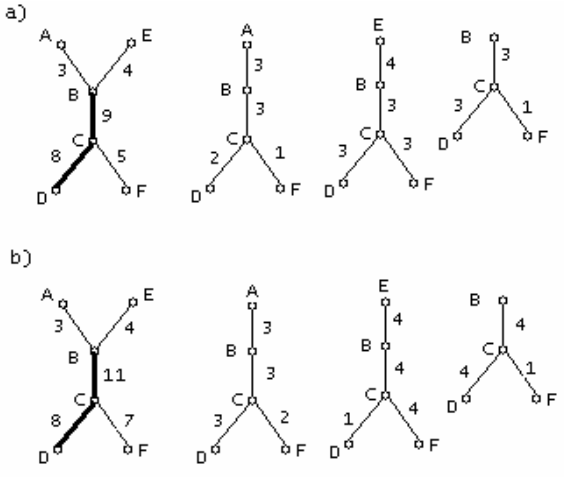

Figure 2. More complex call context

Here the hottest path is actually BCD, which executes 8 times. But we cannot retrieve this information automatically having only the CC-trees in hand. We cannot queue BCD path for possible recompilation automatically when the total counter exceeds threshold because we haven's such a total counter. The solution might be to construct/update CCT for every caller in the chain when a sample is got, but this way we fail to distinguish the frequencies of call to BCD in different contexts. For example, if the situation is like the one shown on Figure 2b, we'll fail to know that BCD path (executes 8 times totally) is actual only for calls from A. For E call site the path EBCF is really hot. The PCC-trees for this case (3 trees shown at the right side of the Figure 2b) reveal it clearly. If we update counters for BC and CD in the tree with B root every time the path is encountered in a sample, at any place, we capture the information about the total number of execution of BCD, but loose the important context information. So we need some combination of the described approaches.

3.1.2 Call Context Map Structure

Call Context Map (CC-Map) structure is designed to address issues depicted in the previous subsection. The higher level of the CC-Map is a hash-table containing references to *MethodProfile* nodes. *MethodProfile* node stores a total counter for the method executions and references to the nodes

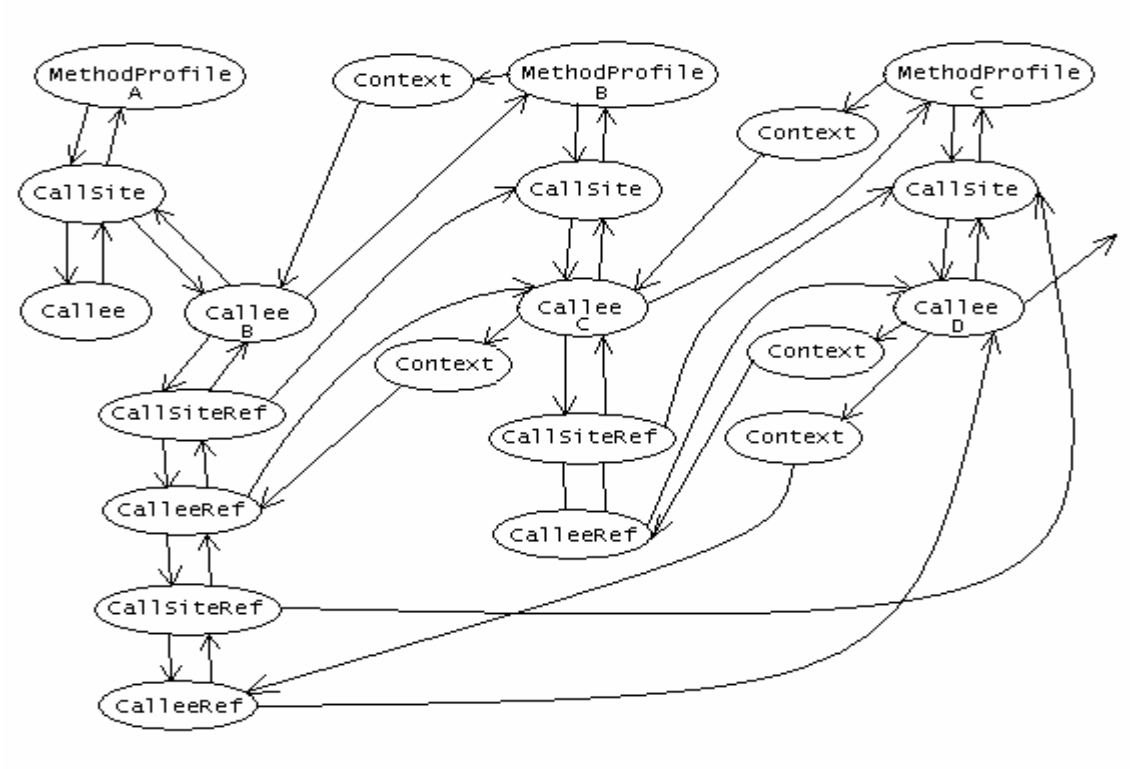

Figure 3. Call Context Map fragment

representing information about calls from this method to the others.

The *Callee* nodes contain accumulated counters for the total number of calls from the concrete caller to the concrete callee, in any context. Additionally, the tree of reference nodes is constructed for every call sequence. These *Ref* nodes contain counters for calls done in the given context and references to the nodes, which store general information about the call.

A fragment of CC-Map structure is shown on Figure 3. Let method A calls method B, B calls C, and C calls D. Every caller profile refers to *CallSite* node that contains general information about the call site – offset, reference to the caller profile, etc. *CallSite* node refers to one or more *Callee* nodes, which store call counters and, in turn, refer to the profiles of callees. *CallSiteRef* and *CalleeRef* nodes refer to the general *CallSite* and *Callee* nodes and *CalleeRef* nodes store the context counters. Every node representing general call information has *Context* references to the nodes, which describe a context of the call.

3.1.3 Advantages of CC-Map structure

CC-Map accumulates a total call count for every caller-callee pair and at the same time it allows retrieving information concerning calls in the specific context. This information is easily available: a compilation controller may lookup contexts by the *Context* references when some counter exceeds a threshold, as well as move up and down through a call chain.

From the *CallSite* and *CallSiteRef* nodes a controller can know whether the call has probably one target (and so consider devirtualization). *CallSite* node provides this information for all calls from a given site, *CallSiteRef* – only for calls done in a given context.

CC-Map is a rather compact structure. Nodes don't store duplicate data. CC-Map allows quick updating, as well as rather quick removing of nodes, which appear cold. Compilation controller need not perform additional analysis of trees to get information necessary for good decision: it can only follow references.

Figure 4 shows an example: a simplified view of CC-Map for the calling sequences presented on Figure 2a and 2b. The *CallSite* nodes are omitted for simplicity, as there is only one call site for each method in this example. You can see that a bi-directional association exists between a node with general information about method call and nodes representing the same call in the different contexts. When an event of a total counter exceeding threshold takes place, a compilation/recompilation controller can quickly look through the contexts to make an appropriate

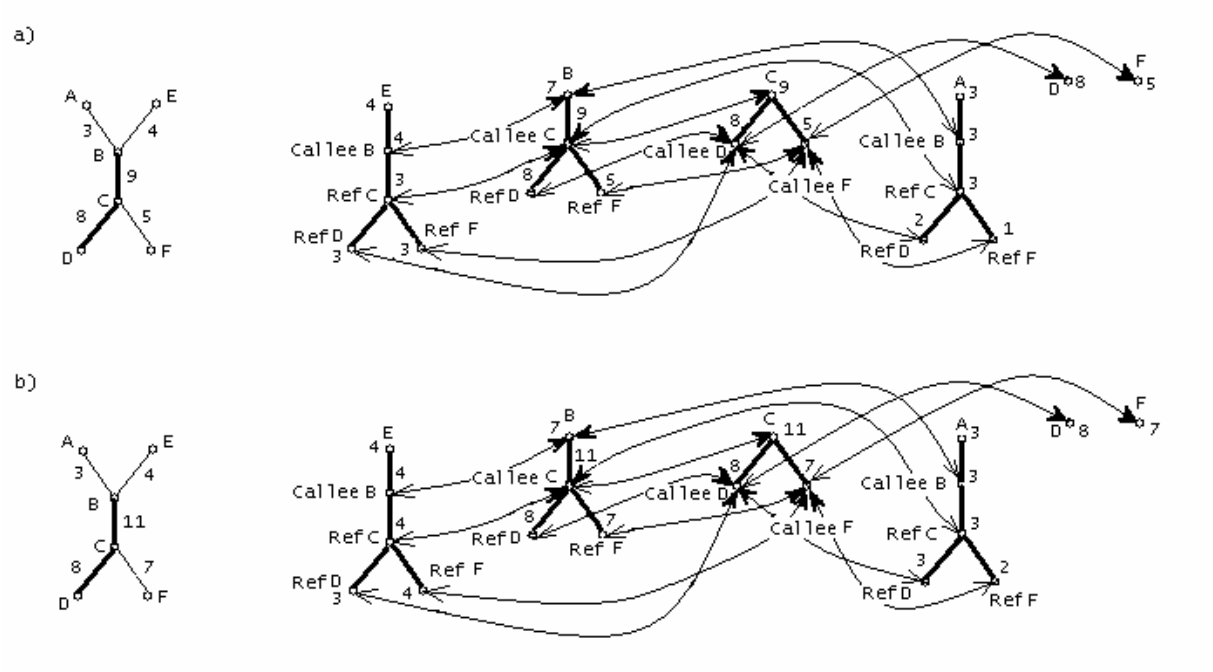

Figure 4. CC-Map for Fig. 2 examples. Bold arrows indicate references from nodes describing call in a given context, thin arrows indicate references form a general information node to call-in-context nodes (this association is represented by "Context" items on Fig. 3). The roots of the trees are MethodProfile nodes containing the total counters for method executions

compilation decision (for example, consider the common callers for de-virtualization or inlining too, especially if only one callee has been detected at the correspondent call sites so far). When analyzing a frequently executed call sequence a controller can browse all general call information nodes and access other contexts from them. It can move up and down the call sequence representation (see Fig. 3) to gather all the information about callers and callees that might affect a recompilation strategy choice.

3.1.4 CC-Map filling and updating

When a sample is being taken, all the data initially is written into a buffer. The stack lookup starts from the top of the stack and ends at the outermost frame or at the first managed method activation record that has already been visited by profiler. The profiler marks managed method activation records when looks them up (the JIT-compiler is configured to push the additional slot on the stack for this purpose), so during the following passes it can distinguish the new frames from the old ones. When the profiler encounters an old (marked as already visited) frame, it records this frame data (as it is needed to register a new call from the frame) and stops looking up the stack.

So, at the start of the buffer we have a reference to the method correspondent to the activation record at the top of the stack (i.e., most inner call), and at the end of the buffer – the outer caller (or the innermost call that hasn't returned from the previous lookup) reference.

The pseudocode for sample buffer processing looks as follows:

```
For(int i = 0; i < end_of_sample; i++)
{
   update MethodProfile(buf[i]);
   if (i > 0)
   {
      update Callee(buf[i],buf[i-1]);
   }
   for (j = i-2; j>=0;j--)
   {
      update CalleeRef(buf[j]);
   }
}
```

The real code is a little more optimized and a little more complicated, but the underlying algorithm is the same.

Profiling Algorithm

Maintaining such a complex structure as the CC-Map requires some effort. Algorithm described in the previous section may take a long time to complete. But we cannot afford to stop user threads for observable intervals because of profiling.

The solution we have chosen is to separate taking sample from thread stack from storing the sample data in the CC-Map. For this purpose we use two profiler worker threads, as well as thread-local and

global queues for samples waiting for the profiler to process them.

Profiling job is launched by the *MarkThreadsWorker* system thread which marks every live managed thread to make it know that it should take a sample when reaches a safe point. Every live managed thread has its own sample buffer and its own short samples queue. The sample is written into the thread local buffer and pushed into the thread local queue. When local queue length exceeds a threshold (rather low, now 10) all its contents is pushed to the global queue. This schema is aimed to decrease the need to grab a global queue lock, and thus to decrease possible pauses caused by waiting for the lock. Little delay in samples processing is not critical because only large numbers are considered when making compilation decisions.

The CC-Map manager thread periodically grabs the global queue lock, takes out a bunch of samples and put them into its own queue. Then it releases the lock and proceeds with processing samples without hurry. Global queue hashes samples by thread id so the CC-Map manager thread can return the processed sample buffers back to their thread so that it need not to allocate new memory. Local thread buffer grows automatically when needed, queued samples buffers grow then they need to adapt to local buffer size. So when threads get back their own buffers, previously queued, these buffers are likely to have appropriate size. If the thread is already finished when CC-Map manager returns processed sample buffers for it, this chain of buffers is put aside to be used by next new thread.

Tuning Sampling Interval

The profiler is, *self-tuning*, it adapts an interval of taking samples to the characteristics of environment where it runs. To do this it uses a simple heuristics: it tracks how often the same activation records appear on the top of the stack. It doesn't take much effort or time: as the profiler already distinguishes between visited and not visited frames and stops at the first visited, we need only to reflect this condition in a sample and check whether this frame is the first in a sample (i.e. it is taken from the top of the stack) when processing the sample. If so, a special counter is incremented.

There are two threshold values defined: maximum percentage of repetitions and minimum percentage of repetitions. CC-Map manager thread evaluates actual percentage of repetitions (of activation record appearance on the top of the stack) every 1000 samples (more precisely, than processed samples portions is more than 1000, because the manager thread handles a bunch of samples in every pass). If percentage of repetitions is lower then minimum threshold, it is considered too low and sampling interval decreases. If percentage of repetitions is higher than maximum threshold, the sampling interval increases.

4. INTEGRATION WITH ROTOR

Rotor has a built-in mechanism for walking the stack, which is used for such purposes as exception handling and security checks[Stu03]. It involves several methods and functions of virtual machine and among them the *StackWalkFrames* method of the VM *Thread* class, which we use to take samples. *StackWalkFrames* takes a function to execute on every encountered stack frame as a parameter, so its work is easily customizable. The advantage of using it is that it already knows how to distinguish managed method frames from unmanaged method frames, can recognize context transitions (e.g. across application domain boundaries), encapsulates calls to Rotor facilities to get metadata references and offsets, and it provides a convenient interface to do jobs on the stack.

We make managed threads call *StackWalkFrames* method at, so called, "safe points", building upon the other intrinsic Rotor mechanism – trapping threads when they know that it is safe to suspend now. This mechanism has been originally used to trigger garbage collection. Checks for a suspension request have been inserted by the JIT-compiler at back edges and everywhere where the next piece of code may take long time to execute[Stu03]. Such checks are also performed by some of runtime helper functions extensively used in Rotor. We utilize this mechanism and add additional check points at the entry of every method. At that new check points we test only for the need to take sample.

We also used the SSCLI core HashMap class to construct the CC-Map in Rotor. SSCLI HashMap class implements a hash table used by VM for its internal needs. It hashes pointer type values by the pointer type keys (so allows storing profile objects by the pointer-to-metadata keys), implements locking for insert, delete and lookup, and takes care of cleaning up itself. It is just what we need. So we choose HashMap as a hash table to store MethodProfile references at the highest level of CC-Map and as a hash table to hold queues of samples waiting for processing in the global samples store.

5. RESULTS

We tested our profiler on SSCLI 1.0. To measure overhead and accuracy of profiling we used tests from a suite supplied with SSCLI. To estimate overhead we chose a set of base tests from *bcl\system*

and *bvt* subdirectories and tests from *bcl\threadsafety* subdirectory of Rotor *tests* directory. To estimate accuracy we used tests from *bcl\threadsafety* subdirectory, where multiple threads execute the same code. As measures we used statistical correlation of the total executions counters stored in *MethodProfile* nodes and Arnold & Ryder overlap percentage[Arn02] for the whole tree comparison. Overlap percentage of trees T1 and T2 is computed as follows:

$\sum_{N \text{ in } T1,T2}$ [min (Weight (N_{T1}), Weight(N_{T2})]

where Weight (N_{Tx}) is:

value(N_{Tx})/$\sum_{N \text{ in } Tx}$value(N),

N is a node holding a counter, value is a value of the counter. When N is not found in Tx (thought it exist in Ty and thus in TxTy set), it is assumed that value(N_{Tx}) = 0.

