The 2nd International Workshop on .NET Technologies

.NET Technologies' 2004

Workshop proceedings

held at
University of West Bohemia
Plzen, Czech Republic

May 31 – June 2, 2004

Edited by

Vaclav Skala, University of West Bohemia, Plzen, Czech Republic
Piotr Nienaltowski, ETH Zurich, Switzerland

.NET Technologies' 2004

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 - Science Press
Na Mazinách 9
322 00 Plzen
Czech Republic

Printed at the University of West Bohemia

Hardcopy: *ISBN 80-903100-4-4*

Preface

This volume contains the proceedings of the 2nd International Workshop on .NET Technologies (.NET Technologies 2004) held in Pilsen, Czech Republic, May 31 – June 2, 2004.

The purpose of the .NET Technologies workshop series is to bring together practitioners and researchers from academia and the industry to discuss the latest developments in .NET and advance the state of the art in the research on related technologies. Interest in these topics has been continuously growing, as a consequence of the importance and the ubiquity of object-oriented technologies.

For .NET Technologies 2004, papers describing theoretical and practical results were solicited in the following areas: software engineering, programming languages and techniques, parallel and distributed computing, algorithms and data structures, educational aspects of .NET, support for .NET on Unix.

Out of 31 regular papers submitted this year, 11 were accepted for presentation at the workshop and are included in this volume. The workshop programme also featured four talks by invited speakers: Bertrand Meyer (ETH Zurich/Eiffel Software), K. Rustan M. Leino (Microsoft Research, Redmond), Harald Haller (sd&m AG), and Damien Watkins (Microsoft Research, Cambridge). The panel discussion on *The next big step in the development of .NET* and a session of short talks completed the programme. The workshop was accompanied by a two-day intensive .NET course for Czech and Slovak participants.

We are very grateful to the Programme Committee members for their tremendous work and for establishing a selective workshop programme. We also thank the invited speakers for their interesting contribution. We highly appreciated the work of the referees. Last but not least, we would like to thank the authors of the submitted papers.

Special thanks are due to the publisher of the Journal of Object Technology (JOT, http://www.jot.fm) who kindly accepted to publish the extended version of the best six papers in a special issue of JOT dedicated to the .NET Technologies 2004 workshop.

We gratefully acknowledge the financial support provided by Microsoft Research Ltd.(U.K.) and Microsoft Czech Republic

June 2004

Vaclav Skala
Piotr Nienaltowski

Workshop Committees

Workshop Co-Chairs

Vaclav Skala	University of West Bohemia, Plzen, Czech Republic
Piotr Nienaltowski	ETH Zurich, Switzerland

International Programme Committee

Mike Barnett	MSR Redmond, USA
Judith Bishop	University of Pretoria, South Africa
Antonio Cisternino	University of Pisa, Italy
Marieke Huisman	INRIA Sophia-Antipolis, France
Josef Kolar	Czech Technical University, Czech Republic
K. Rustan M. Leino	MSR Redmond, USA
Peter Mueller	ETH Zurich, Switzerland
Nigel Perry	University of Canterbury, New Zealand
Michel Riveill	University of Nice, France
Vladimir Safonov	Sankt Petersburg University, Russia
Peter Sturm	University of Trier, Germany
Don Syme	MSR Cambridge, U.K.
Werner Vogels	Cornell University, USA

Reviewers

Volkan Arslan	Mike Barnett	Judith Bishop
Antonio Cisternino	Juerg Gutknecht	Marieke Huisman
Josef Kolar	K. Rustan M. Leino	Peter Mueller
Nigel Perry	Michel Riveill	Marie-Helene Ng Cheong Vee
Vladimir Safonov	Peter Sturm	Don Syme
Werner Vogels	Eugene Zueff	Sebastien Vaucouleur

.NET Technologies'2004

(http://dotnet.zcu.cz)

Contents

Page

Keynotes

Meyer,B.: *Language interoperability at work: Eiffel on .NET (Switzerland)* **i**

Leino,R.M.: *Spec#: Writing and checking contracts in a .NET language (USA)* **i**

Watkins,D.: *C Omega: C#, Concurrency and Data Access (U.K.)* **ii**

Haller,H.: *How to implement large applications successfully in .NET (Germany)* **ii**

Tolksdorf, R.,Liebsch,F., Nguyen,D.M: *XMLSpaces.NET: An extensible tuplespace as XML middleware (Germany)* 1

Contreras,M., German,E., Chi,M., Sheremetov,L.: *Design and implementation of a FIPA-compliant agent platform in .NET (Mexico)* 9

Benedek,Z.: *A framework built in .NET for embedded and mobile navigation systems (Hungary)* 17

Bres,Y., Serpette,P., Serrano,M. et al. - *Compiling Scheme programs to the .NET Common Intermediate Language (France)* 25

Terekhov,A., Boulychev,D., Moscal,A., Voyakovskaya,N.:. - *Teaching compiler development using .NET platform (Russia)* 33

Anderson,T., Eng,M., Glew,N., Lewis,B., Menon,V., Stichnoth,N.: *Experience integrating a new compiler and garbage collector into Rotor (USA)* 41

Guentensperger,R., Gutknecht,J.: *Active C# (Switzerland)* 49

Alvarez Gutierrez,D., Dias Fondon,M.A., Suarez Rodriguez,I.: *Alternative protection systems for OO environments: Capability-based protection and the SSCLI-Rotor (Spain)* 57

Sturm,P., Fischer,D., Fusenig,V., Scherer: *The .NET CF implementation of GecGo: A midleware for multihop ad-hoc networks (Germany)* 65

de Rosa,F., Mecella,M.: *Peer-to-Peer applications on mobile devices: A case study with Compact .NET on Smartphone 2003 (Italy)* 73

Fruja,N.G.: *The correctness of the definite assignment analysis in C# (Switzerland)* 81

Keynote talks

Spec#: Writing and checking contracts in a .NET language

K. Rustan M. Leino, Microsoft Research, USA

Abstract
The use of various forms of contracts, like preconditions, are increasingly receiving more attention within Microsoft. This talk describes the design of Spec#, an experimental superset of the language C#, including pre- and postconditions and object invariants. Spec# gives rise to dynamic checks of contracts. The contracts can also be checked statically using the automatic checker Boogie. The talk also reports on some initial experience and describes some difficult issues in the design.

Language interoperability at work: Eiffel on .NET

Bertrand Meyer, ETH Zurich, Switzerland

Abstract
Eiffel on .NET takes advantage of the language interoperability mechanisms specified by the Common Language Interface to provide the full power of the Eiffel method and language, including Design by Contract, multiple inheritance, genericity and seamless support for analysis and design, while ensuring full compatibility with components and applications rewritten in other CLS-compliant languages. The talk will present the issues that were faced during the implementation of Eiffel for .NET and the technical solutions retained; it will discuss the benefits of multi-language programming and provide a number of application examples.

C Omega: C#, Concurrency and Data Access

Damien Watkins, Microsoft Research, U.K.

Abstract
In the last decade, strongly-typed, garbage-collected object-oriented languages have left the laboratory and become mainstream industrial tools. This has led to improvements in programmer productivity and software reliability, but some common tasks are still harder than they should be. Two of the most critical for the development of the next generation of loosely-coupled, networked applications and web services are concurrent programming and the processing of relational and semi-structured data. C Omega is an experimental language designed at Microsoft Research that makes programming with concurrency and external data simpler and less error-prone.

The C Omega approach to both the data and control issues is to extend C# with new, first-class, types and language constructs, rather than relying on external libraries, mappings and tools. The concurrency extensions, based on the join-calculus, provide a simple and powerful asynchronous programming model, which is applicable in both the local (multiple threads on a single machine) and distributed (asynchronous message over a LAN/WAN) settings. The data extensions add new type constructors giving first-class support of both relational (database tables) and semi-structured (XML trees) data. Generalized member access allows XPath-like processing to be expressed directly within the language, and checked by the compiler.

How to implement large applications successfully in .NET

Harald Haller, sd&m AG, Germany

Abstract
For Microsoft .NET is the platform of choice for software development. In this talk we demonstrate how to design and implement large applications with .NET technology. Based on the experience of several .NET projects we explain design concepts and architecture for client-server systems, web applications and web services that were successfully implemented in projects. Useful extensions of the development environment Visual Studio.NET for team development are explained. Finally we show typical problems during the development and their solution as well as proven concepts for the design of standard components such as GUI, communication, data access and integration of existing applications.

XMLSpaces.NET: An Extensible Tuplespace as XML Middleware

Robert Tolksdorf, Franziska Liebsch,Duc Minh Nguyen
Freie Universität Berlin, Inst. für Informatik, AG Netzbasierte Informationssysteme
Takustr. 9, D-14195 Berlin, Germany
research@robert-tolksdorf.de,franziska@adestiny.de,nguyen@inf.fu-berlin.de

ABSTRACT

XMLSpaces.NET implements the Linda concept as a middleware for XML documents on the .NET platform. It introduces an extended matching flexibility on nested tuples and richer data types for fields, including objects and XML documents. It is completely XML-based since data, tuples and tuplespaces are seen as trees represented as XML documents. XMLSpaces.NET is extensible in that it supports a hierarchy of matching relations on tuples and an open set of matching amongst data, documents and objects.

1 INTRODUCTION

According to [3], middleware for XML-centric applications can be classified as middleware that supports XML-based applications – for example, a class library providing an XML-parser –, as XML-based middleware for applications – for example, a protocol suite that uses XML-representation for messages –, or as completely XML-based middleware – an example is the XML-based XSL language which transforms XML documents.

XMLSpaces ([10, 11]) extends the Linda coordination language by establishing a distributed shared space in which XML documents are stored. A process, object, component or agent contributing a result to the overall system will emit it as an XML document to the XMLSpace. Here, it is stored until some other active entity retrieves it. For retrieval, a template of a matching XML document is given. The matching relations possible are manifold, currently, XMLQueries, textual similarity of XML documents and structural similarity wrt. a DTD are supported.

XMLSpaces follows the Linda concept of uncoupled coordination. Producers and consumers of information do not have to reside at the same location. Also, they do not need to have overlapping lifetimes in order to communicate and to synchronize. The producer can well terminate after putting a document into the space while the consumer does not even exist. The consumer can try to retrieve a matching document while the producer has not started to exist. This uncoupledness in space and time makes the Linda concept attractive for open distributed systems.

XMLSpaces adds to Linda expressibility by providing a richer type of exchanged information. While Linda deals only with tuples composed of a set of primitive data types, XMLSpaces allows any well-formed XML document in tuple fields. The set of matching relations is not fixed but can be extended. The distribution and replication schema implemented in XMLSpaces is well-encapsulated and extensible.

XMLSpaces was implemented at TU Berlin on top of Java using RMI. For the basic tuplespace functionality, it relied on TSpaces, an IBM implementation of Linda with small extensions. In addition, it implemented a set of matching relations and a set of distribution strategies.

Following the above classification, XMLSpaces is middleware that supports XML-applications. In this paper, we describe an evolution of XMLSpaces, called XMLSpaces.NET which goes even further and tries to be a self contained XML-middleware. It consists of two parts. First, the implementation of an XMLSpaces kernel in C# that includes the basic coordination mechanisms and the specific XML support. Second, the implementation of a distributed XMLSpaces on top of the .NET framework. In this paper we describe the ideas for a complete XML-representation for both tuples, subtuples and tuplespaces in XMLSpaces.NET, its architecture and current implementation on the .NET platform.

2 TUPLESPACES IN XML

A generic middleware has to offer means to exchange data, documents and objects among distributed applications. See [3] for a review of the historic distinction between object- and document-oriented middleware. XMLSpaces.NET provides an integrated representation of data in standard Linda-tuples, objects from common programming platforms and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
.NET Technologies'2004 workshop proceedings,
ISBN 80-903100-4-4 Copyright UNION Agency - Science Press, Plzen, Czech Republic

documents in XML representation. The operations – following the Linda coordination language – implemented in XMLSpaces.NET become more powerful since they can be applied to all three mentioned kinds of data of interest in a uniform manner.

2.1 XML-based Tuplespaces

A standard Linda-tuple is a list of fields. Those fields carry values from or denote some primitive types, usually from that of a host language. For richer structuring of tuples, XMLSpaces.NET extends that basic notion by allowing nested tuples. An XMLSpaces.NET-tuple thus contains a sequence of fields or XMLSpaces.NET-tuples and is actually a tree of a certain "depth" with primitive data or objects as leaves. Such a *tupletree* is sufficient to represent all our tuples, since fields cannot contain references. The common Linda operations supported by XMLSpaces.NET always manipulate a complete tuple at a time, so the structure of an existing tupletree is never changed or manipulated.

As mentioned above, we strive for a middleware that supports data, documents and objects. A standard Linda-tuple can be considered as data with fields being primitives from some simple type-system. Lindas standard matching scheme can be applied for such tuples. For now, we leave the aspect of matching nested tuples open.

To support documents, we allow well-formed XML documents as tuple fields. The aforementioned XMLSpaces already allowed for tuples that contained XML documents and offered a set of matching relations to select tuples containing XML documents as fields, for example by referencing a DTD to which a document in a field had to comply. Furthermore, a tuple can contain an object from some programming language – Java objects or .NET objects are examples. Matching on them is object- resp. class-specific.

Our aim is to design an integrated and self contained XML-middleware. So far, we have talked about tuples, primitive data, XML documents and objects. For XMLSpaces.NET we have to find a uniform notion that integrates these. The natural choice is, of course, to use an XML representation for the tuples. A tuple (and a nested tuple, too) is a tree with fields as leaves or nested tuples as subtrees. It is obvious, that there can be an XML representation for such tuples. XML documents in fields are trees, since they are wellformed. Finally, the objects that we want to support can also be considered as trees, at least there can be some tree - based serialization of them. It is a reasonable assumption that in a modern object system, one can generate an XML-based serial representation which maps an object into an XML-document.

With that XMLSpaces.NET takes the idea of an XML based coordination medium a step further, since any tuple in XMLSpaces.NET is an XML document. We can go on to apply that principle to tuplespaces.

A tuplespace is a collection of tuples. In the case of multiple or nested tuplespaces, it is a collection of tuples and spaces. The tuplespaces are in any case also trees.

For XMLSpaces.NET, we consider a tuplespace as a collection of XML documents as described. This collection can be represented, in turn, as another tree similar to the tupletree described. The tuplespace differs from tuples in that it cannot contain fields as direct descendants of the root node.

So – at least conceptually – XMLSpaces.NET considers the complete coordination medium as a single XML document with the first level being the tuplespace (or one or several levels in the case of multiple or nested spaces) and the further levels being tuples and nested tuples. The leaves of this one XML document are the fields which are primitives, XML documents or XML serializations of objects. This view is one contribution of XMLSpaces.NET

2.2 Matching in XMLSpaces

Fields in Linda tuples are either *formals* – containing only a type as in ⟨?int⟩ – or *actuals* containing a typed value as in ⟨2⟩. Tuples that contain formals are considered templates in Linda.

In XMLSpaces.NET an item used with tuplespace operations can be classified as a tuple or a template. A tuple contains only actual fields or tuples as fields, like ⟨1,2⟩ or ⟨1,⟨2,3⟩⟩. A template can also contain formal fields or templates like ⟨1,?int⟩ or ⟨1,⟨?int⟩⟩. The set of tuples is a subset of templates.

We do not introduce the classification as typing in XMLSpaces.NET, since this would require us to consider either tuples as subtypes of templates (they are more special in that they cannot contain formals), or vice versa (templates are more special in that they can contain formals). The *in* and *read* operations expect something that is classified as a template, an out something classified as a tuple. So the item ⟨1,2⟩ is classified by its *use* in an operation as a tuple or a template.

Matching in XMLSpaces.NET distinguishes actuals and formals as in Linda. Any matching tuple and templates must have the same length, that is the same number of fields and subtuples or subtemplates.

We now distinguish two extreme kinds of matching when considering subtuples. *FlatTemplate*-matching performs matching only on the fields of the first level of the tupletree. The content of fields containing primitive data, XML documents or objects is not even tested for equality or type-equivalence but only considered as being of the metatype "tuplefield". Similar, nested tuples and templates are only considered as being of the metatype "subtuple/subtemplate". It suffices that *some* (sub-)subtuple is present in a field, its structure and content is not considered further. In contrast to that, *DeepTemplate*-matching performs a complete recursive matching of the content of contained subtuples and templates considering type- and value-equivalence.

We write $\langle 1,2\rangle_D$ for a template that requires deep matching and $\langle 1,2\rangle_F$ for one with flat matching. A tuple ⟨1,⟨2⟩,3⟩ will be matched by a template $\langle 1,\langle 2\rangle_D,3\rangle_D$, but not by $\langle 1,\langle 0.0\rangle_D,3\rangle_D$. Deep matching is intuitively the standard Linda matching recursively applied to nested tuples. Flat matching transforms the typing to a metalevel. A flat template $\langle 1,\langle 2\rangle_F,3\rangle_F$ matches both ⟨1,⟨2⟩,3⟩ and ⟨1,⟨0.0⟩,4⟩. The template is transformed into ⟨F,T,F⟩, where F means field and T means tuple. Flat and deep matching can be combined. $\langle 1,\langle 2\rangle_F,3\rangle_D$ matches ⟨1,⟨2⟩,3⟩ and ⟨1,⟨0.0⟩,3⟩ but not ⟨1,⟨0.0⟩,4⟩.

Finally, flat matching takes precedence over deep matching. In a template $\langle 1,\langle 2\rangle_D,3\rangle_F$, the second field will be transformed to the metatype T, overriding the deep matching.

This means that $\langle 1,\langle 2\rangle_F,\langle 3\rangle_D\rangle_F$ is equal to $\langle 1,\langle 2\rangle_F,\langle 3\rangle_F\rangle_F$. We therefore make deepmatching the default and require only the notation for flat matching if necessary. So we write $\langle 1,\langle 2\rangle_F,3\rangle_D$ as $\langle 1,\langle 2\rangle_F,3\rangle$ and $\langle 1,\langle 2\rangle_F,\langle 3\rangle_F\rangle_F$ as $\langle 1,\langle 2\rangle,\langle 3\rangle\rangle_F$.

It turns out that there are further interesting relations between flat and deep matching. While flat matching ignores all further characteristics of fields and subtuples, *flat/size* matching requires that subtuples must be of the same size as the one given as template. Size is defined as the sum of the number of fields and subtuples. We write $\langle\ldots\rangle_{FS}$ for a template that requires this matching. The template $\langle 1,\langle 2\rangle_{FS},3\rangle_D$ matches $\langle 1,\langle 0.0\rangle,3\rangle$ but neither $\langle 1,\langle 2,3\rangle,3\rangle$ nor $\langle 1,\langle 2,\langle 3\rangle\rangle,3\rangle$.

The "metatyping" of fields can also be of interest. We introduce *flat/type* matching for that case. Here, subtuples must contain the same number of fields and subtuples. We write $\langle\ldots\rangle_{FT}$ for that kind of matching. The template $\langle 1,\langle 2\rangle_F,3\rangle_{FT}$ matches $\langle 1,\langle 2\rangle,3\rangle$ and $\langle\langle 1\rangle,2,3\rangle$ but not $\langle\langle 1\rangle,\langle 2\rangle,3\rangle$. As a further relation of interest, we introduce *flat/value* matching. Here, subtuples are not considered further while fields have to have equal value. We write $\langle\ldots\rangle_{FV}$. The template $\langle 1,\langle 2\rangle_F,3\rangle_{FV}$ matches $\langle 1,\langle 0.0\rangle,3\rangle$ but neither $\langle 1,2,3\rangle$ nor $\langle 0.0,\langle 2\rangle,3\rangle$.

The relations mentioned are ordered, since $D \Rightarrow FV \Rightarrow FT \Rightarrow FS \Rightarrow F$. Further possible relations are currently under study. The differentiated and extensible view on structural matching of nested tuples is one of the contributions of XMLSpaces.NET.

Further matching is possible which combines the relations above. In the current implementation XMLSpaces.NET also supports a matching based on the FV and FT relations. It checks for value- and type-equivalence for fields on the first level of the tupletree, but only for equal numbers of fields and subtuples in any subtuples.

Three cases of field matching have to be distinguished for which different matching relations are defined:

Primitive data can be matched on type- and value equivalence as in Linda. In addition, we foresee matching relations like comparisons ($\langle \geq 5, \leq 3\rangle$).

Objects are matched on type and object equivalence. Object equivalence is defined here by equal representation of a normalized serialization. It is implemented by comparing the respective SOAP serializations of objects.

Type equivalence of objects and its use in matching is an interesting topic and has led to several proposals in tuplespace research ([2, 8, 9] and others). Objects usually are typed and classified. In most object oriented systems, there is a type- and class-hierarchy. With that, two objects can be in several relations – they can be type compatible if their interfaces are in a subtype relation or can be specializations/generalizations if their classes are in a sub-/superclass relation. The hierarchies mentioned form trees. Again, we have a deep and a flat matching. A template can reference a class or a type like $\langle$?AClass$\rangle_F$. For flat matching, an object matching such a field has to be an instance of that class or type like $\langle$aObject$\rangle$. Deep matching here means that matching objects are instances of direct or indirect subclasses or subtypes like $\langle$bObject$\rangle$ if BClass is a subclass of AClass or the interfaces of the objects are in a subtype relation.

XML documents are matched according to some further matching relation since we lack a definition of normalized equivalence of XML documents.

The flexible and extensible matching of values is another contribution of XML-Spaces.NET.

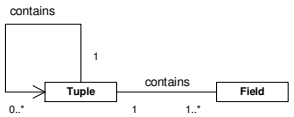
3 ENGINEERING XMLSPACES

In this section we give an overview of the internal structure and architecture of XML-Spaces.NET.

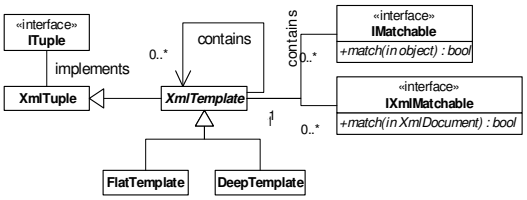
3.1 Local operations

As aforementioned, nested tuples have a tree-structure, therefore it is easy to build a complex nested tuple from the subtuples (subtrees). Fig. 1(a) shows, that two classes with appropriate methods and constructors are sufficient to describe nested tuples.

While nested tuples provide structure to what is put into a tuplespace, fields contain the specific data. A field should be capable of storing any type that is valid in a host programming language that uses XMLSpaces.NET. In addition XMLSpaces.NET adds XML-documents as a valid type.