For performance test the low threshold for repetitions (cases when the same method appears on the top of the stack) was set to 1%, high threshold for repetitions was set to 15%. For the correlation and overlap measurement tests the self-tuning was turned off, because it can affect the correlation results distinctly for short-running tests, as those we used. However the great deal of these differences is produced at the interval when the profiler is tuning, so such results do not reflect the real picture in steady state. Logging of sample interval changes in the process of tuning revealed that the sample interval becomes stable after 1-2 changes. We measured correlation and tree overlap with different sample intervals (with self-tuning turned off) and the best results (95-99%) were obtained with the same interval that the profiler found automatically.

In accuracy test we recorded and compared executions counters and the whole CC-Maps from 10 subsequent runs. The results of every run were compared with results of every other and an average value was computed.

To make the CC-Map accessible even after the VM was stopped running, we dumped the CC-Map (in the *fastchecked* mode) to an XML file at VM shutdown. Then original CC-Maps were restored from XML representation and compared (in XML dump of CC-Map managed methods are identified by the full name and signature to make comparison possible, though at runtime they identified only by pointer to metadata).

Table 1 shows the average correlation for 10 subsequent runs of the same test and average tree overlap percentage. All the tests are from *bcl\threadsafety* suite.

Test Name	Correlation, %	Overlap, %
co8545int32	99	97
co8546int16	99	92
co8547sbyte	99	94
co8548intptr	99	98
co8549uint16	99	95
co8550uint32	99	95
co8551byte	99	97
co8552uintptr	99	97
co8553char	99	96
co8555boolean	99	96
co8559enum	98	75
co8788stringbuilder	99	67
co8827console	99	77
co8830single	99	98

Table 1. Average correlation for total executions counters and overlap percentage extracted from comparison of results of 10 subsequent runs

We can see that though the correlation of simple execution counters is always good (98-99%), overlap percentage sometimes appears lower than 80%. We think, however, this can be probably explained by the fact than the tests themselves were very short.

Tests were run on Celeron433 processor, 256M RAM. Sampling interval was set to 10ms. This is rather short interval for this hardware configuration and for long-running programs in may be longer. However, the tuning mechanism can adjust the interval well. When testing we started from interval 50ms, and for the tests, which performed bad with such an interval, the profiler made it less. For the tests, which performed well, the interval remained unchanged. We see also in Table 1, that for some tests accuracy is even redundant. 95-97% would be enough to consider results statistically significant. For the cases when we can get such accuracy with longer interval, it will not decrease (or it can even increase if the initial interval appears too short).

The profiling overhead was measured on the free build against unchanged Rotor free build, on the same hardware configuration, on the tests from *bcl\system, bvt*, and *bcl\threadsafety* subsets of Rotor core test suit. Initial sampling interval was set to 50ms. Tuning was turned on. Tests were run 2 times, and the total overhead did not exceed 3%. In the future we intend to consider automatic turning off tuning after a certain period of time so that to lower overhead.

6. REFERENCES

[Arn00] Arnold, M., Fink, S., Sarkar, V., Sweeney, P. A comparative study of static and dynamic heuristics for inlining. In ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, Jan. 2000.

[Arn02] Arnold, M. Online Profiling and Feedback-Directed Optimization of Java. PhD thesis, Rutgers University, October 2002.

[Arn05] Arnold, M. and Grove, D. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In Proceedings of the international Symposium on Code Generation and Optimization, March 20 - 23, 2005.

[Jav02] The Java HotSpot™ Virtual Machine, v1.4.1, d2, A Technical White Paper. Sun Microsystems, September 2002.

[Ish00] Ishizaki, K., Kawahito, M., Yasue T., Nakatani, T. A study of devirtualization techniques for a Java just-in-time compiler. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Oct. 2000.

[Stu03] Stutz, D., Neward, T., Shilling, G. Shared Source CLI Essentials. O'Reilly, 2003.

[Sug01] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., Nakatani, T. A dynamic optimization framework for a Java just-in-time compiler. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 2001.

[Sug02] Suganuma, T., Yasue, T., Nakatani, T.: An empirical study of method inlining for a Java Just-In-Time compiler. In: Proceedings of USENIX 2nd Java Virtual Machine Research and Technology Symposium (JVM'02), pp. 91–104, 2002.

[Sug03] Suganuma, T., Yasue, T., Nakatani, T., A Region-Based Compilation Technique for a Java Just-In-Time Compiler, ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003), pp. 312-323, June 9-11, 2003.

[Wha00] Whaley, J. A portable sampling-based profiler for Java virtual machines. In ACM 2000 Java Grande Conference, June 2000.

State Machine Design Pattern

Anatoly Shalyto
Head of Programming Technologies Department
St. Petersburg State University of Information Technologies, Mechanics and Optics
14 Sablinskaya Street
Saint-Petersburg, Russia 197101
shalyto@mail.ifmo.ru

Nikita Shamgunov
Software Design Engineer, SQL Server Engine, Microsoft,
11407 183rd Pl NE #M1071
USA 98052, Redmond, WA
u04921@mail.ru

Georgy Korneev
Assistant Professor of Programming Technologies Department
St. Petersburg State University of Information Technologies, Mechanics and Optics
14 Sablinskaya Street
Saint-Petersburg, Russia 197101
kgeorgiy@rain.ifmo.ru

ABSTRACT

This paper presents a new object-oriented design pattern — *State Machine design pattern*. This pattern extends capabilities of State design pattern. These patterns allow an object to alter its behavior when its internal state changes. Introduced event-driven approach loosens coupling. Thus automata could be constructed from independent state classes. The classes designed with State Machine pattern are more reusable than ones designed with State pattern.

Keywords

design, pattern, automaton, automata, finite automata, finite state machine, behavior, state, transition, state chart

1. INTRODUCTION

Finite automata have been widely used in programming since the appearance of [Kle56] which introduced regular expressions and proved an equivalence of a finite automaton and of a regular expression.

Another area where finite automata are widely used is object oriented programming, in which they are used to design object logic. In this area states that have major impacts on object's behavior (*control states*) are being extracted. Note that these automata are significantly different from those used for regular expression matching. In particular, objects are designed in terms of interfaces and methods (terms that don't exist in classical automata) not in terms of recognizable strings. This paper discusses automata that are used in *OOP*.

In *OOP*, when people think of object behavior, they consider the functionality of its methods. But in many real world applications this definition is insufficient — the internal state of an object should also be considered.

The most famous implementation of an object whose behavior depends on its state is the *State* pattern [Gamma98]. However, pattern description is far from being complete, in different sources [Ster01, Gra02] it is implemented in different ways, sometimes even too verbose. Another disadvantage of the pattern is that the implementation of states in different classes causes distribution of the transition logic among these classes. This adds dependencies between the state classes which lead to different issues in class hierarchies design. In spite of these issues State pattern is used in many practical projects including *JDO* [JDO01].

This paper addresses issues of *State* pattern by introducing a new pattern named *State Machine*. Note that [San95] introduced a pattern with the same name for parallel system programming in *Ada95* but still the authors have chosen this name.

To make reuse of state classes possible we introduce an event mechanism. Events are used to let the automaton know that the state should be changed. This allows centralization of the automaton transition logic and loosens coupling between state classes.

 ISBN 80-86943-11-9

More than twenty possible implementations of *State* pattern are described in [Ada03]. *State Machine* pattern might continue this list. The closest pattern from the list is a combination of *State* and *Observer* patterns [Odr96]. However, this pattern is too complicated and it also introduces a new abstraction layer: `ChangeManager` class. In contrast to relatively verbose *Observer* implementation, in *State Machine* transitions between states are based on event-based mechanism. In [San95] another implementation of *State* was introduced. *State* classes coupling was loosened through a state change mechanism based on a state name. This implementation doesn't reduce semantic dependencies between classes and doesn't provide type safety.

2. Pattern Description

Intent

An intent of *State Machine* is the same as an intent of *State*: to make it possible for an object to alter its behavior when its internal state changes (it looks like an object has changed its class). More extensible design is required, than one provided by *State*.

Note that in the intent description so called *control states* are considered. The difference between *control* and *evaluation* states can be illustrated in the following example. In an imaginary bank management system it might make sense to identify two modes: normal mode and bankrupt mode. This modes would be *control* states. On the other hand particular amount of money on the clients' accounts would be an *evaluation* state.

Motivation

Consider a class `Connection` that represents a network connection. A simple connection has two control states: `Connected` and `Disconnected`. A transition between these states occurs either in case of an error or intentionally — via execution of methods connect or disconnect. In the `Connected` state a user can call methods `send` and `receive` of a `Connection` object. In case of an error `IOException` is thrown and connected breaks. If an object is in the `Disconnected` state, `send` and `receive` methods will throw an exception as well. Consider an interface, implemented by `Connection` class.

```
public interface IConnection {
   public void connect();
   public void disconnect();
   public int receive();
   public void send(int value);
}
```

The basic idea of *State Machine* is to separate classes which implement transition logic (*Context*) and state classes. To provide an interaction between *Context* and state classes we use events which are basically objects that state objects pass to *Context*. A difference from the *State* pattern is the way the next state is determined. In *State* next state is explicitly pointed out by the current state. In the proposed pattern it is done by notifying the *Context* with an event. After that it's a *Context's* responsibility to react and possibly change the state. This is done according to the state chart.

The advantage of this design solution is that state classes may be designed independently. They don't need to be aware of each other.

Note that the state charts that are used in *State Machine* are different from those described in [Aho85].

They consist only of states and transitions marked with events. Transition from the current state S to the next state S* occurs on receiving event E if there is a corresponding transition in the state chart.

State chart for the `Connection` class is shown on figure 1.

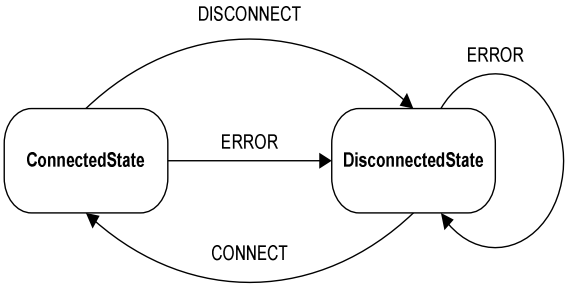

Figure 1. State Chart for class Connection

State classes are called `ConnectedState` and `DisconnectedState`. Event `CONNECT` is used to establish a connection and event `DISCONNECT` is used to break it. `ERROR` is used to indicate an i/o error.

To illustrate the work of the network connection let us take a closer look at its breach in case of an i/o error. If it were implemented through *State* its `ConnectedState` would tell context to switch to `DisconnectedState`. In the *State Machine* case it notifies the *context* through `ERROR` that an i/o error has occurred and the *context* changes its current state. Thus in *State Machine* case `ConnectedState` and `DisconnectedState` classes are not aware of each other.

Application

State Machine could be applied wherever *State* is applied but it also provides additional level of flexibility allowing to reuse the state classes in different automata. It also allows building state class hierarchies.

Structure

Figure 2 shows a structure of *State Machine*.

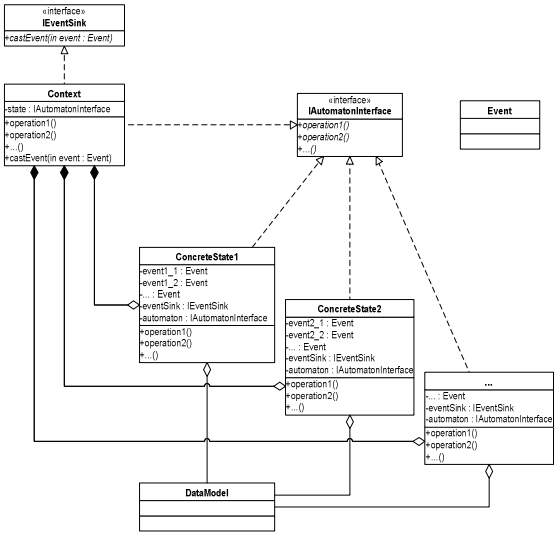

Figure 2. Structure of *State Machine*

`IAutomatonInterface` is an interface of an object to implement, `operation1`, `operation2`, … are the methods of this interface. This interface is implemented by the main class `Context` and by the state classes `ConcreteState1`, `ConcreteState2`, … . Events `event1_1`, `event2_1`, …, `event2_1`, `event2_2`, …, are used to change state. They are instances of the `Event` class. The `Context` class has references to all of the state classes (`ConcreteState1` and `ConcreteState2`) and a reference to the current state. The state classes have a reference to the data model (`dataModel`) and to the event notification interface (`eventSink`). For the purpose of brevity, relations between the state classes and the `Event` class are not shown in the figure.

Members

State Machine consists of the following parts.

- *Automata interface* (`IAutomatonInterface`) — is implemented by the context and is the only way of interaction between the automata and a client. This interface is also implemented by state classes.
- *Context* (`Context`) — is a class that encapsulates transition logic. It implements the automata interface and holds an instance of the data model and the current state.
- *State classes* (`ConcreteState1`, `ConcreteState2`, …) — determine behavior in a particular state. Each of them implements the automata interface.
- *Events* (`event1_1`, `event1_2`, ...) — initiated by the state classes and passed to the context that does a transition depending on the event and the current state.
- *Event notification interface* (`IEventSink`) — implemented by a context. This is the only way of interaction between the state classes and the context.
- *Data model* (`DataModel`) — is a class to provide a shared storage between the state classes.

Note that automata interface in the proposed pattern is implemented by the context and by the state classes. This allows making certain compile-time consistency check. In the *State* pattern such a check is impossible because the context interface doesn't match state classes' interfaces.

Relations

During its initialization the context creates an instance of data model and uses it to create instances of states. It passes the data model an event notification interface (which is a *this* pointer).

During its lifetime an automaton delegates its methods to the current state class. While executing a delegated method the state object might generate an event and notify the context using event notification interface.

The next state is determined by the context on the basis of the current state and the event.

Results

- As in the *State* pattern, the state-dependent behavior is localized in the state classes.
- Unlike the *State* pattern in the proposed pattern transition logic is separated from the behavior in a particular state. The state classes should only notify a context of a particular event.
- Implementation of an automata interface is trivial and could be generated automatically.
- Transition could be implemented as a simple index lookup.
- *State Machine* provides pure (no unneeded methods) interface to a client. To prevent a client from using `IEventSink` we could use private inheritance (in *C++*) or define a private constructor and a static method that creates an instance of `Context`.

 ISBN 80-86943-11-9

- *State Machine*, unlike *State*, doesn't contain redundant interfaces for the context and the state classes — they all implement the same interface.
- It is possible to reuse state classes; moreover, state classes' hierarchies can be created. Note that it is mentioned in [Gam98] that new subclasses are easily added to the state classes. In fact, adding a subclass to a state class causes modification of all the rest of the state classes because the transition logic should be changed. Thus extension of a particular automaton implemented using *State* is being problematic.

Code Sample

The following sample in C# implements `Connection` class described in 2.2. It is a simplified model that allows transmitting and receiving data.

First let's describe interfaces and base classes that are used in this example. These classes are implemented in an assembly `ru.ifmo.is.sm`. Class diagram is shown on figure 3.

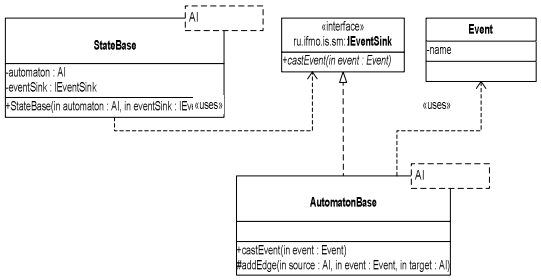

Figure 3. Class diagram for assembly ru.ifmo.is.sm

Let us describe all classes and events from this package:

- `IEventSink` — event notification interface:

```
public interface IEventSink {
   void castEvent(Event ev);
}
```

- `Event` — event class:

```
public sealed class Event {
   private readonly String name;

   public Event(String name) {
      if (name == null) throw new
            NullReferenceException();
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

- `StateBase` — base class for all state classes.