(a) Tuple and Field



(b) Template

Figure 1: Tuples and Templates

After creating tuples and writing them to a tuplespace with an *out*, it is necessary to retrieve them. Linda specifies two retrieval operations, a consuming (*in*) and a non-consuming (*read*) one. To retrieve a tuple from a tuplespace, a template is defined against which a tuple has to match. If the template contains only values it acutally is a tuple. As stated in section 2, one can see Template as a subclass of Tuple and vice versa. For an implementation, however, it is necessary to decide which approach to take. We therefore define Template as a subclass of Tuple, because apart from (actual) fields and tuples, a template can contain templates and formal fields.

At least three groups of types can be stored in a field: primitive types, objects and XML-documents (see section 2.1). In our implementation we can join two groups, primitive types and objects, since they are part of the host programming language C#.

The definined matching-relations on the two remaining groups (types of the host language and XML Documents) are totally different. Types of the host programming language can be checked for their specific type and value, using the programming language operations. The document type of a wellformed XML-document is determined by its structure and its value by the values of the tags, attributes and contained text. An XML-document itself could have a structure and contents that is itself as complex as a complete tuplespace. Matching relations can be defined on different levels of granulation, i.e. an XML-document's structure or even values of a single element or attribute. The most obvious way to define matching relations is by using XPath-expressions. Although XPath already offers a wide variety of matching-relations, many more matching-relations can be imagined, e.g. validation against XML-schema or XQuery.

To keep the creation and maintenance of matching-relations flexible, we have defined two interfaces, which stand for one type of matching-relation each.

With nested tuples, there are at least two different ways of matching (see section 2.2). XMLTemplate is defined as an abstract class, that contains rules for combination of Templates, Tuples, Fields and matching-relations. Any subclass of XMLTemplate can be used interchangeably. By defining a class that extends XMLTemplate it is possible to extend the set of templates. As we have observed in Sec. 2.2, there are many interesting templates for nested tuples that should be realizable via an easy extension-mechanism. The matching-algorithm should be able to decide which template to use at runtime, so new templates are just defined and used in matching without having to change existing code.

3.2 Remote Operation

Any active entity that emits tuples to or retrieves tuples from a TupleSpace is considered to be a client. In order to create and work on a tuplespace, a client needs a TupleSpace object. TupleSpace objects serve as references to tuplespaces on a server. Clients may have many TupleSpace objects, of course. Apart from the traditional Linda-operations (*in*, *out*, *read*, *eval*) a TupleSpace object contains methods to log on or create tuplespaces and manipulate attributes that affect its behavior. Examples of such planned attributes are timeouts, lease-time of objects etc.

The server manages the tuplespaces and the distribution strategies. It has a collection of TupleBuckets, which represent tuplespaces. Any TupleSpace object that a client uses is associated exactly to one TupleBucket. However, many TupleSpace objects may be associated to the same bucket and thereby share the same tuplespace.

We plan to support three types of replication (none, full and partial replication) as described in [10, 11].

In a system where the tuples are not replicated, all servers manage their own tuplespaces only. If a client writes a tuple to a tuplespace that is on the local server, we have the non-distributed case and simply write (*out*) the tuple to the tuplespace. If the target tuplespace is on a remote server, a *Distributor* object forwards the tuple to the server, which manages that tuplespace. An *in* or *read* is executed on the local server first and then performed on remote servers, if the tuple or tuplespace can not be found. This strategy is easy to implement and consumes little resources compared to strategies with replication.

The counterpart to that strategy is the full replication strategy. Every tuple is stored locally and on every remote server. This brings about a lot of communication and organization overhead among the servers, as with every operation all servers have to be notified and their tuplespaces must be changed according to the source server. This strategy offers a high failsafety. The disadvantages, however, are potentially heavy network traffic and a high consumption of resources.

Between these two extremes is the partial replication strategy in order to gain the advantages of both. In a system performing partial replication all servers are regarded as nodes in a rectangular grid. The grid is partitioned into horizontal and vertical stripes, assigning each node to exactly one intersection of stripes. Each horizontal stripe is defined as an *in-set* and each vertical stripe is defined as an *out-set*. Tuplespaces of nodes in *in-sets* must be disjunct, whereas tuplespaces of nodes in *out-sets* are exact copies.

This structure limits all operations to only a subset of servers. All *in* operations are performed on one *in subset* of servers. The advantage for *out* operations is that they are performed on one out-set only. If a tuple is consumed or added, only the nodes in that *out-set* need to be updated.

However, the number of participating servers should be dynamic. This does not affect the non-replication and the full replication strategy, but for the partial replication strategy it is impossible to guarantee a rectangular grid of nodes. To solve this problem simulated nodes were introduced. Whenever the number of nodes is not sufficient to form a rectangular grid, i.e. when new servers want to participate in or leave the distributed tuplespace, the neighbour in the *in-set* of such a "hole" in the grid simulates its presence. As they are members of the same *in-set* they have the same contents.

In addition to these issues, a distributed system performing any kind of replication must guarantee the integrity of its data. Therefore all distributed operations must follow a communication and operation protocol to lock and release tuples and thereby guarantee data integrity.

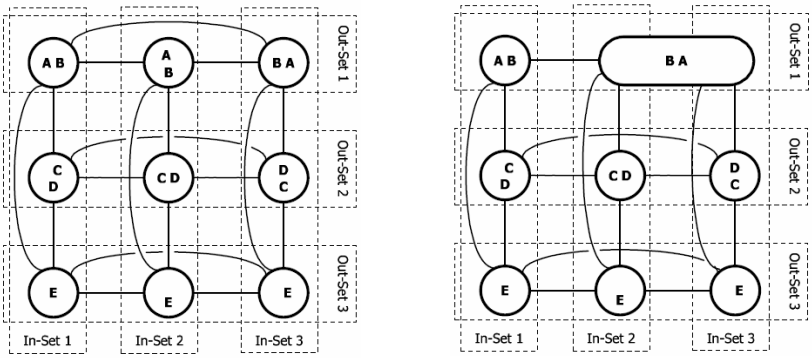
4 IMPLEMENTATION

We use Microsoft's .NET Framework to implement XMLSpaces.NET. It already features functionality we need to implement the Linda-System and the extensions. Languages like VB.NET, C++.NET, Python.NET were extended to work with the .NET Framework. We choose C# as the host language, as it is specially developed for the .NET Framework. All languages, however, compile to the Microsoft Intermediate Language (MSIL) and there should be no significant difference in terms of performance.

After XMLSpaces.NET is released, clients can be written in any host language of the .NET Framework, as they are capable of accessing the same assemblies.

4.1 Tuples

Tuples use the built-in .NET type *System.Xml.XmlDocument* to represent their contents. *System.Xml.XmlDocument* is an implementation of the W3C's DOM and DOM2. It is there-



(a) The Grid of Nodes (b) Example of a simulated node

Figure 2: Intermediate Replication of Tuplespaces

fore an in-memory representation of an XML-Document with methods for manipulation. In order to store data into XML, we need a serialization pattern. Pattern in this context means the XML-structure that represents the types. The .NET Framework has a uniform type-system for all host languages, called Common Type System (CTS). Types are named *System.**, where * is any of the types defined in .NET. Depending on the host language, the available types may vary. For example, C# does not support pointers so the Pointer- Types are not available in C# but they exist in C++.NET. XMLSpaces.NET is capable of handling all possible types, as the type-information is extracted during runtime and stored in the XML-document. On the other hand only clients that know of those specific types (written in a host language in which those types are available) will need to retrieve tuples with such fields.

For these primitive types a serialization is found easily, as we only need a string that represents the value. However, a string representing the value is ambiguous, since ”1” might be *System.Int16*, *System.Int32*, *System.Int64*, *System.Char* or a *System.String*. We therefore need to store the value's type in order to deserialize it correctly. The serialization for primitive datatypes is therefore: *<Field type=”System.*”>VALUESTRING</Field>*.

Objects, in this context are instances of classes, arrays or structs (container for structured data in C#). They are serialized differently, of course. We could use *Reflection* to do the serialization to XML manually, but the .NET Framework already features functionality that serializes an object into a SOAP-document ([12]). Any other XML serialization of objects can be used instead, of course. The serialization for primitive datatypes is therefore: *<Field type=”Soap”>SOAPDOCUMENT</Field>*. It is possible to serialize primitive datatypes into SOAP-documents as well, but we have chosen to serialize into the indtroduced form, because the resulting SOAP-document would be much larger and thus takes more time for matching operations and occupies more memory.

XML-Documents do not need to be serialized, as they can already be represented as strings. The third serialization pattern is *<Field type=”XmlDocument”>XMLDOCUMENT</Field>*.

4.2 Templates

As we have stated in section 3.1, we choose Template to extend Tuple with functionality for matching. It is obvious that we only need to make small modifications. Apart from Tuples a Template may contain other Templates and a field can be substituted by a matching-relation. We implement two interfaces, which form the basis for the extensibility of XMLSpaces.NET. Their serialization is as follows: *<Field type=”IMatchable”>SOAPDOCUMENT</Field>* and *<Field type=”IXMLMatchable”>SOAPDOCUMENT </Field>*.

We can determine if an object is an instance of a class that implements one of those interfaces. Thereby we differentiate two more types that are serialized in Templates, *IMatchable* and *IXMLMatchable*. The SOAP-Formatter of the .NET Framework serializes those objects, which produces well-formed XML-documents.

Only in a template, instances of classes with these interfaces have to be handled separately, as they are needed to perform the matching. In a Tuple templates and those objects would be treated like any other object, allowing even instances of matching-relations and templates to be stored in the tuplespace and be exchanged among clients.

So far only matching-relations where investigated. However, we need an extensibility-mechanism for templates, too. It is necessary to store the type of the template in the XML-representation. Any object in C# has a fully qualified name as its type description, e.g. *XMLSpaces.Templates.DeepTemplate*. We extend the XML-representation of a tuple to contain XML-elements, where the type attribute stores the fully qualified name of the template. On one hand the resulting XML-document contains all information that is needed for matching and keeps the core implementation independent from any modifications. On the other hand, there is no limitation to the number of templates.

4.3 Extending Matching Relations

In C# any class or primitive data type is a sub-class of *object*. We therefore define an interface *IMatchable* with a single method *bool matches(object o)*. Any matching operation

on objects, i.e. primitive data types and instances of objects, can be defined using this interface. This concept is much more powerful than the Linda matching, which is either a type-match, or an exact match of value. Our approach allows the definition of finer relations. A string for example, can be matched in many different ways. A few examples are to match the string exactly, by ignoring the case of the letters, by matching on a substring or its conformity to a regular expression. Depending on the use of XMLSpaces.NET, different matching-relations may be preferred.

XML-documents can be matched in a wide variety of ways. There are existing standards such as XPath, XPointer, XSLT and drafts for future standards like XPath2 and XQuery. It is essential that the set of matching relations for XML-documents is at least as extensible as the set for objects and primitive types. We define the interface *IXMLMatchable* for that purpose. It contains a single method *bool matches(XmlDocument doc)*. Any matching-relation for XML Documents that is not part of the basic set released with XMLSpaces.NET can be defined by implementing this interface. If future development of the .NET Framework integrates, for example, XQuery (which it currently does not), or an API to an existing XQuery system is available, it will be easy to extend the matching-relations of the basic system with that matching relation.

4.4 Matching

A tuplespace consists of a collection of tuples. Following our concept, a tuplespace is a special form of a nested tuple. It contains only tuples and no fields on the first level. Again, we can represent the whole tuplespace as an XML-document. From a higher level a tuplespace can be considered as a tuple of an other tuplespace. This makes it possible to store whole tuplespaces in another and retrieve it at a later time as if it were a tuple.

Matching in XMLSpaces.NET (as in Linda) occurs only on *in* and *read* operations. All arguments passed to them are regarded as templates. Even if a tuple is passed to these methods, a DeepTemplate is wrapped around it to perform an *actual* match. As a tuplespace is an XML-document, we can use XPath, which is implemented in the .NET Framework, to perform a preselection (number of fields and subtuples) of potentially matching tuples. The server then checks if a tuple of that preselected set matches on a given template.

A client requests a tuple by calling *in* or *read* on the TupleSpace object. The call is delegated to the server, which does the preselection on the TupleBucket and performs the match on the collection of potential matches. The first matching tuple is returned to the TupleSpace object and deleted from the TupleBucket. The other tuples are left untouched. The TupleSpace returns either the retrieved tuple to the client, or a null-reference.

The template determines to which depth (DeepTemplate, FlatTemplate, etc.) a tuple is checked and how exact the Fields of the tuple are examined. As stated in Sec. 2.2 there are many interesting types of templates that match a tuple on a very high level (FlatTemplate), where only the metatypes of fields and subtuples are checked, or on a very low level (DeepTemplate), where a template has to match exactly on the tuple. The matching-algorithm traverses the DOM-tree of the XML-document and compares the nodes. Depending on the template the fields and depth are checked differently, so the algorithm has to determine whether there are any nested templates and switch to the algorithm of the nested template.

Whenever an IMatchable or IXMLMatchable object is found in a template, it is deserialized and the *matches()* method is called with the required parameter, i.e. *System.object* for *IMatchable* and *System.Xml.XmlDocument* for *IXMLMatchable*. If any field does not match or any IMatchable or IXMLMatchable object returns false, the algorithm terminates.

Every match operation performs following actions: a) preselect a set of matching tuples on the bucket based on their number of fields and subtuples, b) perform the match method of the template on each tuple in the set of potential matches.

Using the number of fields and tuples we can also decide early whether to continue matching on deeper levels of an XML-document or not. This information limits the matching times on nested tuples as the number of fields and tuples can be checked on any subtupletree.

4.5 Distribution

The .NET's *Remoting Framework* is used to implement the client-server architecture. Using a directory service, such as Microsoft's *Active Directory* [5] or *OpenLDAP* [7], allows a dynamic configuration of the participating servers and the replication mode, i.e. switching the replication mode of all servers during runtime.

However, on a campus network *Active Directory* is not always flexible enough, as the *schema* of the directory has to be modified to meet the needs of XMLSpaces.NET. The schema change might require administrative rights not available to an end-user. OpenLDAP is an alternative in this case. We decided to stay as independent as possible of those technical problems and have implemented an extra class to maintain the server list.

The distributed system differs from the non-distributed one in the use of the buckets. While in a non-distributed system the server directly calls methods on its local buckets, a distributor object manages the calls to the local buckets and the remote buckets. The implementation of the distributed system profits from the XML structure of the TupleBucket. If each tuple is assigned a unique identifier pointing to its source location, a TupleBucket is able to group those tuples in an XML subtree associated to that remote source. This is beneficial for the implementation of the replication as it is easy to sort out tuples of different servers, since all tuples with the same source server are under the same subtree. For an *out* operation the Distributor inspects all servers in the server list for their replication mode, adds the identifier to the tuple and sends it to appropiate target servers, depending on the replication strategy, where they are stored to a TupleBucket's subtree according to its identifier.

For an *in* operation the identifier is ignored and the search includes all tuples in the tuplespace. The removal of a match is easy, as the tuple's identifier points to the correct subtree in each TupleBucket, in which the tuple can be found and therefore speeds up the operation. The XPath API allows a fast search on the XML structure of the TupleBucket using

the tuple's identifier.

In case the replication mode changes, or a server deregisters from the server list, the whole contents of the server's tuplespace can be easily removed by deleting the subtree representing that server's replicated tuplespace. If the replication starts up, the contents of a TupleBucket can be added as a subtree to a remote server's TupleBucket.

For locking a tuple we add a boolean attribute "locked" to the tuple's XML root element. If an operation is being performed on the tuple the attribute has to be set to "true" and else "false". The .NET Framework's native support for XPath queries and the DOM2 make this approach easy.

5 PERFORMANCE

We ran several performance tests on our system, a 2.40 GHz Pentium 4 with 512MB RAM running Microsoft Windows XP Pro and the Microsoft .NET Framework 1.1. As there are many dependencies in the XMLSpaces.NET system, we decided to explore the performance along the following dimensions: 1) type of tuple, i.e. tuples containing primitive data types, objects, or XML documents 2) number of tuples in the tuple-bucket 3) number of potentially matching tuples in the bucket, i.e. tuples that have equal tuplecount and fieldcount as the template or tuple we want to match against

For the implementation of the performance test we designed some reference tuples, which contained 5 fields with primitive data or 5 fields with an object each or 5 fields with an XML document each.

The tests were ran on buckets of size 500, 1000 and 2000. At the beginning of each test the corresponding number of tuples is randomly generated to fill the bucket. The randomly generated tuples built from template fields to make sure they have a determinable form. The tuples only vary in the number of fields and their depth. At this point we assumed two different probabilites on matching tuples: In one experiment, we assumed that 25% of the tuples in the bucket are potential matches, in the other, we assumed 50% of potential matches.

No templates were used to retrieve tuples, as we intended to measure the time taken for an exact match of tuples. Owing to the recursive matching algorithm any match against a template (using matching relations) is usually faster since a template match only compares a fragment of information an exact match does.

Using this testbed we had the system play "ping pong" for each of the above defined type of tuples and got the results shown in Figure 3. Two clients play "ping pong" when each has a tuple the other is waiting for, i.e. one client writes its tuple to the tuplespace and waits for the tuple of the other client. The other client starts by waiting for the tuple and writes its own tuple only after having received the other client's tuple etc.

The observations can be explained easily. A match took longer the more potential matches were in the bucket, as the algorithm tries to match against any of the potentially matching tuples. In the worst case it is the last tuple (or none) that matches the template-tuple.

Apart from the number of potential matches the time elapsed for a match depends upon its type. As explained earlier the serialization pattern for primitive types is relatively compact and, except the type "string", cannot be very long. It is therefore easy to see that this type of matching is the fastest. As objects are serialized to SOAP-format XML documents they should be matched in approximately the same time as equally large XML documents. However, all objects that are represented in the SOAP - format have a large root element in common, which identifies the SOAP version and Common Language Runtime (CLR) the system is running on. If many potentially matching tuples with objects are in the bucket the overhead for comparing that root element is relatively high. One possible optimization is to skip the header and compare only the body of the SOAP envelope. The consequenc is, however, that objects of systems running different CLR are identified as the same object, even though they represent different ones.

The XML representation gives us some advantages. Using the attributes *tuplecount* and *fieldcount* we can make a preselection with XPath. As in our two test scenarios there are 50% or 25% of potentially matching tuples in the tuple bucket, the preselection speeds the matching algorithm up by the maximum factor of two or four.

Currently the matching algorithm is very simple and compares each node in the XML representation of the tuple to the template or the template-tuple. Performance improvement might be achieved if the matching algorithm was to apply the preselection to each subnode. Additionally one can think of an extended preselection that uses the value of the current node. The result could be an even smaller range of potentially matching tuples and a faster matching algorithm.

Of course the performance improvement that is possible depends heavily on the implementation of the XPath API. With an efficient implementation, though, one can still expect further improvements.

6 RELATED WORK

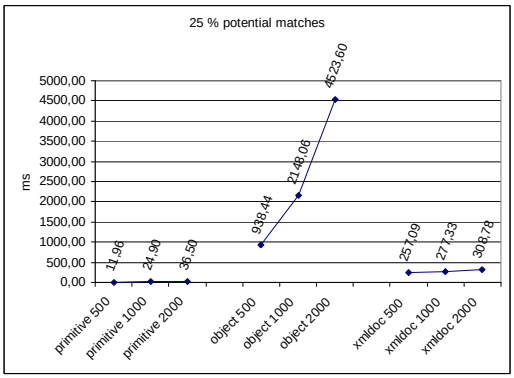
There are several projects documented on extending Linda-like systems with XML documents. However, XMLSpaces seems to be unique in its support for multiple matching relations and its extensibility.

MARS-X [1] is an implementation of an extended JavaSpaces [4] interface. Tuples are represented as Java-objects where instance variables correspond to tuple fields. Such an tuple-object can be externally represented as an element within an XML document. Its representation has to validate towards a tuple-specific DTD. MARS-X closely relates tuples and Java objects and does not look at arbitrary relations amongst XML documents.

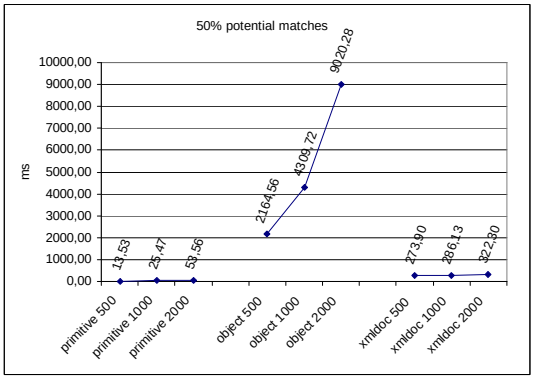
XSet [14] is an XML database which also incorporates a special matching relation amongst XML documents. Here, queries are XML documents themselves and match any other XML document whose tag structure is a strict superset of that of the query. It should be simple to extend XMLSpaces with this engine.

[6] describes a preversion for an "XML-Spaces". However, it provides merely an XML based encoding of tuples and Linda-operations with no significant extension. Apparently, the proposed project was never finished.

TSpaces has some XML support built in [13]. Here, tuple fields can contain XML documents which are DOM-



(a) 25% potential matches



(b) 50% potential matches

Figure 3: Performance

objects generated from strings. The *scan*-operation provided by TSpaces can take an XQL query and returns all tuples that contain a field with an XML document in which one or more nodes match the XQL query. This ignores the field structure and does not follow the original Linda definition of the matching relation. Also, there is no flexibility for further relations on XML documents.

7 SUMMARY AND OUTLOOK

With the XMLSpaces.NET conception we have developed a very extensible XML-based middleware. The further work is on finalizing the set of supported matching relations. The challenge here is to find a set of practically useful relations amongst the wide variety of possible combinations. Also, comparisons like $\langle \geq 5, \leq 3 \rangle$ have to be carefully limited not to deadlock the selection of matches.

As mentioned in the beginning, the XMLSpaces.NET project consists of two parts. The XMLSpaces.NET kernel in C# and the distribution of the kernel itself by applying mechanisms like replication etc. Part of the research on distribution will be to explore possibilities to support detachment of parts of a tuplespace for transportation and manipulation by mobile devices.

Furthermore, we will explore to what extend we can easily incorporated further functionalities like secure spaces by the adoption of the respective XML technologies. We hope that such extensions are quite seamless.

In conclusion, XMLSpaces.NET is a flexible XML-based middleware founded on the tuplespace principles. The main contributions are the integrated view on data, documents and objects, the support for structural matching, the extensibility and flexibility of match mechanisms and consequent usage of XML technologies.

Acknowledgment XMLSpaces.NET is funded under contract 2003-144 by Microsoft Research Cambridge.

REFERENCES

[1] G. Cabri, L. Leonardi, and F. Zambonelli. XML Dataspaces for Mobile Agent Coordination. In *15th ACM Symposium on Applied Computing*, pages 181–188. ACM Press, 2000.

[2] C. J. Callsen, I. Cheng, and P. L. Hagen. The auc c++ linda system. In G. Wilson, editor, *Linda-Like Systems and Their Implementation*, pages 39–73. Edinburgh Parallel Computing Centre, 1991. Technical Report 91-13.