```
public abstract class StateBase<AI> {
   protected readonly AI automaton;
   protected readonly IEventSink
           eventSink;

   public StateBase(AI automaton,
           IEventSink eventSink) {
      if (automaton == null || eventSink
           == null) {
         throw new
           NullReferenceException();
      }
      this.automaton = automaton;
      this.eventSink = eventSink;
   }

   protected void castEvent(Event ev) {
      eventSink.castEvent(ev);
   }
}
```

- `AutomatonBase` — base class for all automata. It provides a method `addEdge` for its subclasses. In addition `AutomatonBase` implements `IEventSink`:

```
public abstract class AutomatonBase<AI>
           : IEventSink {
   protected AI state;
   private Dictionary<AI,
           Dictionary<Event, AI>> edges
           =
         new Dictionary<AI,
           Dictionary<Event, AI>>();

   protected void addEdge(AI source,
           Event ev, AI target) {
      Dictionary<Event, AI> row =
           edges[source];
      if (null == row) {
         row = new Dictionary<Event,
           AI>();
         edges.Add(source, row);
      }
      row.Add(ev, target);
   }

   public void castEvent(Event ev) {
      state = edges[state][ev];
   }
}
```

Classes created according to the *State Machine* pattern form an assembly `Connection`. Class diagram is shown on a figure 5.

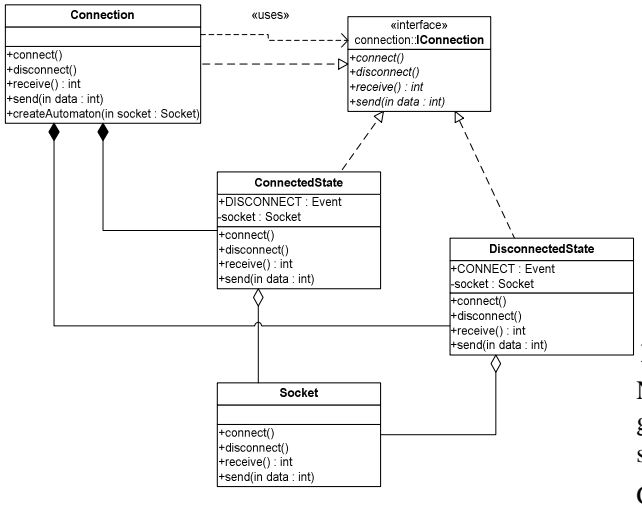

Figure 4. Class diagram for assembly connection

We use class `Socket` as a data model. It implements `IConnection` interface in this example. Control states of the automaton are `ConnectedState` and `DisconnectedState`. In `ConnectedState` we can expect `ERROR` and `DISCONECT` events and in `DisconnectedState` we can expect `CONNECT` and `ERROR` (figure 1).

The code of the state classes follows.

```
public class ConnectedState <AI>
   : StateBase<AI>, IConnection
        where AI : IConnection
{
    public static readonly Event
           DISCONNECT = new
           Event("DISCONNECT");
   public static readonly Event ERROR =
           new Event("ERROR");

   protected readonly Socket socket;

   public ConnectedState(AI automaton,
           IEventSink eventSink, Socket
           socket)
      : base(automaton, eventSink)
   {
      this.socket = socket;
   }

    public void connect() {
    }

    public void disconnect() {
       try {
          socket.disconnect();
       } finally {

           eventSink.castEvent(DISCONNEC
           T);
       }
    }

    public int receive() {
       try {
          return socket.receive();
       } catch (IOException e) {
          eventSink.castEvent(ERROR);
          throw e;
       }
    }

    public void send(int value) {
       try {
          socket.send(value);
       } catch (IOException e) {
          eventSink.castEvent(ERROR);
          throw e;
       }
    }
}
```

Note that state classes only partially specialize generic parameter of `StateBase`. It is used to support inheritance.

Class `DisconnectedState`:

```
public class DisconnectedState <AI>
   : StateBase<AI>, IConnection
     where AI : IConnection {
    public static readonly Event CONNECT
           = new Event("CONNECT");

    public static readonly Event ERROR =
           new Event("ERROR");

    protected readonly Socket socket;

    public DisconnectedState(AI
           automaton, IEventSink
           eventSink, Socket socket)
       : base(automaton, eventSink)
    {
       this.socket = socket;
    }

    public void connect() {
       try {
          socket.connect();
       } catch (IOException e) {
          eventSink.castEvent(ERROR);
          throw e;
       }
       eventSink.castEvent(CONNECT);
    }

    public void disconnect() {
    }

    public int receive() {
       throw new IOException("Connection
           is closed (receive)");
    }

    public void send(int value) {
       throw new IOException("Connection
           is closed (send)");
    }
}
```

Note that state classes define only event generation logic — transition logic is defined in the context.

3. Pattern extensibility

An extension of `Connection` will demonstrate how we can extend automata interface. Let's extend automata interface in the following way.

```
public interface IPushBackConnection :
IConnection {
      void pushBack(int value);
}
```

When calling `pushBack` the value passed as an argument is pushed on top of the stack to be popped in the next call of `receive`. If the stack is empty at the moment when `receive` is called, then the value is being pulled from the socket as in the previous example.

In this case the number of control states doesn't change but the state classes and the automaton must implement an extended interface. Let's call a context of the new automaton `PushBackConnection` and the new state classes `PushBackConnectedState` and `PushBackDisconnectedState`. Here is an implementation of `PushBackConnectedState`. Note that this class extends `ConnectedState` inheriting its logic.

```
public class PushBackConnectedState <AI>
          : ConnectedState<AI>,
          IPushBackConnection  where AI
          : IPushBackConnection
{
   Stack<int> stack = new
          Stack<Integer>();

   public PushBackConnectedState(AI
          automaton, IEventSink
          eventSink, Socket socket)
        : base(automaton, eventSink,
          socket) {
   }

   public int receive() {
      if (stack.empty()) {
         return base.receive();
      }

      return stack.pop();
   }

   public void pushBack(int value) {
      stack.push(new Integer(value));
   }
}
```

`PushBackDisconnectedState` class is implemented in the same way. So we'll only show the `PushBackConnection` code.

```
public class PushBackConnection :
          AutomatonBase<IPushBackConnec
          tion>, IPushBackConnection {
   private PushBackConnection() {
      Socket socket = new Socket();

      IPushBackConnection connected =
            new
            PushBackConnectedState<PushBa
            ckConnection>(this, this,
            socket);
      IPushBackConnection disconnected =
            new
            PushBackDisconnectedState<Pus
            hBackConnection>(this, this,
            socket);

      addEdge(connected,
            PushBackConnectedState<IPushB
            ackConnection>.DISCONNECT,
            disconnected);
      addEdge(connected,
            PushBackConnectedState<IPushB
            ackConnection>.ERROR,
            disconnected);
      addEdge(disconnected,
            PushBackDisconnectedState<IPu
            shBackConnection>.CONNECT,
            connected);

      state = disconnected;
   }

   public static IPushBackConnection
            createAutomaton() {
      return new PushBackConnection();
   }

   public void connect(){
            state.connect(); }
   public void disconnect() {
            state.disconnect(); }
   public int receive() { return
            state.receive(); }
   public void send(int value) {
            state.send(value); }
   public void pushBack(int value) {
            state.pushBack(value); }
}
```

A class diagram for `PushBackConnection` is shown on figure 5.

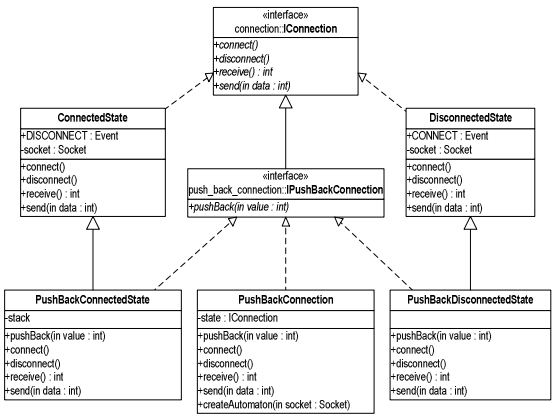

Figure 5. Class diagram interface extensibility example

In a similar way we can reuse state classes when creating a new automaton.

4. Conclusion

State Machine pattern improves *State* and inherits its main idea — to encapsulate the state-dependent behavior in a separate class.

The new pattern improves *State* in the following aspects.

- When using *State Machine* it is possible to design state classes independently. Thus the same state class could be used in several automata. This eliminates the major disadvantage of *State* — reuse issues.
- In *State* transition logic is distributed throughout state classes which introduces coupling between them. *State Machine* addresses this issue. It separates transition logic and the behavior in a particular state.
- As opposed to *State*, *State Machine* doesn't cause interface redundancy.

In *State Machine* you still need to implement trivial delegation of the automata interface methods to the current state. Such a delegation could be done automatically with the aid of *CASE* tools. Another option is to modify a programming language to support automata in a natural way. The authors are working on such language.

5. REFERENCES

[Aho85] Aho A., Sethi R., Ullman J. Compilers: Principles, Techniques and Tools. MA: Addison-Wesley, 1985, 500 p.

[Ada03] Adamczyk P. The Anthology of the Finite State Machine Design Patterns. http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Adamczyk-State-Machine.pdf

[JDO01] Java Data Objects (JDO). http://java.sun.com/products/jdo/index.jsp.

[Kle56] **Kleene S. C.** Representation of Events in Nerve Nets and Finite Automata, 1956 //Issue [6]. — P. 3–41

[Gamma98] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns. MA: Addison-Wesley Professional. 2001. — 395

[Gra02] Grand M. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML. Wiley, 2002. — 544 p.

[Odr96] Odrowski J., Sogaard P. Pattern Integration — Variations of State // Proceedings of PLoP96. http://www.cs.wustl.edu/~schmidt/PLoP-96/odrowski.ps.gz [San96] Sandén B. The state-machine pattern // Proceedings of the conference on TRI-Ada '96 http://java.sun.com/products/jdo/index.jsp.

[San95] Sane A., Campbell R.. Object-Oriented State Machines: Subclassing, Composition, Genericity // OOPSLA '95. http://choices.cs.uiuc.edu/sane/home.html.

[Ster01] Steling S., Maassen O. Applied Java Patterns. Pearson Higher Education. 2001, P. 608

Building .NET GUIs for Haskell applications

Beatriz Alarcón
DSIC, UPV, Camino de Vera s/n,
46022 Valencia, Spain
balarcon@dsic.upv.es

Salvador Lucas
DSIC, UPV, Camino de Vera
s/n, 46022 Valencia, Spain
slucas@dsic.upv.es

ABSTRACT

.NET is an emerging Microsoft's project which promotes a new framework for Software Development emphasizing the use of Internet resources and the interaction between components written in different programming languages. Whereas functional programming languages such as Haskell are well-suited for developing tools to analyze, verify and transform programs, typical Haskell compilers do not provide sophisticated capabilities such as support for XML-Web services, assisted GUI development, HTML processing, etc., which are frequent in most .NET development frameworks. We show how to integrate software components developed in a functional language as Haskell together with (graphic) components developed in C# or another .NET language. To achieve our objective we use the facilities offered by .NET to import COM components, on the one hand, and the technology developed to generate COM components from Haskell modules, on the other.

Keywords: COM, Haskell, Interoperability, .NET, Programming environments.

1 INTRODUCTION

International efforts to develop a global framework to use software resources have in Java and .NET their most well-known exponents. .NET is an emerging Microsoft's project which promotes a new framework for Software Development emphasizing the use of Internet resources and the interaction between components written in different programming languages [Cha02]. Within the .NET platform we can integrate already existing technologies and products as well as new elements. The XML project promoted by the WWW consortium[1] is also related to this effort through the use of XML to document programs in .NET, the support of Web services based on XML, etc.

The scientific communities that develop languages and declarative software technology are carrying out an important effort to make use of this kind of initiatives. Functional languages like Haskell[2] offer many programming features and resources which make them powerful tools for developing software projects and rapid prototypes. However, typical Haskell compilers (e.g. GHC, Hugs,...) do not provide visual tools for easily defining graphical user interfaces (GUIs), as, on the contrary, many other programming languages have. Although there are several libraries and systems which can be used to develop GUIs in Haskell (e.g., *wxHaskell*[3], *Gtk2Hs*[4], *HToolkit*[5], etc.), a Haskell programmer can waste too much time in giving form to his application if he make use of such tools due to the lack of a graphic assistant which makes easier the design of a GUI. With an Integrated Development Environment (IDE) like Visual Studio .NET, this is pretty simple. The support to define Web services offered by the .NET platform is a second aspect of Haskell applications for which we could argue similarly.

Of course, having graphic libraries for functional languages is very interesting and useful. Unfortunately, we can not affirm that such libraries (e.g., *wxHaskell*, which we have used to develop a large Haskell application like the termination tool MU-TERM [Luc04]) behaves like a completely stable and handy system (yet) since you have to make sure that you have the same version of the GHC compiler installed that requires the version of wxHaskell you want to use. The design, description, and use of forms and graphic controls is not very easy and it can take time to obtain what one is looking for. Moreover, it is necessary to get a grip on three basic concepts: widgets, layout and events.

This gave us a first motivation to start the research in this paper. Another (more general) motivation comes from the frequent need (in software development) of combining software pieces of code written in different programming languages. Of course, this is the well-known problem of *interoperability* of software components in sofware engineering and there are a number of *middleware* solutions available for dealing with this (also for Haskell applications, as we will see below). However, as far as we know, no attempt to *use* the .NET technology in practice (i.e., with a real Haskell application) has been reported yet. We have also tackled this task: In 1999, Finne et al. [FLMP99] explored the possibility of encapsulating Haskell programs like COM objects (Microsoft's *Component Object Model* [Rog97, COM04]). Why couldn't we take a step further and achieve our goal by means of COM and .NET interoperability? Microsoft has left opened the possibility of using already existing COM components in .NET; thus, a Windows programmer does not need to rewrite

[1] http://www.w3c.org
[2] http://www.haskell.org
[3] http://wxhaskell.sourceforge.net
[4] http://haskell.org/gtk2hs/
[5] http://htoolkit.sourceforge.net/

all his applications to run them under .NET. In our case, we show how to take advantage of this to pack Haskell programs as software components and integrate them into applications written in other languages, for example in C#, the most popular .NET language. Let's give a brief overview of our approach.

Our starting point is *HaskellDirect* (*HDirect* [Fin99, FLMP99, HDi99]) a framework for Haskell FFI (*Foreign Function Interface*) based on the standard IDL (*Interface Definition Language*) which allows to specify a programming interface in a programming language independent manner. There are many possibilities that HDirect offer to the programmer: Creating Haskell bindings to external libraries, creating external bindings to Haskell libraries, creating Haskell client interfaces to COM objects, and creating Haskell COM objects. In our case, starting from a Haskell component, we build a COM component which is encapsulated into a *Dynamic Link Library* (DLL), making it able to interoperate with Windows applications and, in particular with .NET applications. Our particular interest is furnishing Haskell applications with .NET GUIs, but most of the discussion is completely general and independent from this concrete goal. HDirect implements in Haskell all the required functionality to build a COM component and exempts the programmer from the knowledge of the COM specification since it is generated automatically. Next, we make use of the .NET facility to import COM components which can be used as external functions to implement the C# event handlers for the controls in the .NET GUI.

The paper is organised as follows: Section 2 briefly describes .NET graphic controls. Section 3 introduces a simple case study which we use to illustrate our development. Section 4 explains how to build a COM component from a Haskell module. Section 5 addresses the problem of its integration into .NET. Section 6 reports on the results obtained on a concrete (realistic) application of our technique. Section 7 displays our conclusions and lines of future research.

2 OVERVIEW OF .NET GRAPHIC CONTROLS

When a Windows programmer writes a .NET application (in, e.g., C#), he or she can take advantage of the System.Windows.Forms namespace, which provides a variety of control classes for developing rich user interfaces. Some controls are designed for data entry in the application (e.g., TextBox and ComboBox controls). Other controls display application data (e.g., Label and ListView controls). The namespace also provides controls for invoking commands within the application, such as the Button and MainMenu controls. In this paper we are specially interested in showing how Haskell applications can take advantage from .NET technology, specially from .NET GUIs. Thus, we only consider the information (or *data*) that graphic controls and Haskell components should (usually) *exchange*. Although other control properties (e.g., control labels, colors, etc.) could also be managed through Haskell components, we will not consider them in detail here; we center the attention on the non-graphic part of this information exchange . Extending the treatment of controls to achieve such more generality would be managed in a similar way, if necessary.

The hierarchy of .NET controls is very large. Here, according to [FPB$^+$03] we mention the most common controls (which are also the most frequently used, in our personal practice). We consider that these controls suffice for giving a complete account of the problems and solutions that any other control could rise and require to achieve our purpose.

The table in Figure 1 shows the Haskell-like data as could be considered to be managed by each .NET control. This table shows that with few simple Haskell datatypes can be managed all necessary information, regarding our main purpose of having the graphic part of the application developed in .NET (C#) and the 'logic' of the program written in Haskell.

3 A SIMPLE CASE STUDY

In order to discuss the techniques developed here, we use a simple case study. It includes a simple graphic interface to introduce and manipulate strings by means of simple transformations:

- converting the characters of the string into capital or small letters,
- removing spare blank spaces, and
- simple encryption (based on the well-known Caesar's method)

The length of each string is also stored (as an integer value). In order to highlight the role of Haskell as the language which actually implements the logic of the application, the use of C# here is strictly limited to provide a GUI, i.e., to ease the introduction and visualization of strings by means of graphic controls. The length of the *current* string is displayed in a read only text control. The different transformations are triggered by means of buttons. The current string is selected from a *ComboBox* which shows the strings introduced so far (see Figure 2).

In the Haskell part, we have the structures of functional data which are necessary to control the state of the system: we store each pair string-length in a list that is indexed by a integer that points out at the *current* position of the list (`Focus`):

WINDOWS FORMS	ASSOCIATED DATA
Button, GroupBox, Panel, Label, Splitter	-
CheckBox, RadioButton	Bool
ListBox	([Int],[String])
ComboBox	(Int,[String])
ListView	[[String]]
TrackBar, ProgressBar, NumericUpDown	Int
TextBox, RichTextBox	String
MainMenu, OpenFileDialog, SaveFileDialog, FolderBrowserDialog	-

Figure 1: .NET controls and data

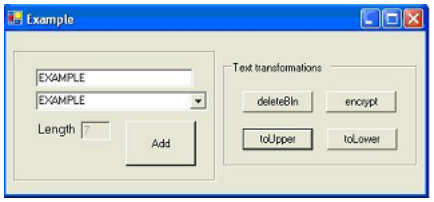

Figure 2: Simple example of interoperability