[3] P. Ciancarini, R. Tolksdorf, and F. Zambonelli. Coordination Middleware for XML-centric Applications. *Knowledge Engineering Review*, 17(4), 2003.

[4] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.

[5] Mircosoft. Active Directory, 2004. *http://www.microsoft.com/windows2000/technologies/directory/ad/defau% lt.asp.*

[6] D. Moffat. XML-Tuples and XML-Spaces, V0.7. http://uncled.oit.unc.edu/XML/XMLSpaces.html, last seen May 6, 2002, Mar 1999.

[7] OpenLDAP Community. Openldap. Website, 2004. http://www.openldap.org.

[8] A. Polze. The object space approach: decoupled communication in c++. In *Proceedings of TOOLS USA'93*, pages 195–204, 1993.

[9] R. Tolksdorf. Laura: A coordination language for open distributed systems. In *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems ICDCS 93*, pages 39–46, 1993.

[10] R. Tolksdorf and D. Glaubitz. Coordinating Web-based Systems with Documents in XMLSpaces. In *Proceedings of the Sixth IFCIS International Conference on Cooperative Information Systems (CoopIS 2001)*, number LNCS 2172, pages 356–370. Springer Verlag, 2001.

[11] R. Tolksdorf and D. Glaubitz. XMLSpaces for Coordination in Web-based Systems. In *Proceedings of the Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises WET ICE 2001*. IEEE Computer Society, Press, 2001.

[12] World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1. W3C note for public discussion, 2000. http://www.w3.org/TR/SOAP/.

[13] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[14] B. Y. Zhao and A. Joseph. The XSet XML Search Engine and XBench XML Query Benchmark. Technical Report UCB/CSD-00-1112, Computer Science Division (EECS), University of California, Berkeley, 2000. September.

Design and implementation of a FIPA compliant Agent Platform in .NET

Miguel Contreras Ernesto Germán Manuel Chi Leonid Sheremetov
Instituto Mexicano del Petróleo
Eje Central Lázaro Cárdenas #152
Col. San Bartolo Atepehuacán.
México, D.F. 07730, México
mcontrer{egerman, machi, sher}@imp.mx

ABSTRACT

The aim of this paper is to describe the design and implementation of an agent platform called CAPNET (Component Agent Platform based on .NET) that is fully compliant with the specifications of the Foundation for Intelligent Physical Agents (FIPA) and implemented as 100% managed code in the .NET framework.

Keywords

Distributed Computing, Multi-Agent Systems , FIPA, Agent Platform, .NET Framework.

1. INTRODUCTION

Agent-based computing has the potential to significantly improve the theory and the practice of modeling, designing, and implementing complex distributed computer systems [Jen00, Woo99]. Autonomous agents are entities that can complete their objectives while situated in a dynamic and uncertain environment, that can engage in rich, high-level social interactions, and that can operate within flexible organizational structures and systems.

Agent-based software should be robust, scalable and secure. To achieve this, the development of open, stable, scalable and reliable architectures that allow compliant agents to discover each other, communicate and offer services to one another is required. These architectures go beyond the capabilities of the typical distributed object oriented programming techniques and tools. The FIPA's Agent Platform (AP) reference model seems to be an effective approach to address this problem [Fip00].

An AP is a software architecture that controls and manages an agent community allowing the survival and mobility of an agent in a distributed and heterogeneous environment.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2004 *workshop proceedings,*
ISBN 80-903100-4-4
Copyright UNION Agency – Science Press, Plzen, Czech Republic

In the last few years, several APs have been developed, and special attention has been paid to interoperability and compatibility issues. In this sense, the FIPA reference model has emerged as a standard for interoperability sustaining the development of APs. Most of the FIPA compliant APs were developed in Java language, such as JADE [Jad00], FIPA-OS [Nor99] and Zeus [Zeu00] just to mention a few of them. One exception to this trend was the original CAP [She01] agent platform implemented using Microsoft DCOM and ActiveX technology. The purpose of CAP was to enable the construction and operation of multi-agent systems (MAS) using Windows programming languages and platform.

Our recent work has lead to the development of a completely new AP, named CAPNET under the novel Microsoft .NET Framework and Compact Framework in 100% managed code written entirely in the C# language. This new agent platform uses a great number of the technologies available in .NET and the windows platform, such as Web Services (WS), remoting, asynchronous callbacks, delegates, XML, database connectivity, performance counters, event log, Winforms, Windows Services and network access, among others.

The main objective of CAPNET is to bring an integrated infrastructure that covers the programming, deployment, administration and integration with legacy applications of MAS (Fig. 1). It consists of a run-time environment that supports

MAS deployment, development environment in the form of agent templates, programming tools and a component gallery and some connectors to enable the integration with enterprise applications. The run-time environment is described in this paper focused on the design and technical details of its implementation on the .NET Framework.

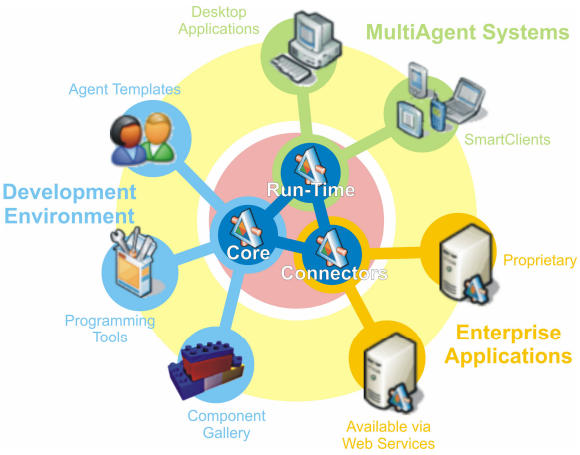


Figure 1. MAS development and deployment in CAPNET

This paper is structured as follows: in section 2, basic concepts of agents and multi-agent systems are presented along with a short description of the FIPA specifications. In section 3, the CAPNET architecture is defined. In section 4, the most important implementation details are addressed and technically described. Finally, the main features and advantages of this work are discussed, along with some future work.

2. AUTONOMOUS AGENTS

An agent is a computational entity that interacts with one or more software counterparts or real-world systems [Fra96]. Unlike traditional computer programs, agents exhibit the following capabilities to various degrees: autonomy, reactiveness, proactive-ness, mobility, intelligent behavior and social abilities.

The autonomy and pro-activeness features of an agent allow it to plan and perform tasks defined to accomplish the design objectives. The social abilities enable an agent to interact in MAS and cooperate or compete to fulfill its goals. An agent may be static or mobile. In the latter case it is able to migrate along with its associated data, state, and logic to another host to interact with local resources and other agents to perform a given task.

The open nature of the MAS is provided by the agent organization, similar to that of distributed enterprises, and supported in the agent platform's tools, which are responsible to provide flexibility both in component aggregation and interaction between them [She03]. The AP reference model of the FIPA provides the framework of normative work, inside which the agents exist and operate; it also establishes the logical and temporal contexts for the creation, operation and destruction of agents [Fip00]. The reference model considers an AP as a set of four components: Agents, Directory Facilitator (DF), Agent Management System (AMS), and Message Transport System (MTS). The DF and AMS are special types of agents that support the management of other agents, while the MTS provides a message delivery service (fig. 2).

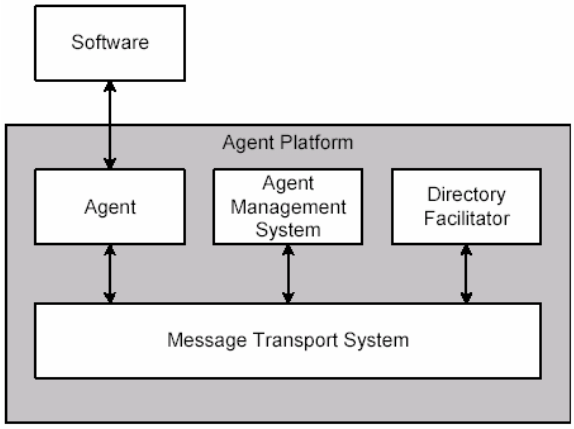


Figure 2. Agent Management Reference Model [Fip00].

The functionality of the main components of the FIPA spec are: the AMS provides white-page and life-cycle service, maintaining a directory of agent identifiers (AID) and agent state. The DF is the agent who provides the default yellow page service. The MTS is the software component controlling all the message exchange within the platform, including messages to/from remote platforms.

The agents are the main parts of a platform. An agent encapsulates one or more services inside a unified and integrated execution model. FIPA maintains an open concept of what an agent is, to be able to include a great number of agent applications, and not limit the form in which they are implemented. The Software refers to all those systems that do not have characteristics of agents but that are used by agents to fulfill their tasks. Here, domain specific systems, as well as the Application Programming Interfaces (API) for handling communication protocols, databases, security algorithms, etc. are included.

3. CAPNET ARCHITECTURE

The main objectives of CAPNET design are to construct a platform that enables developers to easily

create and integrate distributed agent applications in a consistent and scalable way. The Platform should be able to interoperate with applications developed in other APs as well.

The architecture of CAPNET is shown in fig. 3. In this architecture four main parts are shown: i) application agents, ii) agent management and directory facilitator services, iii) built-in security services, and iv) message transport system and connectivity techniques. The message transport mechanisms are the core of the platform, supporting a wide variety of transport types. Among the considered transport managers are all those required to ensure platform interoperability with other widely available APs such as those mentioned in section 1.

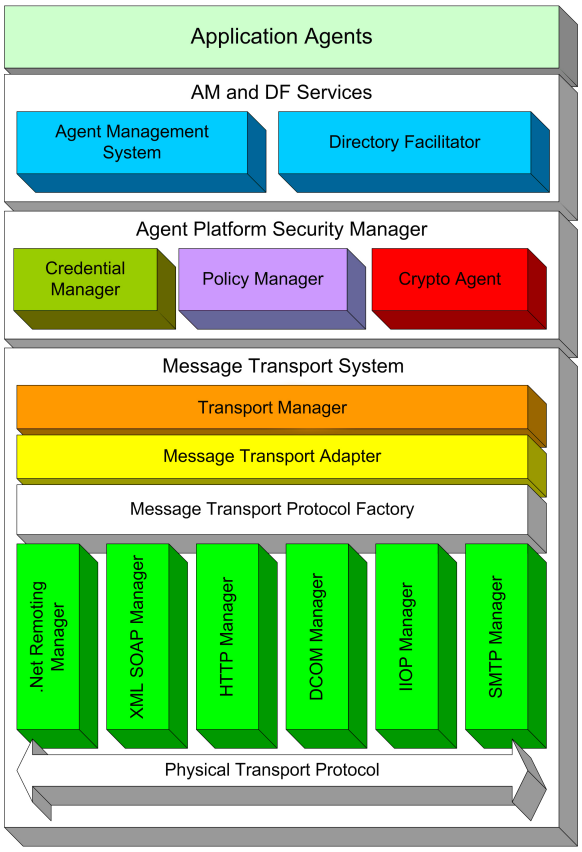


Figure 3. CAPNET architecture

The services of transport, delivery and reception of messages represent a central point within the CAPNET platform. In CAPNET, an agent communicates with others using a service provided by the MTS. This level of abstraction allows developers of multi-agent systems to design them based on schemes of loosely coupled messaging systems between components. With this type of asynchronous messaging the communication can be seen as a distributed messaging system similar to a Message Oriented Middleware (MOM) [Ber96] or publish/subscribe architectures [Pal03].

To assure MTS reliability, some additional mechanisms are implemented. When an agent sends a message, it knows if the MTS has delivered it to the destination agents or not. If some adressee could not be contacted then the emitting agent receives a notification indicating that the message could not be delivered. It is the responsibility of the emitting agent to maintain its state or to block its execution while it waits for the answer.

Two cases for message delivery exist. The first one takes place when the addressee agent is accessible through the same MTS and the internal mechanism of the MTS is used. In the second case, the message addressee is registered in other MTS. In this case, a component called Message Transport Adapter (MTA) has been implemented to determine the type of message transport that is needed to complete the operation and to use the appropriate technological infrastructure for it. The MTA uses the services of Message Transport Protocol Factory (MTPF) to instantiate the component that really implements the access to the physical transport. This component in our architecture is called Transport Manager (TM).

If a new transport mechanism is required, a new TM component has to be created implementing the standard interface known by the MTPF and the MTA. If that interface is implemented, the interaction with the MTPF is assured and the particular implementation of the transport mechanism is completely open to the developer, and can include any form of communication both synchronous and asynchronous, such as raw sockets, MSMQ, FTP, SMTP, etc. The TM has to be registered in the platform in order to be used. At the time of writing the TMs for HTTP and .NET Remoting have been implemented, and several more are in process.

In a heterogeneous multi-agent environment, security becomes an extremely sensitive issue. Security risks exist throughout the whole agent life-cycle: agent management, registration, execution, agent-to-agent communication and user-agent interaction. Agent Platform Security Manager is out of the scope of this paper, the details on its architecture and implementation can be found in [San03].

4. IMPLEMENTATION DETAILS

4.1 The Message Transport Mechanism

The MTS was implemented as a singleton remoteable object [Ram01] that can be instantiated by the agents in order to be able to send and receive messages. This remote object is able to deliver messages to the agents based on delegated method and publish/subscribe mechanisms for events.

The mechanism for message delivery is a type of asynchronous callback allowing messages to be delivered as they are received by the MTS. The agents use a special class working like a listener of incoming messages. This listener contains a delegate method which is invoked when the MTS receives a message.

When agents are initialized they subscribe to the MTS sending a listener object. This object is registered in the events manager administered by the MTS. When a message has to be delivered by the MTS, it consults events manager to look for the listener of the destination agent of the message. This way it can invoke the delegated method that was registered for the listener of the agent.

If an agent wishes to send a message (fig. 4), it contacts its MTS. When a message is received by the MTS, it is processed as specified by FIPA. When it is prepared for delivery to the destination agent(s), if any of them is located in a different CAPNET, MTS contacts the MTA to delegate the delivery. The MTA determines the required type of delivery based on the destination agent's address. Using this information the MTA obtains a concrete TM from the MTPF. That TM is the object capable of performing the real delivery of the message and once obtained, the MTA invokes the delivery functionality on it.

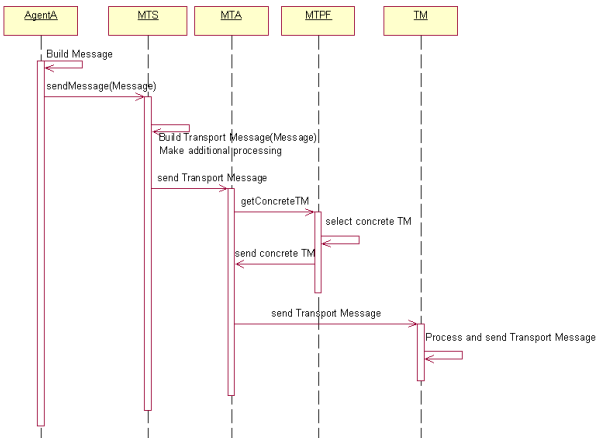


Figure 4. Sequence for message sending.

When a message is received by a TM, it extracts the important fields and constructs a new platform specific Transport Message object to be sent to the known MTA. This MTA forwards the Transport Message to the MTS, which sends the message to the destination agent. Message receiving is similar.

4.2 The AMS and DF Services

The AMS and DF services of the platform have been implemented using a multi-tier architecture (Fig. 5). We shall only describe the AMS Service, since the implementation of the DF is almost identical.

The agent tier (Tier 1 in the figure) is implemented by the functionality provided by the BasicAgent class (described in the next section), and involves all the communication, interaction and conversation mechanisms that provide the social interaction capabilities. This layer is responsible for listening to other agents requests, implemented as an ACL (Agent Communication Language) *request* message specifying the action and its parameters, and launching the corresponding services.

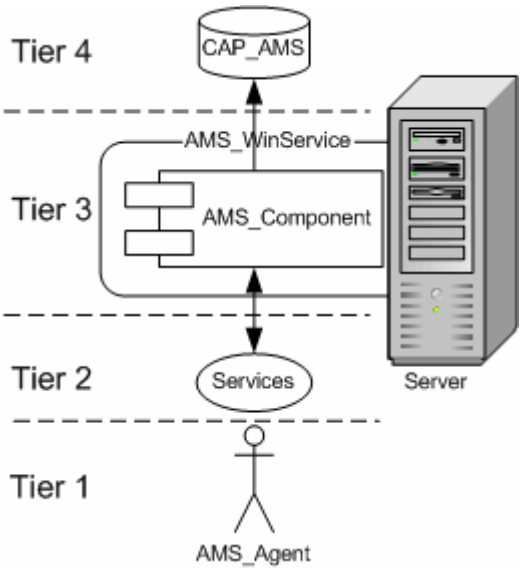


Figure 5. AMS Service implementation.

The application-logic tier (Tier 2) is implemented in the services (actions) that the AMS_Agent is able to perform. The actions that this agent is able to perform are: registration, deregistration and modification of agent descriptions in the white page directory, and search of agent descriptions.

The data access tier (Tier 3) is implemented in the AMS_Component which is a remoteable component hosted in a Windows Service. This component designed as a "singleton" exposes two interfaces: the IAMS interface for the AMS_Agent and the IAMSAdmin interface for the administration and monitoring applications, such as the Society Viewer (that shows the registered agents and their interactions) or the AMS_Administrator among others.

The AMS_Component registers and updates constantly a set of performance counters. These counters allow system administrators to monitor and record the state of the platform and to raise events or generate alarms under certain conditions they are interested in.

The data tier (Tier 4) is implemented in a relational database using Microsoft SQL Server as the DBMS that stores Agent Descriptions of all the registered agents.

4.3 Agent Implementation

The construction of a set of basic elements that constitute the internal architecture of the agents in a FIPA compliant environment is crucial for application development. These elements provide the ways to develop agents, their unique identifiers, registry information, ACL message construction, message reception and handling, content codification and representation with different content languages, the platform services description and application agents services. In the following sections these elements are described in more details.

4.3.1 Basic Agent

In order to be successful or, at least, easy to use, the AP has to provide some mechanism for agents creation. Being the central element of the applications, a basic agent class takes advantage of the entire infrastructure in a transparent way. This class allows carrying out several tasks like i) instantiating the local MTS and registering its listener in it for incoming messages reception, ii) registering of AMS and DF services, and iii) processing of received messages. AMS and DF registration have been implemented using the conversation mechanism designed to control the messages exchange. This mechanism is described in the following section.

In order to support agents programming for the AP, an API is provided. This API includes classes to construct the unique agent identifier (AID), DF service descriptions (ServiceDescription), the local platform description (APDescription), AMS agent descriptions (AMSAgentDescription), DF agent descriptions (DFAgentDescription), etc. Figure 6 (in appendix) shows the diagram of CAPNET utility classes.

Two mechanisms for message processing are developed: polling and callback. Polling is the mechanism that allows an agent to process messages in a synchronous way and is used by the conversations mechanism to control the predefined message sequence in an interaction protocol. Callback works similar to the MTS message delivery mechanism. Events are declared in each agent to process each one of the message types or a pre-established conversation. The particular mechanism to be used is dynamically determined according to the attributes of the message (conversation-id and protocol).

4.3.2 Conversation Manager

A conversation manager is an internal component of each agent linked to its communication capacity. Conversations are important to facilitate the interaction between different agents to carry out some tasks within a multi-agent environment. Besides facilitating the interaction, a conversation manager allows an agent to control its course of action on the basis of the results that are obtained during their conversations. A conversation manager allows an agent i) to add new conversations required during the communication process, ii) to add agent interaction protocols (AIP) that can be used to control the message sequence in a conversation, and iii) in general, to offer access to the completion state and results of a conversation.

A conversation is added when a message establishes a conversation and AIP identifiers. A new conversation is dynamically created determining (by means of the .NET reflection mechanism) the AIP from the CAPNET library. It is important to mention that an AIP can have several implementations. Each new conversation is handled in a new execution thread in such a way that an agent can carry out parallel interaction through simultaneous conversations.

We have defined a set of classes and interfaces that helps to create conversations and AIPs in a standard way. At the moment, two interfaces for the synchronous and asynchronous conversations are implemented. When a new interface is created, it must inherit from a conversation class to establish its attributes (conversation-identifier, AIP class to be used in conversation control and delay time-out between messages) and must implement some of the interfaces of conversation type, that mainly serve to give access in run time to an AIP's concrete implementation.

4.3.3 Agent Communication and Content Languages

In order to provide communication functionality, FIPA-ACL is implemented [Fip00]. XML is used as the standard encoding for messages. The content of the ACL messages is represented in a content language allowing agents to obtain and handle objects from the agent's knowledge base. It enables knowledge interchange and handling between heterogeneous applications and guarantees high interoperability and autonomy degrees. For CAPNET message coding two languages are implemented: FIPA-RDF0 (using XML representation with validation through schemes) and FIPA-SL (using the string representation scheme, a grammar and parser for construction and validation of these content objects).

4.3.4 Dynamic action invocation

Agents can carry out actions in favor of others. Because the action requests are codified in a sort of text format and are not obtained directly through method invocation, agents extract the requested action and achieve dynamic invocation by means of .NET reflection mechanism. The content language allows expressing an action, its internal results and its arguments. The agent itself determines how to extract the action and its attributes to carry out the invocation.

4.4 Administration Tools

Along with the platform, a set of tools for its configuration and administration have been developed. These tools include a society viewer, a ping agent, AMS and DF administrator, Pocket PC version of the AMS and DF administrator, platform's communication infrastructure configurator, etc.

One of the most important tools is the AMS administrator (fig. 7), because it allows monitoring of agents registered at the AP, their state and properties.

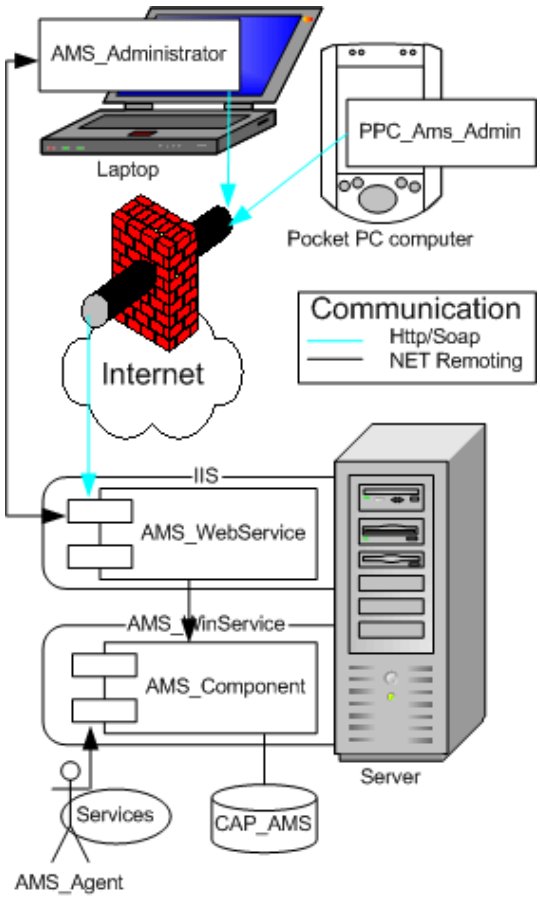


Figure 7. AMS Administration architecture.