```
type Focus = Int
type Length = Int
data HL = H_L [(String, Length)]
          Focus deriving Show
```

The algebraic data type `HL` contains all necessary information to implement the required functionality explained above. The following mappings manipulate this data structure:

```
- Adds a new string and its length
addPair :: HL -> String -> HL
- Obtains the 'current' string
getString :: HL -> String
- Updates the 'current' string
writeString :: HL -> String -> HL
- Length of the 'current' string
getLength :: HL -> Int
- Sets the (index of) 'current' string
setFocus :: HL -> Int -> HL
```

The following mappings implement the transformations over strings.

```
toUpperCase :: String -> String
toLowerCase :: String -> String
deleteB :: String -> String
encrypt :: String -> String
```

Haskell files and other (IDL, C#, etc.) archives as explained below can be retrieved from `http://www.dsic.upv.es/~balarcon/example.zip`.

4 INTEROPERABILITY BY MEANS OF COM IN HASKELL

Microsoft's COM technology is used to create re-usable components (possibly written in different programming languages) and connect them together. In the following, we show how to use COM technology to connect Haskell with .NET components.

4.1 Haskell modules and COM components

A Haskell program that implements a COM component consists of four parts:

- The application code, written in Haskell by the programmer.
- An IDL specification establishing those Haskell functions which we want to make accessible through the DLL.
- A set of Haskell modules which are automatically generated from the IDL by the HDirect tool.
- A Haskell library module, Com, that exports all the functions needed to support COM objects in Haskell and a C library module that provides some run-Time support (RTS)

In the following sections we briefly describe and discuss these steps of the process.

4.2 The IDL of the Haskell component

IDL is a declarative language which is used to describe interfaces and classes disregarding any programming language [Hlu98]. An IDL specification describes the interface of a component.

The IDL code in Figure 3 is used in our case study. We have followed the example in [FLMP99], the indications of the manual of HDirect [Fin99] and the information about IDL [Hlu98]. We declare all (and only!) Haskell functions that we wish to have accessible from C# code together with their arguments and the type of the returned value.

Now we are going to describe the IDL code. This is useful to understand what we are going to obtain from COM [Rog97, Ste04, COM04]. On the basis of the IDL code, we are going to build the skeleton of the object that we want to encapsulate. For that purpose, we have a `library` (Example), an `interface` (Iexample) and a `class` (EXAMPLE).

 ISBN 80-86943-11-9

```
[ uuid(35E80A56-3664-4d91-9C6C-3018496A8D61) ,
helpstring("Haskell COM component") ,
version(1.0) ]

library Example {

importlib("stdole32.tlb");

[ object,
 uuid(4DB0C045-CC9F-4607-B79A-26D27E0C1594)]

interface Iexample : IUnknown {

 HRESULT addPair([in,string]BSTR in);
 HRESULT getString([out,retval] BSTR *out);

 HRESULT getLength([out,retval] int *out);
 HRESULT setFocus([in] int in);
 HRESULT toUpperCase();
 HRESULT toLowerCase();
 HRESULT deleteB();
 HRESULT encrypt();

 };

[ object,
 uuid(49D98D24-DC88-4d24-8C5D-404FE510644D)]
coclass EXAMPLE {
  [default]interface Iexample;

 };

};
```

Figure 3: IDL code for the case study

A *type library* is a binary file that contains the same information that we could find in a C or C++ header file. It includes the names of the classes and the interfaces which are implemented in the server and the number and type of parameters for each method of their interfaces. Note that it also contains the GUID (Globally Unique IDentifiers), a very important part of the model of COM programming, for each class and interface. A GUID is a structure of 128 bits "statistically guaranteed" to be unique. In our case we have used the tool Create GUID (which is part of Visual Studio .NET) to generate them.

A *COM interface* is a collection of linked methods that perform a functionality. All are based on the *IUnknown* interface; each of them receives a unique *interface identifier* (IID).

A *COM class* is the implementation of one or more COM interfaces, while a COM object is an instance of a class. Each object has a class identifier (CLSID). CLSID and IID are subgroup of GUID.

The name of our interface is Iexample and inherits from IUnknown the use of methods QueryInterface, AddRef and Release. Inheritance from multiple interfaces is not allowed. The first attribute, `object`, which is locked up in brackets next to the GUID, identifies the interface as a COM interface. For each method in the interface, we specify the parameters with which the method will be called (from C#). The attribute `in` indicates that the parameter is used as an input given to the method (e.g., in addPair), `out` indicates output (e.g., in `getString`). The attribute `string` is used with parameters that are pointers to characters. The `retval` keyword indicates that the parameter must be interpreted as the returned value of the function. It must do it in this way, because the literal return of the method is a HRESULT type, which is used to give back the information of errors.

4.3 Encapsulating a Haskell component as a COM component

Once the IDL has been specified, the next step is to generate the *proxy* and the *skeleton* of our component. In order to generate those modules we use the following (HDirect) command:

```
ihc -fcom example.idl -s -skeleton
```

This generates two Haskell files: EXAMPLE.hs and ExampleProxy.hs. The first one contains the *skeleton* of the methods that implement our component, that is, the Haskell structure for the methods declared in the IDL. The second one provides a *proxy* that adapts our methods behind an interface COM to make the communication possible.

Regarding the definition of the *skeleton*, HDirect accomplishes three fundamental tasks:

- To import the necessary Haskell modules to give support to the characteristics of the interface specified by the IDL.
- To introduce a `State` type to implement the (necessary) persistence of the functional data by means of a mutable variable that can be initialized, read and modified by means of the functions of the `IOExts` library of *GHC*[6].
- To include Haskell declarations corresponding to the functions defined in the IDL. Haskell functions will have an additional argument corresponding to the state of the application (that will be able to be read or modified) and a monadic return type `IO t` where `t` is the (non monadic) type returned as indicated in the IDL (`String` or `Integer`, in our case).

The following step is to fill up the *skeleton* with the Haskell code of our methods. In our case, the `HList` module contains the methods in pure functional code, so we will fill up the *skeleton* with the corresponding calls to methods and the operations to read and write, the state by means of `readIORef` and `writeIORef` (defined in `IOExts`). For instance, for `deleteB`, we have:

[6] Glasgow Haskell Compiler, `http://www.haskell.org/ghc`.

```
module EXAMPLE where (...)

import IOExts

-Pure Haskell Component
import qualified Hlist

data State = State(IORef HList.HL)
    ⋮
deleteB :: State
        -> Prelude.IO ()
deleteB (State st) = do
        hl <- readIORef st
       ;str' <- Prelude.return
                (HList.deleteB (HList.getString hl))
       ; writeIORef st(HList.writeString str' hl)
```

4.4 Creating a COM DLL from Haskell modules

The next step is to compile the two new files to generate the .hi and .o files and the *stubs* of the *proxy*:

```
ghc -c EXAMPLE.hs  ExampleProxy.hs
```

Now we have to decide how to encapsulate our component. HDirect provides solutions to build servers of internal processes (DLLs) or servers of external processes (EXEs). We have chosen to implement a DLL. Although it entails a bit more effort, the user benefits from a simpler use of the COM model, as the COM object is loaded without any intervention from the user.

In order to implement a DLL, the next step is to include the ComDllMain.lhs and dll_stub.c modules in the directory and compile them. Finally it is necessary to provide a Main module (required by GHC for descriptive purposes).

Once the module Main has been compiled, we build the type library (.tlb) using HDirect from the IDL and the proxy, generating *example.tlb*:

```
  ihc -s -fanon-typelib -v -c example.idl -o
ExampleProxy.hs -output-tlb=example.tlb
```

The type library is a resource that we must bind to our DLL. Resources are specified using a special and very simple text file, called resource script or .rc file. The file contains the specification of the resources that we want to include in the program or DLL (in our case the type library) for compiling it with the resource compiler. The resource compiler converts the file .rc into an object file (.o). The resource compiler is a GNU binary utility called *windres*. We use it along with *cygwin* to include *example.tlb* in our project. Now, we can build the DLL.

5 INTEGRATION OF COM INTO .NET

At this point, we must insert the COM DLL into our Visual Studio.NET project[7]. Having the DLL, it is necessary to register the generated component. The simplest way is using *regsvr32.exe*, in the command-prompt window. COM only uses a registry branch: HKEY_CLASSES_ROOT. Under it, we can find all the CLSIDs of the components installed in the system. A CLSID is contained in the registry as an alphanumeric string with the following format: {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}.

5.1 Using COM components in .NET applications

A .NET client cannot directly communicate with a COM component because the interfaces exposed by the COM component cannot be read from the .NET application. The data types, the mechanisms for managing errors, etc., are different for *managed* and *unmanaged* objects[8]. In order to simplify the interoperation between the components of .NET Framework and the unmanaged code, the CLR (Common Language Runtime) hides the differences between them both to clients and servers. This is achieved by means of a RCW (Runtime Callable Wrapper)(to understand the whole process see Figure 4) . The .NET SDK provides RCW to obtain it, thus a .NET application can see the unmanaged component as if it was managed. In .NET there are several ways to do this:

- Using the Type Library Importer utility (*Tlbimp.exe*), provided together with the .NET Framework.
- Making reference to the COM component directly from the C# application.

Tlbimp is a console application that converts the type definitions found in a COM type library into equivalent definitions in a .NET assembly. The assembly produced by the Tlbimp.exe tool is a standard .NET assembly that can be examined with *Ildasm.exe* (MSIL disassembler).

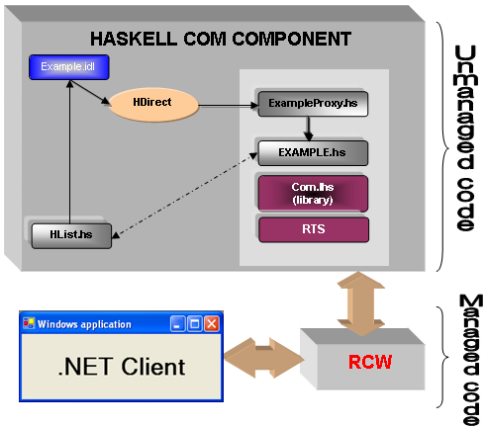

Figure 4: Interoperability with .NET from Haskell

After registering our DLL, we use *Tlbimp* and we run VS.NET. From our Windows application we click the

[7] We use Visual Studio.NET 2003.

[8] The .NET native CLR code is called 'managed', in contrast to any other machine-dependent code which is 'unmanaged' [Cha02].

right button on the *References* file in the VS *Solution Explorer*, we select *Add reference* and look for the assembly which we have just generated. Now it can be used exactly as any other .NET assembly: we just create an instance (denoted by `h`) of the appropriate class:

```
ExampleClass h = new ExampleClass();
```

Now we can access to Haskell functions as if they were C# functions (see Figure 5).

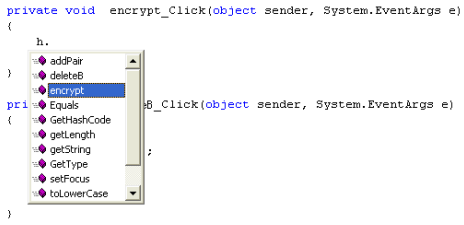

Figure 5: Haskell functions in C#

We can use them to program the event handlers on the GUI that we have developed.

6 A .NET VERSION OF MU-TERM

MU-TERM is a termination proof tool for (Context-Sensitive) Rewriting Systems. (Context-sensitive) Rewrite Systems are useful for describing semantic aspects of a number of programming languages (e.g., Maude, OBJ2, OBJ3, or CafeOBJ) and analyzing the computational properties of the corresponding programs, in particular termination (see [DLM+04, Luc01, Luc02]). The tool implements the generation of the appropriate orderings and transformations for proving termination. MU-TERM is written in Haskell and *wxHaskell* was used to develop the graphical user interface. The system consists of around 30 Haskell modules containing more than 5000 lines of code. We refer the reader to [Luc04] for more information about the use and functionality of the tool. Compiled versions and instructions for the installation are available on the MU-TERM WWW site.

We have developed a new (hybrid) version of MU-TERM which, having the same functionalities (implemented by the same Haskell modules), includes a GUI written in C# which replaces the old one. Let's take a look to the windows which constitute the GUI of MU-TERM (see Figures 6 and 7) and let's consider the corresponding controls. As it can be noticed, the controls to manage in the interface are *MenuFile*, *Button*, *ComboBox*, *CheckBox*, *TextBox*, *ListBox*, etc. In Section 2 we discussed them and their associated data. We have applied the process described in Sections 4 to 5 to MU-TERM and the obtained results were very satisfactory. The new version of MU-TERM is now composed of the same number of Haskell modules but the WinMuTerm.hs module, which contained about 1200 lines of code, has been replaced by a new module WinMuTermNET.hs, that contains less than 800 lines.

On the other hand, the C# part of the .NET version of MU-TERM (consisting of six new modules with about 2000 lines altogether, most of them generated automatically(!) by the graphic assistant) includes a new C# module WinMuTermNET.cs that implements the creation of the new user interface and manages the events transforming them in function calls to Haskell code by means of exchanges of strings and integers. This C# component uses the COM DLL generated from WinMuTermNET.hs (together with the other Haskell modules). The .NET version of MU-TERM is available on the MU-TERM WWW site.

7 CONCLUSIONS AND FUTURE WORK

We have shown how to integrate software components developed in Haskell together with (graphic) components developed in C#, or other .NET language. Our starting point is *HDirect* which permits to build a COM component from a Haskell module, and making it available as a COM DLL which can interoperate with .NET applications. We have shown the practicality of this approach by giving a new .NET GUI to a Haskell tool like MU-TERM. Other remarkable aspects are:

- it is a complete experience of 'weak' integration of software components written in a functional language like Haskell in a software development platform like .NET that still does not manage the inclusion of sources written directly in this language.
- it is a pioneer experience in the functional programming community, since MU-TERM is the first complex software written in Haskell that uses COM technology by means of HDirect.
- it is also a pioneer experience for the academic community interested in the interoperability of program analysis software tools, specially regarding tools for proving termination, where interoperability of different tools can be important [9].

In the world of functional languages, there are more or less complete approximations to .NET for the languages[10]. Regarding Haskell, a full-featured Haskell development environment has been recently implemented. It is called *Visual Haskell* [AM05]. Although it is an interesting contribution for the Haskell community, it does not treat the possibility of building graphic user interfaces for Haskell programs using the .NET resources. In their project they have also used HDirect, although they did not find it completely

[9] See, for instance, `http://www.lri.fr/~marche/termination-competition`.

[10] ML `http://www.cl.cam.ac.uk/Research/TSG/SMLNET` or Mondrian `http://www.mondrian-script.org`.

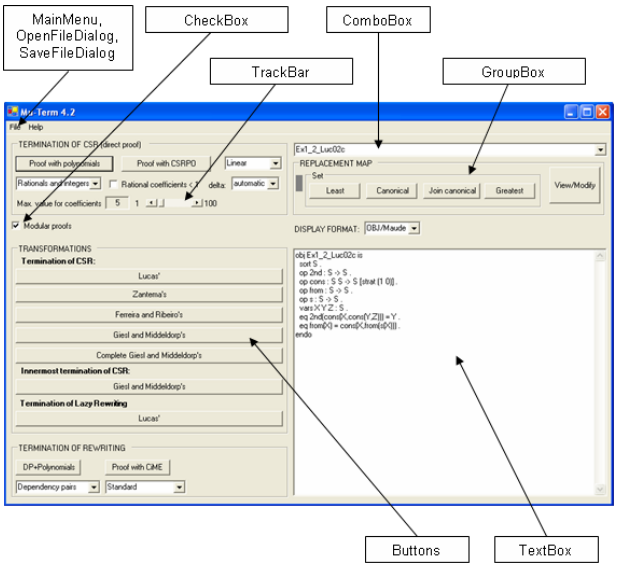

Figure 6: Principal window of MU-TERM

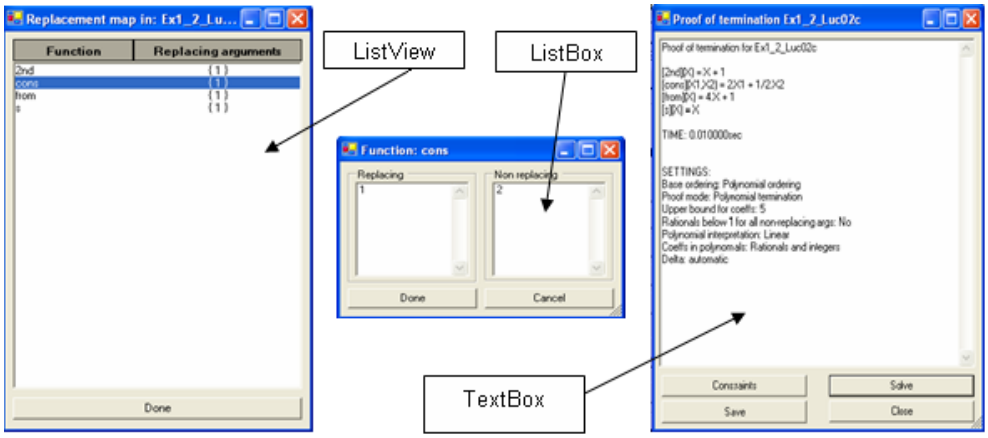

Figure 7: Rest of MU-TERM windows

appropriate for their purposes. This is also in contrast to ours: due to the simplicity of the information exchange between the C#-based user GUI and the core Haskell application, we find HDirect to be easy to use (although it took time to reach a sufficient *know-how*). For instance, HDirect limits the structures of Haskell data that are directly interchangeable by means of COM to strings and integers (of 32 bits). This can be a problem for most applications, but it is not problematic for developing GUIs, since the involved data types (see Section 2) are easily exchangeable in such format.

These initiatives to integrate functional languages into the .NET framework reveal the interest of the community to converge to this platform. Our experience is also encouraging. We plan to develop the theoretical aspects of our work, and also envisage possible extensions of this experience to other tools and programming languages in the future. In particular, we want to explore the use of the .NET facilities for using Web Services based on XML with these tools and programming languages. A first candidate, again, could be the termination tool MU-TERM.

ACKNOWLEDGEMENTS

Work partially supported by Spanish MEC grant SELF TIN 2004-07943-C04-02, Acción Integrada HU 2003-0003, and EU-India Cross-Cultural Dissemination project ALA/95/23/2003/077-054.

REFERENCES

[AM05] K. Angelov and S. Marlow. Visual Haskell. In *Proc. of Haskell Workshop, Haskell'05*, pages 5-16, ACM Press, 2005.

[Arc01] T. Archer. Inside C#. McGraw-Hill, 2001.

[Cha02] D. Chappell. Understanding .NET. Addison Wesley, 2002.

[COM04] COM, Component Object Model. `http://www.etse.urv.es/EngInf/assig/ens4/2004/net4a.pdf`

[DLM+04] F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving Termination of Membership Equational Programs. In P. Sestoft and N. Heintze, editors,

Proc. of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, PEPM'04, pages 147-158, ACM Press, New York, 2004.

[Fin99] S. Finne. HaskellDirect UserÂ«s Manual. November, 1999.

[FLMP99] S. Finne, D. Leijen, E.Meijer, S. Peyton Jones. Calling hell from heaven and heaven from hell. In *Proc. of 4th ACM SIGPLAN International Conference on Functional Programming, ICFP'99*, Sigplan Notices 34(9):114-125, 1999.

[FPB+03] J. Ferguson, B. Patterson, J. Beres, P. Boutquin, and M. Gupta. C#'s bible. Microsoft Press, 2003.

[HDi99] H/Direct: supporting component programming in Haskell. `http://www.haskell.org/hdirect/design.html#toc3`

[Hlu98] B. Hludzinski. Understanding Interface Definition Language: A Developers Survival Guide, 1998. `http://www.microsoft.com/msj/0898/idl/idl.htm`

[Hoa03] T. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM*, 50(1):63-69, 2003.

[Luc01] S. Lucas. Termination of Rewriting With Strategy Annotations. In *Proc. of LPAR'01*, LNAI 2250:669-684, Springer-Verlag, Berlin, 2001.

[Luc02] S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):293-343, 2002.

[Luc04] S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting In V. van Oostrom, editor, *Proc. of 15h International Conference on Rewriting Techniques and Applications, RTA'04*, LNCS 3091:200-209, Springer-Verlag, Berlin, 2004. Available at `http://www.dsic.upv.es/~slucas/csr/termination/muterm`.

[Rog97] D. Rogerson. Inside COM. Microsoft's Component Object Model. Microsoft Press, 1997.

[Ste04] P. Steele. 15 Seconds: COM Interop Exposed. 2004. `http://www.15seconds.com/issue/040721.htm`

[Tro02] A. Troelsen. COM and .NET Interoperability. Apress, 2002.

Self-contained CLI Assemblies

Bernhard Rabe
Haso-Plattner-Institute,
University of Potsdam
P.O. Box 90 04 60
14440 Potsdam, Germany
bernhard.rabe@hpi.uni-potsdam.de

ABSTRACT

High-level programming languages and bytecode-based virtual execution environments have become popular in software development. Bytecode-based runtimes extend embedded system by techniques to improve safety, help portability and interoperability. The ECMA/ISO Common Language Infrastructure (CLI) specifies a bytecode-based execution environment (Common Language Runtime) and a comprehensive class library. CLI applications suffer from long startup time, high memory consumption and the amount of referenced assemblies. Startup time is determined by resolving references and high memory consumption through big class library assemblies. Often CLI applications use a small subset of the CLI class library, but the whole memory footprint is basically determined by the class library. To overcome memory requirements of the class library, a minimal application format that includes all essential class library functionality is reasonable. Self-contained CLI assemblies as an approach for size-optimized deployment format are presented in this paper.

Keywords

CLI, assembly format, space-optimization.

1. INTRODUCTION

High-level programming languages and bytecode-based execution environment have become popular in development of desktop systems. The *Common Language Infrastructure* (CLI) [Int03a] as implemented in the .NET Framework [Mic05a] has been a popular platform for creating component-based applications, because of:

- Platform independence of bytecode-based executables
- Fine granular security restrictions
- Revisable code
- Component model

It would be beneficial if CLI applications could be executed on memory restricted systems that are not covered by existing CLI implementation. .NET developers could then reuse their code for these systems instead of reimplementing their applications from the ground up using C or C++.

Embedded systems differ from desktop systems in various aspects:

- Hardware resources are often limited: memory size, processing power, power supply.
- Software capabilities: Faulty programs can crash the system, because memory protection is not available.
- Capabilities for developer interaction, for debugging, or communication bandwidth are often limited.

CLI technology is integrated seamlessly in Rapid Application Development tools as Microsoft's Visual Studio suite for desktop development just as for embedded development. Compiler and tools are available for multiple programming languages e.g. C#, C++ .NET, or Delphi. The CLI could offer developers of embedded systems the same advantages as for desktop systems.

Due to the predictable nature of the sandbox-mode execution of CLI instructions, programming errors never result in system crashes, but cause exceptions

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

to be thrown. This allows for a simpler postmortem analysis of a fault. Due to the support for rapid prototyping, simulators for the target can be more easily created. Ideally, much of the code would only use standard library functions of the CLI, so that simulators are only necessary for the target-specific hardware.

The CLI as implemented in the Microsoft .NET Framework, the Microsoft Compact Framework [Mic05b], or the Mono Project [Mon06a] does not meet the requirements of limited resources of systems. There are few implementations of the CLI for small mobile devices e.g. for Symbian OS based [Gef05a], or for Linux based [Dot06a].

The memory footprint of an executable assembly is calculated by the assembly itself, the custom libraries used, the *Base Class Library* (BCL) and the *Common Language Runtime* (CLR). These are four items where size optimization can occur. In this paper the first three items were focused on. CLR optimization would harm the "compile once run everywhere" approach of CLI.

In this paper we present an approach to reduce the memory footprint of an executable assembly in that way the unused library functionality is not required to be present at runtime.

This can be achieved by compacting an assembly with its used library functionality into a self-contained assembly. The self-contained assembly will contain only required library functionality and will become smaller than the combined libraries. Furthermore the number of referenced assemblies which are required to be loaded is reduced to the self-contained assembly itself. The self-contained assembly is smaller than the sum of previously referenced assemblies.

This work is based on the PERWAPI [Gou05a] library, which is extended to the needs of creating self-contained assemblies.

The rest of this paper is structured as follows: Section 2 briefly reviews the Common Language Infrastructure. In Section 3 the mechanism of executing CIL-code is discussed in detail. Next, self-contained assemblies as approach for optimized memory footprints and predictable behavior in are presented in section 4. Section 5 gives an overview of related work followed by conclusions and future plans.

2. COMMON LANGUAGE INFRASTRUCTURE

The CLI standard specifies the executable format, a virtual runtime environment (*Virtual Execution System* (VES)) and a set of libraries as implemented in the Microsoft .NET Framework, Shared Source Common Language Infrastructure (SSCLI) [Mic02a], or in the Mono project.

CLI executables, called assemblies are encoded in the *Common Intermediate Language* (CIL) instruction set. An assembly is the deployment unit of the CLI and may consist of multiple files (modules). An assembly is loose coupled with the BCL and other assemblies in a way similar to native applications and shared libraries.

CIL is a stream of bytecodes similar to processor instructions. Most opcodes are one byte long, a few 2 bytes long and may have an optional parameter (up to 8 bytes long). Every method consists of a header, a body and a possible footer. To evaluate opcodes a stack is used. Bytecodes are located in the method body.

Metadata

Assemblies are equipped with metadata about references, type names, method names... Metadata are organized in a number of named streams. These streams are divided into 2 types: metadata heaps and metadata tables. For executing assemblies the following metadata tables are basically involved:

- *Assembly*: Assembly defined in the PE file.
- *AssemblyRef*: For execution required assemblies.
- *TypeRef*: Used types defined in external assemblies. Every type in this table refers its resolution scope that is located in the *AssemblyRef*-table for the relevant cases.
- *TypeDef*: Contains all types that are defined within an assembly.
- *Method*: All methods that are declared by types in *TypeDef*-table. Every row in the *Method*-table is owned by one and only one row in the *TypeDef*-table.
- *MemberRef*: All methods or fields of external defined types that are accessed within the assembly. There is merely a 'forward-pointer' from each row in the *TypeRef*-table.

References in metadata tables are tokens into table rows and heaps or relative virtual addresses within the assembly. Heaps are constant pools used for metadata and CIL code.

Costa and Rohou [Cos05a] show that metadata size varies from 40 percent up to 80 percent of the whole assembly size for representative set of programs. The metadata split 70 percent to 30 percent into constant pool (heaps) and tables. Section 3 will show that major parts of the *#String* are not required for executing

CIL code. For example textual descriptions of variables and properties are needed for reflection purposes only.

Version compatibility

To overcome the problem resulting from different versions of dynamic libraries on Windows systems [And00a] the CLI introduced a version management that builds up on version numbers and public keys. An assembly version number consists of four parts: major, minor, build and revision number. To make an assembly reference distinct the assembly must have a strong name. Strong names guarantee name uniqueness by relying on unique key pair. All shared assemblies that reside in the GAC must have a strong name. The BCL of actual CLI implementation have all the same standard public key that does not require a private key to sign. This is done to provide vendor independent execution of assemblies. That means an assembly which has references to the BCL (mscorlib.dll) may behave differently with different BCL implementations.

3. EXECUTION OF .NET ASSEMBLIES

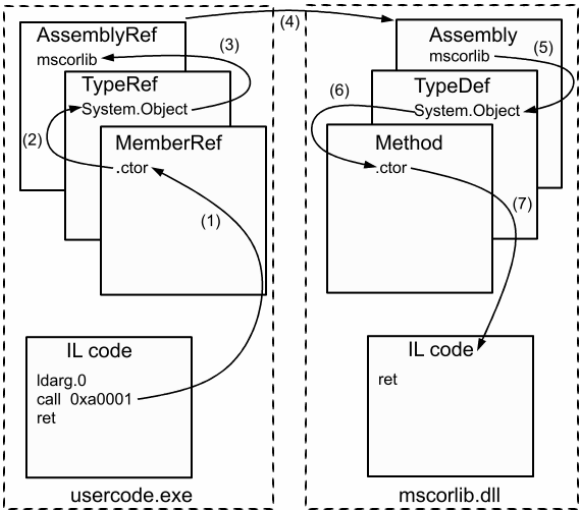

Figure 1: Resolving of an external method

When the CLR loads an assembly and starts executing a method all assemblies referenced within that method have to get loaded too. This means that all assemblies referenced in this assembly will be loaded, even though they might not be needed most of the time the application is executed.

A way to reduce the number of loaded modules is to merge multiple modules into one [Mic06a]. In terms of the CPU, assembly loads have fusion binding and CLR assembly-loading overhead in addition to the LoadLibrary call, so fewer modules mean less CPU time. In terms of memory usage, having fewer assemblies also means that the CLR will have less state to maintain.

To create the executable image the CLR has to locate referenced CIL code within an assembly. The complexity of this task is different for assembly internal and assembly external references. Figure 1 shows how CIL code of an external method will be located:

1. A CIL operation (call) has a token operand that points to a *MemberRef*-table row.
2. The *MemberRef*-table row contains the name of the method and a token into the *TypeRef*-table.
3. In the *TypeRef*-table row the namespace, the type name and a token into the *AssemblyRef*-table are included.
4. The *AssemblyRef*-table row provides the target assembly name and optional a version number as well as a public key token.
5. Within the referenced assembly the CLR looks into the *TypeDef*-table for the requested type. This is done by a linear search with string and signature comparison until the matching row is found.
6. The linear search for the matching method row in the *Method*-table is optimized in the way that the start of the relevant rows is known.
7. The matching *Method*-table row provides the address to CIL code within the PE-file.

This task must be repeated for every external method. In comparison with an external method call requires an internal method call a single lookup in the Method-table to get the address of the CIL code within the assembly. Recapitulating it has been reflected that loose coupling of assemblies and consequential external references cause the following drawbacks:

- *Memory consumption*: each external assembly must be loaded and metadata tables have to build up.
- *Processing power*: multiple indirections, linear search, string and signature compare during reference resolving cause additional CPU time in contrast with internal references.
- *Memory footprint*: combination of functionality into a single assembly (mscorlib.dll) causes a high CLR memory footprint if only a single type is referenced.
- *Revisable code*: CIL within an assembly can be inspected for validity. External assemblies especially the BCL may be implemented differently and makes it impossible to predict the behavior of CIL code.

These drawbacks can be minimized if all external referenced functionality is assembled to a single assembly. This harms the loose coupling, but it allows lower memory footprints and to analyze the assembly in terms of CIL code.

4. SELF-CONTAINED CLI ASSEMBLIES

A key feature of the CLI is the revisable bytecode-based execution of assemblies. The verification is done at runtime. But there are also needs for static revisable code before runtime e.g. prevent exceptions while runtime.

The loose coupling and dynamic linking of applications and libraries assemblies does not permit an static evaluation of CIL code, because CLRs may provide different implementations of relevant assemblies.

To overcome version conflicts of assemblies, CLI introduced strong names and side-by-side execution of different versions of the same assembly.

This works fine for most strong named assemblies, but fails for the BCL.

A static revisable assembly might not have dependencies to CLR-provided assemblies. With the self-contained assembly approach a static revisable format based on CIL code is proposed. This approach lifts up problems through different implementations of referenced assemblies.

Self-contained assembly features are:

- Minimal memory footprint
- Predictable behavior based on CIL-code
- Reduced startup time

The memory footprint of the runtime environment for an assembly is calculated by the CLR, the relevant libraries and the assembly itself. In general every assembly uses BCL features (e.g. `System.Object`). The BCL is represented as mscorlib.dll [Ecm02a]. But mscorlib.dll implementations of .NET Framework, Mono, Portable.NET [Dot06a] and Rotor provide different additional features, which are not used by most assemblies. Independently from the amount of mscorlib.dll features by an assembly the memory footprint for the BCL is fixed. Self-contained assemblies do not need additional library assemblies and form together with the CLR the minimal footprint for an execution environment. This feature targets mainly memory restricted systems.

Prediction of execution behavior of a self-contained assembly is possible, because all executable CIL codes are within the assembly. A static behavior evaluation can be done before runtime and allows for example prediction of memory consumption.

Dynamic linking of assemblies at load time causes delays until the first CIL code is executed. The time is needed for loading assemblies and resolving references. Self-contained assemblies does not require additional assemblies, therefore the startup time is shortened.