This tool is designed to connect directly to the AMS_Componet using the IAMSAdmin interface via .NET Remoting. When communication using remoting is not possible (because of network restrictions or using the Pocket PC version of the administrator) a Web Service is used to form a bridge to the AMS_Component. The latter approach however has a drawback: the administrator has to request constantly the last state of the AMS, whereas using remoting the Administrator is able to subscribe to the published events of the AMS_component, in order to receive instant notifications of any changes occuring in the AMS as a result of registration, deregistration or modification of agent descriptions in the AMS.

5. DISCUSSION AND CONCLUSIONS

In this paper we have presented an Agent Platform named CAPNET that constitutes an excellent example of a distributed system built on top of the .NET Framework. One particular feature of this platform is that its primary goal is to enable the developers to construct another kind of distributed, flexible and open systems (MAS) using it as the basic infrastructure for communication and administration of the elements that will be part of them.

The architecture proposed for the core of the CAPNET is divided into three main blocks: directory services, security and message transport services. Along with those elements a set of tools for the administration, monitoring and development of MAS for the platform have been constructed.

Since agent communication plays a key role in social interaction, a great effort was invested in order to make it very extensible and interoperable. To achieve this, the low level transport mechanisms (TM) were isolated from the core of the MTS and integrated into it using the "factory pattern" that enables the possibility to have several concrete implementations of TMs for different protocols or communication techniques. Another advantage of using this design pattern is that it will easily accept the implementation of load balancing techniques in the future.

The implementation of the CAPNET required the extensive use of remoting for different tasks such as agent communication and administration. The use of remote delegates allowed the agents to subscribe to the events published by the MTS, and also enabled the administrative tools to get instant notifications of the changes in the state of the platform services.

All the remoteable AP components involved in the message transport mechanisms (MTS and Remoting TM) and directory services (AMS and DF) were implemented as Server Activated Singleton Objects and hosted in Windows Services.

This particular implementation has the advantage of high availability of these components and will eventually lead to the clusterization of these services to increase the reliability of the AP in future versions.

The use of XML as the standard encoding for messages has several advantages. Some of them are: native support on the .NET framework for managing XML documents, easy integration with the modern commercial and industrial applications available and the natural integration to the semantic languages.

In order to take advantage of the features of the development platform, the CAPNET services report their state to the operating system via performance counters and the event log.

In order to make CAPNET compatible with the .NET Compact Framework (CF) and to be able to deploy administration tools and agent systems in mobile devices, alternative mechanisms to remoting for communications had to be implemented. These mechanisms included the use of Web Services, easily accessible for applications written for the CF and capable of bridging to the main CAPNET infrastructure.

The communication capacities of the platform were stress tested with a custom made benchmark application that involved the creation of 100 agents distributed in 10 hosts in a single segment of a 10Mbps LAN. The benchmark measured the time it took for each agent to send one message to each other agent, and one to itself (sending a total of 10,000) XML encoded messages with a length of 300 bytes/each. The results obtained showed that the full load of 10,000 messages took a variable time of 83 to 108 seconds to be delivered in 40 different simulations at different times (the network infrastructure was not exclusive for this purpose and had peek use hours).

A very important feature to be implemented in future versions of CAPNET, will be the support for agent mobility, that will enable the agents to traverse hosts to perform their tasks.

Several prototype MAS are under development using CAPNET, which will help to test its functionality for concrete applications. Most of these prototypes were earlier implemented using JADE or the first version of the CAP and include supply chain configuration, secure desktop, contingency management (fig. 8). Details can be found in [She04, Smi04].

6. ACKNOWLEDGMENTS

Support for this research work has been provided by the Mexican Petroleum Institute within the project D.00006 "Distributed Intelligent Computing".

The authors would like to thank Microsoft™ México, especially Luis Daniel Soto and Felipe Lemaitre for their support for the realization of this work and also to all their colleagues for the helpful advices on the CAPNET architecture and Ana Luisa Hernandez for her contribution to the development of the CAPNET message encoding mechanism.

7. REFERENCES

[Ber96] Bernstein, Philip A. "Middleware: A Model for Distributed Services." Communications of the ACM 39, 2 (February 1996): 86-97.

[Fip00] FIPA Specifications. http://www.fipa.org/specs/

[Fra96] Franklin, S. and Graesser, A., “Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents”, Proc. ATAL’96, Springer-Verlag, Berlin, Germany, 1996.

[Jad00] JADE Programmer’s guide, Bellifemine F., Caire G., Trucco T., Rimassa G. JADE 2.5.

[Jen00] Jennings N. R. On agent-based software engineering. Artificial Intelligence, 117, 277–296.

[Nor99] Nortel Networks FIPA-OS http://www.nortelnetworks.com/products/cements/

[Pal03] Pallickara, S. Fox J., Yin J., Gunduz G., Liu H., Uyar A., Varank M. A Transport Framework for Distributed Brokering Systems. Proc. PDPTA'03.Volume II pp 772-778.

[Ram01] Rammer I. Advanced .NET Remoting, 1st edition, Apress, ISBN: 1-59059-025-2

[San03] Santana G., Sheremetov B. L., and Contreras M. Agent Platform Security Architecture, Computer Network Security **(***Proceedings of the MMM-ACNS 2003, St. Petersburg, Russia, September 2003)*, V. Gorodetsky, L. Popyack, V. Skormin (Eds.), LNCS 2776, Springer Verlag, 2003, pp.457-460

[She01] Sheremetov, L. & Contreras, M. Component Agent Platform. *In Proc of the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems, CEEMAS'01*, Cracow, Poland, 26-29 of September, 2001, pp. 395-402.

[She03] Sheremetov L., Contreras M., Chi M., Germán E., Alvarado M. Towards the Enterprises Information Infrastructure Based on Components and Agent. In Proc. of the 5th International Conference On Enterprise Information Systems, Angers, France, 23-26 April, O. Camp, J. Filipe, S. Hammoudi and M. Piattini (Eds.) Escola Superior de Tecnologia do Instituto Politécnico de Setúbal, Portugal, 2003. pp. 340-347.

[She04] Sheremetov L., Contreras M. and Valencia C. Intelligent Multi-Agent Support for the Contingency Management System. International Journal of Expert Systems with Applications (Special Issue), Pergamon Press, 26(1): 57-71, 2004.

[Smi04] Smirnov, A. V. Sheremetov, L. B. Chilov, N. Romero-Cortes, J. Soft-computing Technologies for Configuration of Cooperative Supply Chain. *Int. Journal Applied Soft Computing*. Elsevier Science, Vol. 4(1):87-107, 2004.

[Woo99] Wooldridge M. "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence", G. Weiss (Ed.), MIT Press, Cambridge, MA, 1999.

[Zeu00] ZEUS: Nader A., Thompson S. A Toolkit for Building Multi-Agent Systems Proceedings of fifth annual Embracing Complexity conference, Paris April 2000.

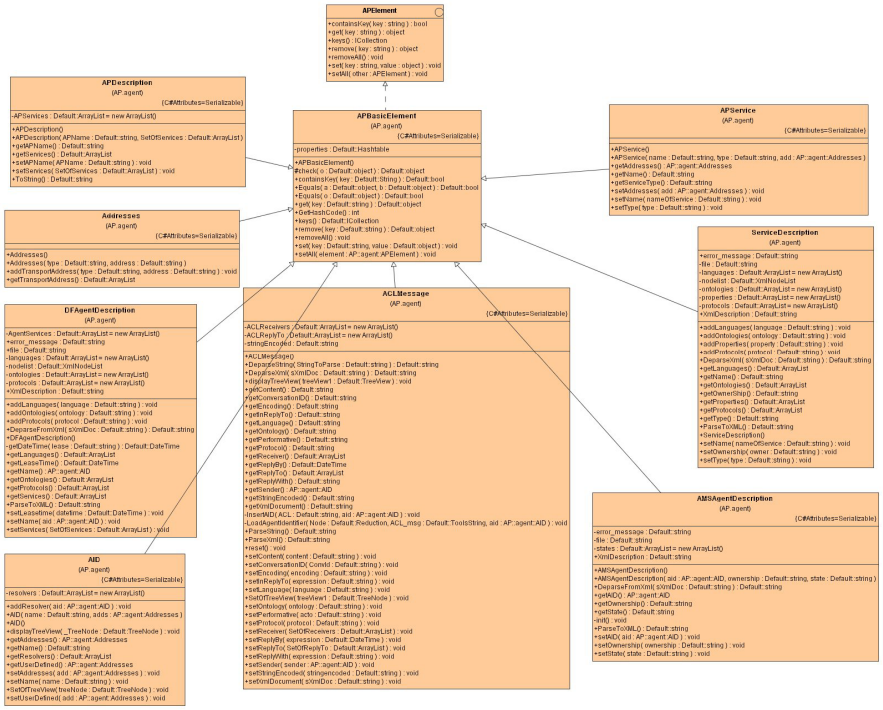


Figure 6. CAPNET Utility Class Diagram.

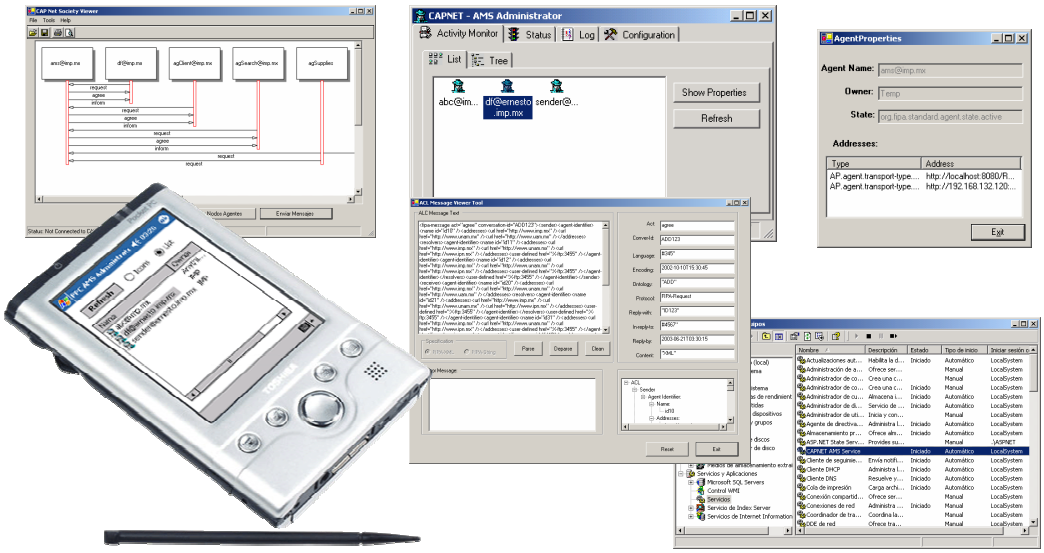


Figure 8. CAPNET Tools (Contingency Management System application).