```
public class Hello{
  public static void Main(string[] args){
     Object obj=new Object();
     Console.WriteLine("Hello World!");
  }
}
```

Figure 2: Simple C# Hello world

Figure 2 shows a C# program cutout that has a Main-method where an instance of Object is created and "Hello World" is printed out. The second program in figure 3 shows the IL-code[1] of the Main-method generated by the Ildasm tool. The local variable `obj` disappeared, because it is not used furthermore. A instance of `System.Object` is created with a call of `.ctor()` from the mscorlib assembly. Then the string "Hello World" is printed out by an call of `System.Console::WriteLine` from the mscorlib too.

```
...
.method public hidebysig static void  Main(string[] args) cil managed
{
 .entrypoint
 .maxstack  1
 newobj     instance void [mscorlib]System.Object::.ctor()
 pop
 ldstr      "Hello World!"
 call       void [mscorlib]System.Console::WriteLine(string)
 ret
}
...
```

Figure 3: IL code of the compiled Main-method

The program in figure 4 is generated from the second program where the `System.Object` type was included. The `System.Object::.ctor()` call does not leave the assembly scope. The rest of the program behaves the same.

The two IL-programs differ also in the .maxstack value, because the Microsoft C# compiler generates a Fat-method header and the PERWAPI library a Tiny-

[1] The C# source code was compiled with .NET Framework v1.1 compiler and optimization (/optimize+) enabled.

method header. None of the requirements for a Fat-header are satisfied, so the 1 byte Tiny header is a better alternative for size optimization.

```
....
method public hidebysig static void  Main(string[] args) cil managed
{
 .entrypoint
 .maxstack  8
 newobj     instance void System.Object::.ctor()
 pop
 ldstr      "Hello World!"
 call       void [mscorlib]System.Console::WriteLine(string)
 ret
}
...
```

Figure 4: IL-code of Main-method with System.Object included

This demonstrates the adaptable level of containment for specific aims. The `System.Object` type was included and the reference to `System.Console ::WriteLine()` was kept.

Creating self-contained assemblies

Self-contained assemblies do not have any external references. This means a CLR should able to execute a self-contained assembly without loading the BCL or other managed assemblies.

In contrast to statically linked native binaries, the CLI abstracts from the operating system and the underlying hardware. This fact makes it feasible to build a CLR independent CLI assembly.

To get a self-contained assembly, the relevant assembly must be disengaged from type references to external assemblies. This work can be done by processing IL textual representation or by using an assembly manipulation library.

In this project the library approach is used, because ILDASM approach requires a lot of text substitution and depends on available CLI framework tools.

The Reflection API of the .NET Framework does not supports access to CIL code. Microsoft's new compiler framework Phoenix allows assembly modifications within a compiler run. After evaluation of capabilities of different assembly manipulation frameworks the work presented in this paper finally bases on PERWAPI [Gou05a] developed at the Queensland University of Technology. PERWAPI provides an abstract representation of the PE-file embodied as object oriented structure. The library is implemented in C# and is released as available for free. PERWAPI was extended to support the creation of self-contained assemblies.

Figure 5 shows the creation of self-contained assemblies with the Linker tool and an optional configuration. The assembly on the left side references the BCL (mscorlib) and may have references to multiple custom libraries.

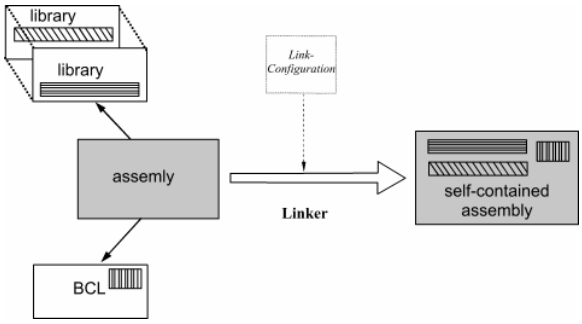

Figure 5: Creation of self-contained assemblies

The PERWAPI-based linker tool resolves references controlled by an optional configuration file. The configuration allows the instrumentation of the assembling process inside the linker tool. The source for a type to import could be set or types that should kept as references.

Every type defined in an assembly must be reviewed for the following list of elements:

Custom Attributes

A Custom Attribute points to a type constructor method and contains optional constructor values. Attributes can occur at assembly level, type level, and method level.

Type

A Type has a parent type except `System.Object` and may implement a number of interfaces. Methods describe operations that may be performed on that type. Fields are named subtypes of a type.

Interface

Interfaces are special types that do not have a super type and contain no CIL code.

Method

A Method is a named operation and is characterized by the types of its parameters. Besides the parameter types also the return type and possible Custom Attributes have to set to the resolved type. Local variables are unnamed subtypes within a method resolution scope. CIL code may have a type, method or field parameter. Exception clauses are defined by a code range and the type of the exception.

Event

Events are handled like fields of a type.

CIL code

The following types of IL codes must be checked for references to types, methods or fields references:

- Type Op.: castclass, newarr, initobj, ...

- Method Op.: call, calli, callvirt, newobj, ...
- Field Op.: ldfld, ldflda, stfld, stflda, ...

The challenge of assembling self-contained assemblies is to verify types for references and generate a consistent PE-file. The current version of self-contained assemblies addresses CLI v1.1 features only. There are further size optimizations practicable. To reduce the size of the constant pool, some kind of type descriptions can be shorten or eliminated. Custom type names not required by the CLR, except special names e.g. type constructor.

Proof of concept results

The current implementation of self-contained assemblies targets desktop CLR like .NET, Rotor, Mono or Portable.NET.

```
public static int Main(string[] args){
  Object obj=new Object();
  return 1;
}
```

The above C# program has a single external reference (`System.Object::.ctor`) in CIL representation. But for the self-contained version a second method from `System.Object` must be imported, because the CLR calls the destructor (`Finalize()`) of the CLI-base type without further reference.

The compiled[1] assembly with mscorlib reference had a size of 3072 bytes. The size of the CLR is not considered, because it assumed to be constant. So the memory footprint with .NET v1.1 mscorlib.dll is 2141184 bytes.

The self-contained version has an oval size of 2048 bytes and contains no references. These results are prestigious in no means, but the potential of self-contained assembly optimization.

To process more complex programs a clean BCL implementation is reasonable, because existing mscorlib.dll implementations are using none BCL features[2] for BCL functionality.

CLR implementation issues

The CLI defines a lot of possibilities for optimized CLR implementations. This section discusses these optimizations in terms of portability of self-contained assemblies among different CLR.

The CLR is responsible for resolving references to assemblies and loading types. References to external types are available in textual representation. CLI metadata are organized as a number of cross referenced tables. A referenced in type in an external assembly can have references to the same assembly or the external assemblies. The CLI suggests resolving all references before start the execution. Therefore all related assemblies must be loaded to create a consistent memory image.

For optimization issues the CLI introduced build in primitive types e.g. `bool`, `char`, `object`, `string`, `...`, which does not induce type references as long no type specific operation were performed.

In contrast to Java the CLI provides an internal mapping of primitive type to their wrapper types. The CLR knows the mapping of primitive types to their wrapper types e.g. `object≡System.Object`. The mapping of primitive types to BCL types, inside the CLR, is realized with string compare, because a type reference is given in textual representation. For types implemented inside a self-contained assembly this mapping is possible further on.

The CLI supports multiple ways to implement type methods. Possible implementation flags [Lid02a] for types inside the BCL:

- *cil*: The method is implemented in CIL code.
- *internalcall*: This flag indicates that the method is internal to the runtime and must be called in a special way.
- *runtime*: The method implementation is provided by the runtime itself.
- *pinvokeimpl*: The method has unmanaged implementation and is called through the platform invocation mechanism P/Invoke.

A *cil* implemented method can be executed by any CLR. An *internalcall* method is not portable among CLR implementations. This flag can occur in the BCL and additional features provided by the CLR. A *runtime* supplied implementation is also CLR dependent. The *pinvokeimpl* flag indicates the CLR provided mechanism (P/Invoke) to call native code. Figure 7 shows three different implementations of the System.Object::Equals(object) method.

The Microsoft .NET Framework uses the *internalcall* manner to perform the comparison. This implies the existence of a dispatch table for *internalcalls*.

Microsoft .NET v1.1.4322
.method public hidebysig newslot virtual instance bool Equals(object obj) cil managed internalcall {}

[1] csc /optimize+ simple.cs

[2] Class attribute System.Runtime.InteropServices.ClassInterfaceAttribute::.ctor in .NET v1.1 System.Object implementation

 ISBN 80-86943-11-9

```
Mono v1.1.13.2
.method public hidebysig newslot virtual instance bool Equals(object obj) cil managed
{
 .maxstack 8
 IL_0000: ldarg.0
 IL_0001: ldarg.1
 IL_0002: ceq
 IL_0004: ret
}
```

```
Compact Framework v1.0.500
.method public hidebysig newslot virtual instance bool Equals(object obj) cil managed
{
 .maxstack 8
 IL_0000: ldarg.0
 IL_0001: ldarg.1
 IL_0002: call bool System.PInvoke.EE::Object_Equals(object, object)
 IL_0007: ret
}
.method public hidebysig static pinvokeimpl("mscoree" as "#17" winapi) bool Object_Equals(object obj1, object obj2) cil managed preservesig {}
```

Figure 7: Implementation of System.Object:: Equals(object) in .NET, Mono and Compact Framework

Mono provides a implementation based on CIL code, which makes the implementation portable.

In the Compact Framework BCL `System.Object ::Equals(object)` is implemented with a additional call through the P/Invoke mechanism.

The current version of self-contained assembly's implementation is portable among different CLR as long as no implementation specifics are used. One can benefit from self-contained features as long as is executed with the CLR that provided the BCL implementation.

5. RELATED WORK

There are several approaches to optimize Java class files to meet the requirements of small embedded devices. The optimizations are often done on a per class basis.

IBM's WebSphere® Studio Device Developer (WSDD) [IBM06a] includes the SmartLinker tool (formerly JAX [alp06a]) to optimize J2ME [Sun06a] applications.
SmartLinker removes unused code, merges classes, and introduces short identifiers to reduce the overall code size. Resulting applications are composed in the Java Executable format (JXE), which is not interoperable with jad/jar format as specified in J2ME.

Rayside et al. [Ray99a] propose a modified Java class file format with significant space reduction with little or no runtime penalty.

Clausen et al. [Cla00a] use macros for multiple occurrences of code fragments and an extended JVM with macro support.

The JamaicaVM[aic06a] developed by aicas GmbH includes a builder tool for integrating Java bytecode and a corresponding Virtual Machine implementation into a single executable application binary. Bytecode is embedded as C-Array definition and linked with the JamaicaVM library.

TinyVM[Sol06a] is a firmware replacement for the Lego™ Mindstorm™ RCX hardware. The firmware executes (interprets) Java programs that are compacted into custom images.

The Lego.NET [Osm05a] project has developed a GCC front-end which translates CIL code into native machine code of the Lego™ Mindstorm™ RCX processor.

Microsoft's .NET Compact Framework is a subset of the .NET platform for mobile and embedded devices. The Compact Framework class libraries occupy at least 2 Megabyte of memory. The assembly format and execution environment differ only in trifles from the desktop version.

Microsoft's ILMerge[Mic06a] is a utility that can be used to merge multiple .NET assemblies into a single assembly. ILMerge does not support a selection of types which should be merged together.

AppForge, Inc. offers with Crossfire[App06a] a product for multi-platform applications for mobile and wireless devices based on .NET. The CIL bytecode is transferred into a custom executable format that is executed by platform specific Crossfire-Client software.

6. CONCLUSION AND FUTURE WORK

This paper proposes an approach of self-contained assemblies to reduce memory consumption and shorter startuptime while executing the assembly. CLI assemblies are loose coupled with other assemblies (shared class libraries, custom libraries).

Creating of self-contained assemblies is done at type level with a customized version of the PERWAPI assembly manipulation library. The compaction of assemblies bases on referenced types of an assembly and requires no source code, nor compiler support. Self-contained assemblies are size optimized in terms of assembly footprint and memory consumption while execution.

Furthermore the effect of an executed self-contained assembly is identical among the acceptance the CLR is CLI-complaint and no CIL-code is executed outside of the assembly.

The customized PERWAPI library allows adaptive compaction at type level that means certain types remain as references.

It has to be analyzed to what extent the abstraction of CLR internals from the BCL implementation could be realized CLI-compliant.

The proof-of-concept results must be analyzed in terms of memory consumption, startup time and execution performance with CLR implementations.

Self-contained assemblies could offer useful features for embedded systems development, for predictable execution behavior and more generally for an adaptive deployment format.

7. ACKNOWLEDGMENTS

We would like to thank the reviewers for their useful comments and suggestions.

8. REFERENCES

[Aic06a] aicas GmbH. JamiacaVM. Available at *aicas.com*, 2006

[And00a] Anderson R. The End of DLL Hell. Microsoft Cooperation. Available at *msdn.microsoft.com/library/en-us/dnsetup/html/dlldanger1.asp*, 2000

[App06a] AppForge, Inc. Crossfire homepage. Available at *www.appforge.com/products/ crossfire*, 2006.

[Cla00a] Clausen L.R., Schultz U.P., Consel C., and Muller G. Java bytecode compression for low-end embedded systems. ACM Transactions on Programming Languages and Systems, 22(3):pp.471–489, 2000.

[Cos05a] Costa R.,and Rohou E. Comparing the size of .net applications with native code. in CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp. 99–104, ACM Press, 2005.

[Dot06a] The DotGNU project. Portable.NET. Available at www.dotgnu.org, 2006

[Ecm02a] Ecma international. Standard Ecma-335, Common language infrastructure (Cli). Available at *www.ecma-international.org/ publications/ standards/Ecma-335.htm*, 2002.

[Gef05a] Gefflaut A., van Megen F., Siegemund F., Sugar R. Porting the .NET Compact Framework to Symbian Phones – A Feasibility Assessment. .NET Technologies'05 conference proceedings, UNION Agency – Science Press, ISBN 80-86943-01-1, 2005

[Gou05a] Gough J., and Corney D. PERWAPI-a pe file reader/writer. Available at *www.plas.fit.qut.edu. au/perwapi*, 2005.

[IBM06a] IBM. WebSphere Everyplace Micro Environment. Available at *www-306.ibm.com/ software/wireless/wsdd*, 2006

[Int03a] International Standards Organisation. Informationtechnology – Common Language Infrastructure, ISO/IEC 23271:2003(E) First edition, 2003.

[alp06a] alphaWorks/IBM. JAX. Available at www.alphaworks.ibm.com/tech/JAX, 2006

[Lid02a] Lidin S. Inside Microsoft .net il assembler.Microsoft Press, 2002.

[Mic02a] Microsoft Corporation. Shared source common language infrastructure. Available at *msdn.microsoft.com/net/sscli*, 2002.

[Mic05a] Microsoft Corporation. .NET Framework. Available at *msdn.microsoft.com/netframework*, 2005.

[Mic05b] Microsoft Corporation. .NET Compact Framework. Available at *msdn.microsoft.com/ netframework/programming/netcf*, 2005.

[Mic06a] Microsoft Research. ILMerge, Available at *research.microsoft.com/~mbarnett/ILMerge.aspx*, 2006

[Mon06a] The Mono project. website. Available at *www.mono-project.com*, 2006.

[Osm05a] Operating systems and middleware group. Lego.net website. Available at *www.dcl.hpi.unipotsdam.de/research/lego.NET/*, 2005.

[Ray99a] Rayside D., Mamas E., and Hons E. Compact java binaries for embedded systems. In CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, page 9. IBM Press, 1999.

[Sol06a] Solorzano J.H. TinyVM website. Available at *tinyvm.sf.net*, 2006.

[Sun06a] Sun Microsystems, Inc. Java Platform, Micro Edition. Available at *javasoft.com/j2me*, 2006

PMPI: A multi-platform, multi-programming language MPI using .NET

Mohammad M. El Saifi Edson Toshimi Midorikawa
Department of Computer Engineering and Digital Systems
Polytechnic School – University of São Paulo
Sao Paulo - SP - Brazil

{mohamad.saifi, edson.midorikawa}@poli.usp.br

ABSTRACT

Implementation of the MPI standard on heterogeneous platforms is desirable because it permits using resources discarded by existing MPI implementations of homogenous systems. This paper describes PMPI, as partial implementation of the MPI standard on a heterogeneous platform. Unlike other MPI implementations, PMPI permits MPI processes written in different programming languages to run on multiplatform system. PMPI is built on top of .NET framework. PMPI can span multiple administrative domains distributed geographically. To programmers, the grid looks like a local MPI computation. The model of computation is indistinguishable from that of standard MPI computation. This paper studies the implementation of PMPI with Microsoft .NET framework and MONO to provide a common layer for a multiprogramming language multiplatform MPI application. We show the obtained results using PMPI, and compare them to MPICH2. The obtained results will show that the use of .NET framework for PMPI is feasible and can be optimized for performance.

Keywords

MPI, Parallel Computing, HPC, .NET Framework, MONO

1. INTRODUCTION

For many years, parallel computation was always an attractive alternative for obtaining high-performance computing [Dongarra et al. 2003] [Foster 1995]. With the use of multiple computational nodes interconnected by a high-speed network, clusters of computers are the most common platform of parallel machines. The recent introduction of multi-core microprocessors will result in parallel computers becoming available on desktops.
MPI is perhaps the best known standard used in parallel computation allowing nodes spread across the network to collaborate to achieve a common computational goal [Andrews 2000] [MPI Forum 1994].

The limitation of MPI is two fold. On the one side, most existing MPI implementations, such as MPICH2, execute only on homogeneous platforms [MPICH2 2006]. Accordingly, idle cycles that are spread across a variety of machine architectures and operating systems across networked PCs, are discarded because of the lack of an MPI that executes on a heterogeneous platform. These idle cycles are increasingly being recognized as a huge and largely untapped source of computing power

On the other side, almost existing MPI implementations use C, C++ or FORTRAN programming language. Accordingly, researchers and programmers who collaborate on the solution of the same problem need to stick to one of the languages that supports the MPI library they intend to use.

The implementation of MPI that can tap into those idle resources on heterogeneous platforms is desirable because it allows researchers and programmers, who need high performance computing and have available heterogeneous platforms around their campus, to use all available resources [Kelly, Roe and Sumitomo 2002][Kelly and Roe 2002][Kelly and Mason 2003]. Having the ability to use MPI on heterogeneous systems maximizes computational power resources.

In addition to using MPI on a heterogeneous platform, programmers want to use a variety of programming languages in their computational program. In the same MPI computation, programmers want nodes to run applications written in different programming languages simultaneously using MPI standards. This becomes a merit when we have multiple programmers participating in the solution of a unique problem, where each programmer is writing a program that runs on a separate node such as same data multiple program solutions. This permits programmers to explore their abilities and skills in their preferred programming language, and to use the programming language that best suit the solution of the problem.

This paper studies the feasibility of implementing MPI standard on a heterogeneous platform by implementing the component PMPI. PMPI aims to provide programmers and researchers with a framework that takes care of a transparent communication infrastructure between the heterogeneous nodes in a MPI computation in a robust and secure manner. The programmer is left to concentrate only on the application specific computational aspects. We take advantage of the .NET framework to provide application programmers with a choice of the programming language, all of which can use the same PMPI framework classes.

There are different choices that can be made to implement the PMPI component. We choose the .NET framework [Ritchter and Balena 2002] for this purpose as the first tentative and used .NET Remoting [McLean 2003] [Rammer 2002] as the communication infrastructure for PMPI. In this implementation, PMPI acts as a remote-object based framework for creating MPI parallel applications. The framework is built using the extensibility features of the .NET Remoting framework.

Unlike the Java virtual machine, the .NET runtime is designed to be language independent. Accordingly, developers can create their applications using any language that targets the CLR such as: C#, Visual Basic, Visual C++ or one of many other .NET languages such as Eiffel, Perl, Cobol, Component Pascal, Smalltalk, or Fortran [Ritchter and Balena 2002]. Today there are about twenty six different programming languages that target the .NET framework [Ritchter and Balena 2002]. PMPI enables programmers to program in a normal MPI fashion, without being concerned what platform or programming language other participating nodes will run.

The main contribution of this paper is to study the feasibility of implementing MPI on a virtual machine and show performance results compared to other existing MPI implementation. This offers programmers who have heterogeneous systems with a library that can reap the available computational power on available machines.

The remainder of this paper is organized as follows. Section 2 describes the architecture of PMPI. Section 3 describes the programming model of PMPI. Section 4 explains the sample application used in the tests. Section 5 describes the results and some preliminary performance figures. Finally, section 6 concludes and discusses future and related works.

2. ARCHITECTURE

PMPI architecture follows the standard structure of a layered networking architecture. PMPI is composed of three components. The first component is PMPI which contains MPI implementation. The second one is the agent that runs on each node participating in the MPI computation. The agent is responsible for starting MPI programs on nodes, and offers administrative information about nodes, in addition to information about administrative domains. The third component is PMPI Gateway, or PIP (Platform Interface Portal). The PIP serves as a gateway to administrative domains to overcome problems raised by firewalls and NAT separating different administrative domains.

Each administrative domain has a PIP known to all agents. Inside PMPI component, there is an address resolution layer that is transparent to programmers. This layer decides on whether to direct MPI calls directly to other nodes or to their corresponding PIPs. This permits programmers the freedom to concentrate on their problem rather than communication implementation.

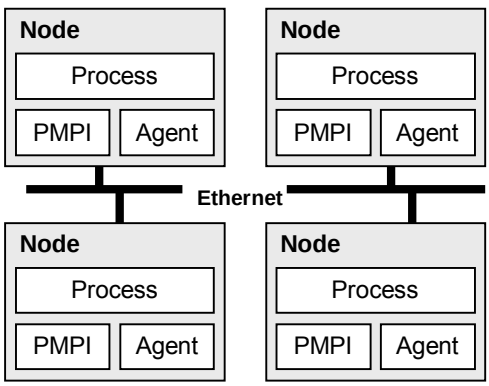

Figure 1: Four nodes using PMPI

Figure 1 shows a basic PMPI infrastructure. The figure shows a structure with four nodes running on one administrative domain connected by local Ethernet network. The processes may be running on different platforms, and each process may be written in a different programming language.

PMPI communication infrastructure is constructed on .NET Remoting, and in turn, is based on TCP/IP. .NET Remoting can be customized to support other protocols [Rammer 2002].

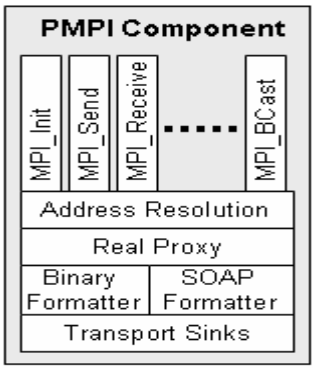

Figure 2: PMPI layered view

Figure 2 shows PMPI component layers. On the top, we have the MPI interface that is available to programmers. When a MPI call is made, it passes through the address resolution module to check which administrative domain the destination node belongs to, and what communication method is to be uses to reach the node that costs less. For example, nodes behind firewalls may be reachable only through port 80 using the SOAP protocol which is firewall friendly in contrast to the binary protocol. On the other hand, SOAP consumes more network bandwidth and is less efficient than binary formatting [McLean 2003].

Figure 3 shows a sketch of a MPI computation spanning two administrative domains where each administrative domain is located behind a firewall. In this figure, MPI calls made from one administrative domain to the other are done through the PIPs of the administrative domain. The PIP will serve as a proxy on behalf of nodes making the call. The scenario in figure 3 assumes that we have barriers in both administrative domains. In other words, nodes in administrative domain 1 cannot reach nodes in administrative domain 2 directly using remote object calls. Instead, they should use the PIP proxy service to exchange messages.

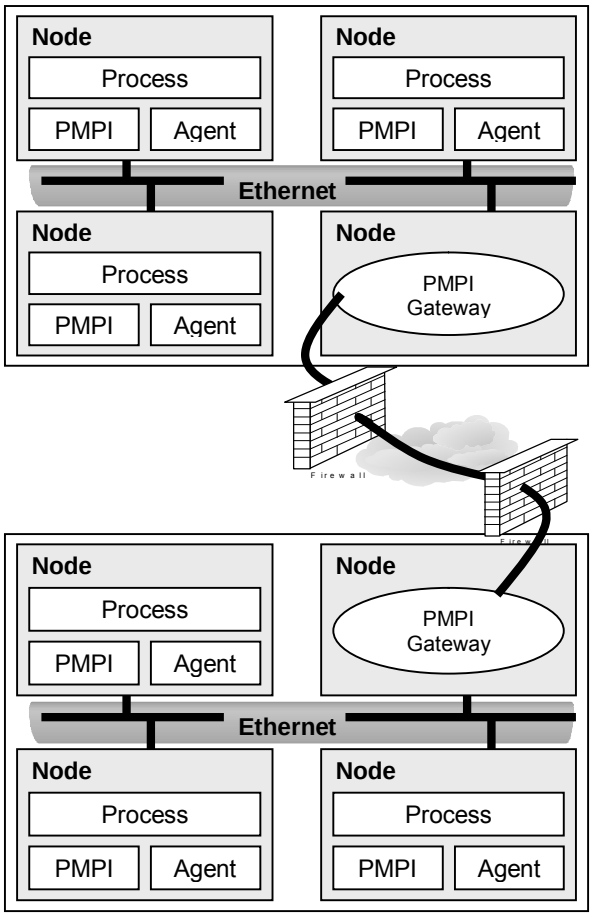

Figure 3: Using PMPI on two administrative domains

To better understand the idea, let's take an example where node A in administrative domain one will make MPI call to node B in administrative domain two. The address resolution layer of PMPI running on node A detects that node B is running on another administrative domain and there is no way to reach node B directly because of a firewall or NAT. The address resolution layer directs the call to the PIP node of administrative domain one. The PIP in turn directs the MPI call to PIP of administrative domain two. The PIP of administrative domain two receives the call and directs it to node B of its domain. If the call is synchronous, then the PIP of administrative domain one block node A until it receives a notification from PIP of the other administrative domain that node B has received the call. The PIP acts as proxy on behalf of the nodes in their corresponding administrative domain.

The rest of this section is divided into two subsections. The first describes MPI standard. The second describes PMPI architecture and constructs.

2.1 MPI: Message Programming Interface

In the message-passing library approach to parallel programming, a collection of processes executes programs written in a standard sequential language augmented with calls to a library of functions for sending and receiving messages. MPI is a complex system. In its entity, it comprises 129 functions, many of which have numerous parameters of variants [Foster 1995].

In the MPI programming model, a computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes. In most MPI implementations, a fixed set of processes is created at program initialization, and one process is created per processor. However, these processes may execute different programs. Hence, the MPI programming model is sometimes referred to as multiple program multiple data (MPMD) to distinguish it from SPMD model in which every processor executes the same program.

Processes can use point-to-point communication operations to send a message from one named process to another; these operations can be used to implement local and unstructured communications. A group of processes can call collective communication operations to perform commonly used global operations such as summation and broadcast. MPI's ability to probe for messages allows asynchronous communication. Probably MPI's most important feature from a software engineering viewpoint is its support for modular programming. A mechanism called a communicator allows the MPI programmer to define modules that encapsulate internal communication structures [MPI Forum 1994].

2.2 PMPI Basic Architecture

PMPI is built on top of .NET framework. We are using Microsoft .NET framework 1.1 for Microsoft Windows and Mono 1.0.5 for Linux. Although Mono can run on Power PC, BSD and other operating systems and architectures, we based our initial implementation on Windows and Linux operating systems although this can be expanded to other operating systems without any modification in the code.

The initial implementation of PMPI was devoted to implement functionality rather that performance. Because of this, we selected higher level implements of the .NET framework to implement PMPI. For the communication layer, we used .NET Remoting which is based on remote object communication. The classes that make up the .NET framework are layered, meaning that at the base of the framework are simple types, which are built on and reused by more complex types. .NET Remoting is one such complex type which in turn is built as layers where each layer can be customized to programmer needs [Jones et al 2004]. This adds extra overhead compared to using simple raw classes such as socket class [Rammer 2002].

We used C# as the programming language. All .NET programming language compilers targets the CTS (common type system) of the framework. C# compiler helps the programmer adhere to CTS types by setting the "CLSCompliantAttribute" attribute to true [Bock 2003]. In this way, the compiler generates an error whenever you try to use a non CTS type. This guarantees that the generated code is accessed by all .NET programming languages since all .NET programming languages target the CTS [Ritchter and Balena 2002].

Each node participating in the MPI computation should have the .NET framework installed. Nodes running Windows operating systems should install Microsoft Framework 1.1 on their machines. Nodes running Linux should install Mono 1.0.5. Although there are newer versions of the framework for both platforms, PMPI has been tested on earlier frameworks.

In addition to the framework installed on the machines participating in the MPI computation, the nodes should have PMPI installed on each node. The initial implementation of PMPI needs to have bidirectional communication between the nodes. Accordingly, firewalls can cause problems. The implementation of PIP is not yet implemented.

Initially, PMPI implemented 20 MPI functions. Those functions cover basic, asynchronous, collective and modular commands. When MPI computation starts, each node registers PMPI object at a known end point to other nodes using .NET remote object. With .NET remoting, the framework creates a thread pool to receive the calls made against the remote object. When node A sends data to node B within the same administrative domain, node B's PMPI will receive the data and releases the calling object immediately, node A in this. When node B calls MPI_Receive, PMPI will check to see if there is a message with the corresponding tag and source. If it finds a corresponding message, then a pointer to the message is passed to MPI_Receive, and the call returns immediately in node B. If no corresponding message is found with the requested tag-source, the call in node B is blocked until node B receives the requested message. If node A uses synchronous MPI_SSend, then PMPI layer on node A blocks until node B

sends a release signal after the process in node B makes a call to MPI_Receive.

PMPI uses a hash table data structure to control received message. The key of the hash table is a combination of the source, tag, and communicator ID. The value of the hash table points to a queue whose elements contains a data structure composed of the received message, message size, message type and synchronization objects that the receiving thread will block on. When the node calls MPI_Receive with a particular tag, source and communicator, PMPI checks the hash table for pending messages in the queue. If it finds a message, it pops the message from the queue in a FIFO manner and wakes up the thread using the synchronization objects found in the read queue element. When the waked thread terminates, the message is passed to the MPI_Receive call. Note that if the call is made using MPI_Ssend, which is a synchronous send, the receiving thread will block the sending thread until it is waked up again by MPI_Receive in the manner explained above. If it comes that MPI_Receive is called before a MPI_Send and PMPI finds the queue empty, then it blocks the call on synchrozination objects, enqueue the call with the synchronization objects in the queue whose pointer is stored in a hash table. Later, when PMPI is invoked by MPI_Send, PMPI checks first if a pending MPI_Receive exists. If it find a pending receive, then it pops the queue, wakes the thread using the popped synchronization objects and returns.

When it comes to collective operations, PMPI uses a thread pool to perform the collective task. PMPI uses a simple algorithm for collective tasks. Each communicator has a master node known to all participating nodes. The communicator master node is responsible for coordinating the collective calls. In other worlds, its the master communicator node who decides when the collective call is done. PMPI implements this by using a thread pool in the communicator master node. When the collective call is made, PMPI checks if the node is the master in the target communicator. If it is not, then it uses a methodology similar to Send_Receive explained before. If it finds the node to be the communicator master, then it creates one thread for each node in the communicator, and blocks on the synchronization object. When the thread in the pool terminates, it verifies if other threads in the pool had terminated; if not, then the thread blocks on a synchronization object. If the thread happens to be the last one, then the thread wakes all other threads using the synchronization object. By this means, the communicator master manages the collective operation.

The agents will be a separate component. For MS Windows, the agent is implemented as Windows Service. The agent will be responsible for starting the programs on participating nodes. In addition, the agent will supply managing data about the nodes themselves such as available memory, CPU load, speed, administrative domains and other managing data. Today, most operating systems implement the Web-Based Enterprise Management (WBEM), which is an industry initiative to develop a standard technology for accessing management information in an enterprise environment. WMI is the Micorsoft implementation of WBEM.

The PIPs are part of PMPI architecture but are not yet implemented. PIPs will be implemented using Web Services. The remote object model explained will be substituted by Web Service model. The PIP will be a gateway on behalf of the calling node. The architecture and implementation of PIP will consider having two communicating PIPs on behalf of the send and receiving nodes.

3. PROGRAMMING MODEL

The programming model is as simple as any existing MPI implementation. The master node initializes the MPI computation using a XML computation file. PMPI is object based. Therefore, the MPI functions should be called as object methods.

When PMPI is initialized, it publishes a remote object at a known end point. Each participating node knows the address and port of all other nodes in the MPI computation. When the program calls a MPI function, PMPI receives the function call and transmits it to the corresponding node after resolving its address internally. Although current implementation did not target nodes running behind NATs and firewall, PMPI layered implementation makes it easy to build semantics to solve the complications raised by firewalls and NATs with out programmer awareness. This helps the programmer to devote his efforts on programming rather than MPI communications. Future works will customize the real proxy of the .NET Remoting object to intercept message calls and select the destination accordingly.

We wrote applications in VB.NET, C#, managed C++, and J#. We ran each application on a different node. All four nodes ran under Microsoft Windows XP operating system. For MONO running on Linux Redhat 9, we were limited to C# since it is the only existing non-beta compiler. For simplicity, we used only the above programming languages, but this can extend to any available .NET programming language. The MPI computation ran as if programs

at all nodes were written in the same programming language.

```
MPI obj = new  MPI();
obj.MPI_Init(args);
id=obj.MPI_Comm_Rank(MPI_Comm_World);
tasks=obj.MPI_Comm_Size(MPI_Comm_World);
obj.MPI_Send(offset, 1,
             MPI_Integer, dest, mtype,
             MPI_Comm_World);
obj.MPI_Send(rows, 1, MPI_Integer, dest,
              mtype, MPI_Comm_World);
```

Figure 4: Part of the sample application

Figure 4 shows part of the sample application written in C# where the code initializes an MPI computation, gets its task Id within COMM_WOLRD, gets COMM_WORLD size, sends data to "dest" node and later receives data from "dest" node. Note that the MPI functions are methods of a PMPI object called "obj". These methods are either static or instance methods. Static methods of PMPI enable us to write multithreaded programs running on a machine where all threads use the same PMPI object. Also, it is possible to start multiple PMPI objects where each object participates in a different MPI computation with out the need to MPI communicators.

4. SAMPLE APPLICATION

We used as a sample application the master-worker model for matrix multiplication (A x B = C). The results of this sample are compared to MPICH2 for Windows in the next section.

The master (task Id 0) sends matrix B to all participating nodes (workers), and distributes the rows of matrix A into worker nodes evenly. Workers perform the multiplication and send back the result to the master node. Master node accumulates the results from all workers into matrix C. The sample application was taken from the examples that install with MPICH2. In this sample application, the master does not participate in the MPI computation. It just sends the data to workers and gets back the results into matrix C.

5. RESULTS

The performance tests are done with the sample application written in C#. We set the number of columns in matrix A to 1200 and the number of columns of matrix B to 500. We varied the number of rows of matrix A to 2400, 4800, 9600 and 19200 respectively. For each problem size, we executed the application on one to all six nodes.

The tests are executed in three sets. The first set of tests is the results obtained executing the sample application on a homogeneous platform corporate network. The second test is done on the same corporate network with both PMPI and MPICH2. The last test is done on a cluster using homogeneous and heterogeneous platforms.

5.1 Results using Corporate Homogenous Platform

We tested the application first on standalone machines with out using parallel MPI computation. We rewrote the application taking out all MPI commands and compiled them using Microsoft Visual C++, Microsoft C# and MONO C# compilers.

The corporate network was composed of AMD 1.5 GHZ, 512 KB cache CPUs with 256 MB RAM and 40 GB HD. The nodes run under Windows XP. One node had dual operating systems: Windows XP and Redhat 9. The obtained results are as follows. C# managed code application executed 27% slower than C++ application on machine running Windows XP or Windows 2003 operating system. On machine running Linux Redhat 9 with mono .NET framework, C++ executed 10 times faster than C#! Comparing .Microsoft NET C# running on Windows XP to MONO 1.05 C# compiler Running under Linux Redhat 9, Microsoft C# executed 5 times faster than MONO C#.

Before going any further, let me clarify some details about array access in managed world and some performance issues. Each time an element of an array is accessed, the CLR ensures that the index is within the array's bound. This prevents you from accessing memory that is outside the array, which would potentially corrupt other objects. If an invalid index is used to access an array element, the CLR throws a System.IndexOutOfRangeException exception.

The index checking comes at a performance cost. If we have confidence in our code, we can access an array without having the CLR perform index checking. This feature is not allowed in all .NET languages and is not CLS complaint. Accordingly, only .NET languages that have this feature will benefit from fast array access such as C#.

To give an idea on how much gain we get using fast array access, we show the following results. C# using managed array access executes 20% slower than C# using fast array access on the machine running Windows XP. On Linux, C# using managed array access, executed 5 times slower than C# using fast array access. As we note, the performance gain in Linux is huge (500%).

The problem with fast array is that not all .NET languages support it since it is not a CLS compliant. In addition, it is harder to code than managed array access since it uses pointers. Accordingly, the benefit of using fast array is limited to only a subset of .NET programming languages.

Later, we executed the application using both MPICH2 and PMPI using managed array access with PMPI. The sample application running on PMPI nodes was written with C#, Java.NET, managed C++ and VB.NET. The compiler choice did not affect the result. We used a various combination of the programming languages and we got the same results. The results are shown only for Windows OS since we used MPICH2 for windows.

In figures 5, we show a comparison between PMPI and MPICH2 for different problem sizes executing on 6 nodes. The results demonstrate that PMPI executed slower than MPICH2 between 40% and 70%.

Figure 6 shows the linear relation ship between the number of nodes and the execution time. As we increase the participating nodes, the execution time decreases linearly.

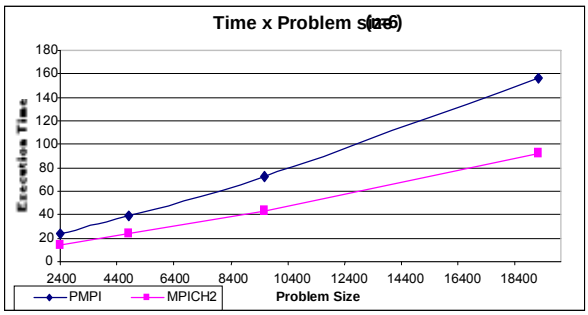

Figure 5: comparison between PMPI and MPICH2

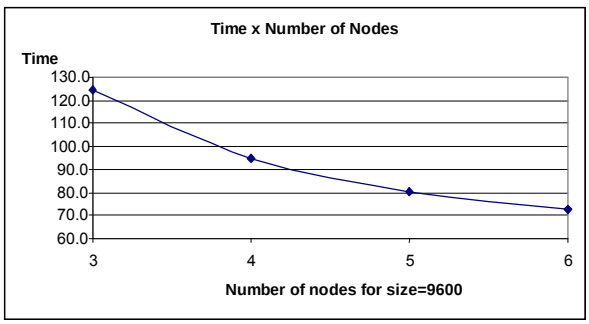

Figure 6: Execution time as a function of participating nodes

5.2 Results using cluster with a Heterogeneous Platform

The cluster, named BIO, is composed of 8 nodes each with dual 2.0 GHZ, 512 KB cache CPUs with 512MB RAM and 40 GB HD.

As before, we tested the application first on a standalone machines with out using parallel MPI computation. We rewrote the application taking out all MPI commands and compiled them using Microsoft C# compiler and mono C#.

Later, we executed the application on the cluster using up to six nodes where nodes varied between nodes running Windows 2003 server and nodes running Linux Redhat 8. The result is shown below in figure 7. As the figure shows, Microsoft .NET platform performed better than MONO .NET framework. When we mixed the nodes between Windows and Linux operating systems, PMPI executed with performance equivalent to the average of executing on each platform independently.

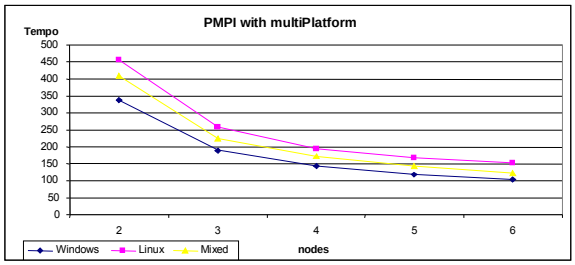

Figure 7: PMPI on a heterogeneous platform

5.3 Result analysis

As shown in section 5, PMPI executes as a linear function of the problem size. The execution time increased linearly as we increased the matrix size. Also, as we increase the number of nodes, the execution time decreased almost linearly.

Although PMPI executes slower than MPICH2, the main overhead is a result of managed array access and the use of high construct communication construct of the .NET framework. This overhead was expected and is subject for future work.

In addition, we detected that the use of thread pool within the program structure, degraded PMPI performance in a master-worker model. This loss of performance resulted from the fact that the operating system has full control of the thread pool which resulted in activating threads to receive the results from nodes while other threads were still sending data to other nodes. With a custom thread pool, PMPI will have full control on the executing thread, and in turn, can block receiving threads while PMPI is sending. This will improve a lot performance especially when we have large number of nodes. This happens because as we increase the number of nodes, we have greater the tendency of nodes completing their jobs before the master.

Moreover, there are some other code tuning of PMPI that can improve performance such as reducing .NET framework boxing, a mechanism that .NET framework exchange data between the allocated stack and managed heap. Boxing in .NET managed code is known to have performance cost and minimizing it can improve performance a lot.

6. RELATED WORKS

In this section we discuss related work the can be use parallel computing on a multiplatform. In [Fer05], experiment with implementation of parallel programs using C# running on Unix and Windows is done. In [Will01], a binding between an already implemented MPI interfaces and C# is done. In [Car00], a Multiplatform MPI implementation is done for JAVA programming language. However, none of the above works have focused and worked with a Multiprogramming Language MPI.

7. CONCLUSION AND FUTURE WORKS

The first implementation of PMPI was shown to be feasible and it is possible to execute MPI standards on a multi-language and multiplatform systems. Although the first implementation showed that PMPI is slower than MPICH2, the difference is explained by known issues and these issues can be eliminated. Care should be taken when using a heterogeneous system including Linux with managed array access. As shown in the preliminary results, mono performs very poor with managed array access. In such a case, we should consider using fast array access.

The next step in this project is to span PMPI to multiple administrative domains that span geographic area across the internet. In addition, lower communication constructs can improve performance in addition to use a custom thread pool to manage threads instead of the operating system thread pool. This will give us a complete control on the threads. Also, we will do a comparison between JavaMPI to PMPI .

REFERENCES

[And00a] Andrews, G.R. Foundation of Multithreading, Parrallel, and Distributed Programming, pp 115-243, 2000.

[Rit02a] Ritchter,J. and Balena,F. Applied Microsoft Dotnet Framework Programming in Microsoft C# 2002.

[Fos95a] Foster, I.. Designing and Building Parallel Programs, pp 275-310, 1995

[Don03a] Dongarra,J. and Foster,I. and Fox,G. and Gropp, W., Kennedy,K. and Torczon,L. White,A. Sourcebook Of Parallel Computing. 2003.

[Ram02a] Rammer, I. Advanced Dotnet Remoting in C#.2002.

[Boc03a] Bock,J. and Barnaby,T. Applied Dotnet Attributes. 2003

[East04a] Easton, M.J. and King, J. Cross-Platform Dotnet Development. 2004

[Jon04a] Jones, A., Ohlund,J. and Olson, L. Network Programming for the Microsoft Dotnet Framework. 2004.

[Ard02a] Ardestani, K. and Ferracchiati, F. and Gopikrishna,S., Redkar,T., Sivakumar, S., Titus, T. Visual Basic Dotnet Threading. 2002.

[Sha03a] Sharp, J. and Jagger, J. Microsoft Visual C# Dotnet. 2003.

[McL03a] McLean, S. and Naftel,J. and Williams,K. Microsoft Dotnet Remoting. 2003.

[Mar04a] Mariani,R. , Bohling, B., C.Smith, and S.Barber. Improving Dotnet Application Performance and Scalability. 2004.

[MPI94a] MPI FORUM. 1994. The MPI message passing interface standard. University of Tennesse,Knoxville.

[MON05a] The MONO project. http://www.go-MONO.com

[ECMa] ECMA ISO/IEC 23270, ISO/IEC 23271 and ISO/IEC 23272. http://www.ecma.ch and http://msdn.microsoft.com/net/ecma

[Kel02a] Kelly,W., Roe,P. and Sumitomo,J. , *G2: A Grid Middleware for Cycle Donation using Dotnet* , The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, June 2002.

[Kel02b] Kelly,W. and Roe,P., *Donating Cycles over the Internet Using Web Services* , The Eighth Australian World Wide Web Conference, Sunshine Coast, July 2002

[Fer05] Ferreira, F and Sobral, Joao, *ParC#: Parallel Computing with C# in .Net*,* Springer-Verlag Berlin Heidelberg 2005

[Will01] Willcock,J and Lumsdaine,A and Robison,A, Using MPI with C# and the Common Language Infrastructure Indiana University Computer Science Department Technical Report 570

[Car00] Carpenter,B, Getov,V, Judd,G, Skjellum,T and Fox,G MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience* Volume 12, Number 11. September 2000