A Framework Built in .NET for Embedded and Mobile Navigation Systems

Zoltán Benedek
Budapest University of Technology and Economics
Dept. of Automation and Applied Informatics
Goldmann György tér 3, 4.em
Hungary 1111, Budapest
zoltan.benedek@aut.bme.hu

ABSTRACT

With the appearance of low cost high performance embedded and mobile computers the applications running on these computers can offer novel services on board of a wide variety of transportation vehicles. These services include providing navigation aid based on GPS (Global Positioning System) and digital maps in the field of personal in-car navigation. Considering public transportation vehicles the most important services are automatic next-stop annunciation, electronic sign control, automatic passenger counting, dispatch communication and Wireless LAN (Local Area Network) data management. Most of these applications can be decomposed into a common set of components. This paper describes a component-based framework developed in .NET to support the development of navigational applications. The core of the framework is the component configuration, component wiring and communication infrastructure which facilitates the low coupling of components and also enables the tight integration of services. Utilizing this infrastructure and building a predefined set of component building blocks the development time and cost of specific applications can be reduced significantly.

Keywords

navigation system framework, component framework, embedded event-driven applications

1. INTRODUCTION

Recently a boom in the market of general purpose personal mobile devices (Pocket PCs, Smartphones, etc.) can be observed. In addition, GPS receivers can easily be connected to most of these devices, then installing a map software (e.g. Microsoft Packet Streets) a complete navigation system can be developed. As an operating system Microsoft Mobile 2003 SE is one of the candidates targeting these devices.

Though the complete functionality is available, it is not possible to directly use these personal devices as on-board controllers on public transportation vehicles. In this field the environment calls for more ruggedized devices. Fortunately, the appearance of low cost high performance PC/104 (and other PC compatible) embeddable computers enables the application of powerful Operating Systems, such as Windows CE or Windows XP Embedded. The first public transportation vehicles equipped with on-board computers appeared quite a long time ago. Most of these systems were deeply embedded and provided relatively simple services, such as GPS based location identification and automatic next stop annunciation. Modern on-board computer systems can offer significantly wider ranges of services [Zhao97], as it will be described in details in section 2.

Applying modern managed *runtime platforms* (such as the .NET Framework or the .NET Compact Framework) on top of the Operating System these computers open new ways for *rapid application development* providing high quality integrated services. The .NET Framework is a *general purpose framework* which provides general services that are used by most applications.

This paper presents a *specialized component based framework* built on top of the .NET Framework to maximize the productivity developing embedded navigation applications. With the help of this framework the developers should be able to create

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2004 *workshop proceedings*,
ISBN 80-903100-4-4
Copyright UNION Agency – Science Press, Plzen, Czech Republic

complex navigational applications offering strongly integrated high level services. These services are realized by components, the low coupling and flexible configurability of components is guaranteed by the framework according to the fundamental *design for change and reuse* concept.

The core services of this framework constitute the infrastructure that takes care of the component related (configuration, startup, communication) and the threading issues, so that application developers can fully focus on solving application specific problems. Utilizing this infrastructure the first step is to develop components, then to configure and wire them together (in an XML configuration file) according to the application specific requirements.

Although the framework currently requires the full .NET framework due to making use of a few legacy components by COM Interop, care has been taken to minimize the utilization of features that are not available in the .NET Compact Framework to ease porting in the future. The navigation framework currently targets the Windows XP Embedded platform with the full .NET Framework installed.

A remarkable amount of research has been conducted related to embedded component based frameworks [Muller01], [Doucet02]. Most of these frameworks have been developed to realize applications running on low performance embedded chips and therefore a lot of emphasis has been put on performance, memory optimization, real-time execution requirements and similar issues, which can be best achieved by unmanaged C or C++ runtime environments. As a typical example, the Balboa component framework [Doucet02] uses C++ for component development and defines a high level component integration language (CIL) that supports introspection and loose typing. Most achievements of this environment are readily provided by the .NET Framework. Instead of focusing on component integration problems in unmanaged environments the framework described in this paper targets embedded systems offering high level, complex, strongly integrated services by loosely coupled reusable components. Also, medium to high performance embedded computers are supposed to be available running managed execution environments.

2. EMBEDDED AND MOBILE NAVIGATION SYSTEMS

Public transportation systems exhibit a set of highly complex navigation applications for on-board computer systems [Bened00], [Bened04]. These systems can automate several tasks, such as making next stop announcements or driving electronic next-stop, route and destination signs. Though the vehicle position can be determined applying a GPS, in most cases *dead reckoning* capabilities based on evaluating odometer and compass information are also required to handle those cases when GPS satellite visibility is blocked by high buildings or tunnels.

Processing normal operating records on-board systems can feed a number of statistical calculations: passenger counting on a per stop basis, logging detected off-route events (detours), detecting late arrivals, early departures and alarm conditions represent a great source of valuable business information for transit authorities.

The offload of collected data and the update of the on-board route-schedule database can be fully automated if the vehicle and the garage have Wireless LAN installed and appropriately configured.

Images taken by on-board cameras can also be captured so that accidents and incidents can be played back to clarify what happened in a specific situation.

Vehicles can be connected online to the dispatch centre via some remote communication link, such as GPRS (General Packet Radio Service). Vehicles report their position and status (alarm, delay, device status, vehicle health problems) so that dispatchers can see and effectively handle fleet related problems.

As described above, a typical on-board system is connected to several back-end and front-end systems. Figure 1 depicts a whole fleet information system built around the on-board system.

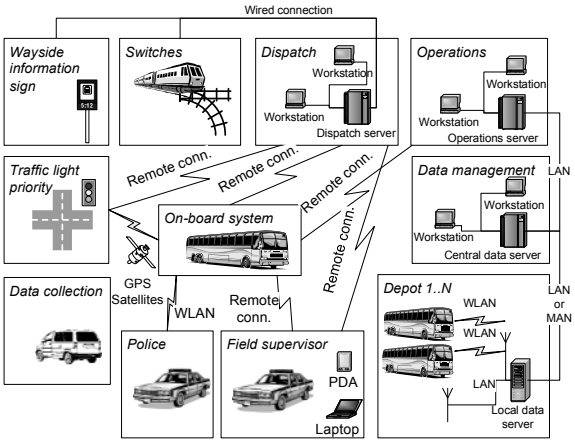


Figure 1. The system architecture of a fleet information system

There are several key points regarding the software modules running on the on-board computer. Beyond the fact that a number of tasks have to be handled by the same computer, these tasks have to work in a tightly integrated way. For instance, when the driver changes route, the new route identifier has to be has to be sent to the electronic signs installed on the

vehicle. In addition, it has to be logged into a file (source of statistical data), it has to be sent to the dispatching centre via the remote communication link and it has to be used as labeling data for the captured video frames. When developing applications the programmers have to be able to handle this complex net of interrelationships between the system modules. There is another important aspect: different transit authorities may have significantly different requirements, different vehicle infrastructures, so each software component should be easily replaceable and new modules should be easily pluggable without affecting the others.

Figure 2 illustrates the most important logical components required on board in case of a public transportation vehicle.

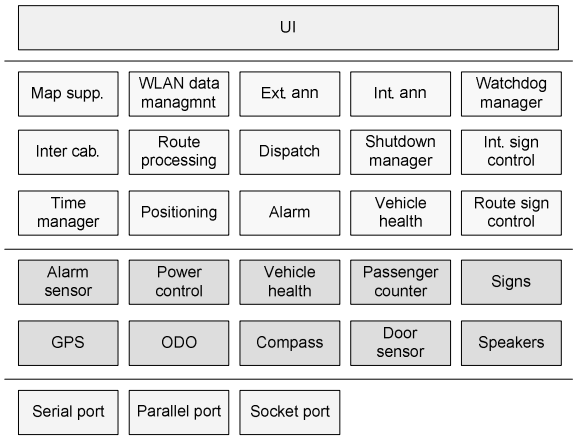


Figure 2. On-board components

In the following sections the component model of the framework, the details of component definition, component configuration, system startup, inter-component communication and threading issues will be presented.

3. COMPONENTS

Component Definition

As mentioned earlier, the framework follows a component-based approach. A component in our case has a more specific meaning than the general technical definition of most books and papers. A component conceptually is a building block encapsulating some processing logic and providing services for other components. It is a wrapper around some domain specific logic allowing the framework to treat components uniformly and to provide services for component startup, initialization, configuration, shutdown, according to the Service Configurator design pattern [Dougl99]. Components can contain an arbitrary number of objects possibly realizing very complex component logic.

Components are implemented in .NET assemblies to facilitate flexible system configuration: enabling or disabling a component will result in the given component being loaded or not loaded by the framework at startup, respectively. Components are specialized based on the layer they belong to. As a matter of fact, three layers are defined. *Port type components* (serial ports, parallel ports, sockets) form the lowest layer, they communicate with the outside world. *Device type components* abstract GPS, odometer and other devices for the higher level logic. *Application logic components* perform the real domain specific tasks. Fig. 3 shows one simple configuration made up by eight components.

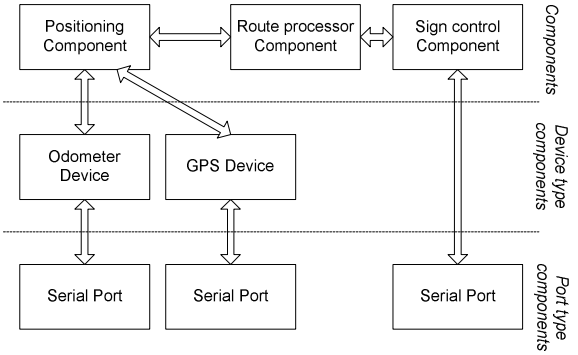


Figure 3. A simple system configuration

Component Structure

Each component has to have certain predefined interfaces and has to follow some predefined rules so that it can be managed by the framework. Components are composed of three parts as shown in Figure 4.

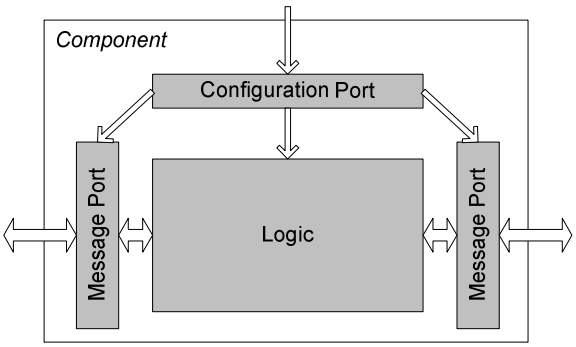


Figure 4. Component structure

The *Logic* part represents the objects performing the domain specific tasks the component is responsible for.

The *Configuration Port* is the connection point to the central Component Manager object, which is the central configuration management unit of the whole framework with respect to component startup, shutdown and configuration. The ConfigurationPort class provides built in support for the component for

starting/stopping/restarting and configuring based on a configuration file section belonging to the component. The component developer has to derive a new class from the ConfigurationPort class. This class has a CreateSubComponents method, which shall be overridden by the component developer to instantiate the objects internal to the component.

The *Message Port* objects implement message-based communication with other components, as it will be described in more detail later on. They basically receive messages from components they are connected to and forward messages to the Logic part. They also transform events raised by the Logic part to messages to be published to other components.

Although the development of a new component may look complicated, all what a component developer in most cases has to do (besides writing the component logic code) is derive two classes from the ConfigurationPort and MessagePort classes and override some of their methods.

4. COMMUNICATION

Conceptual Considerations

The cornerstone of being able to create applications that are manageable and extendible despite the complex interrelationships between their components is to define a communication infrastructure that yields low coupling between components. Instead of using strongly typed interfaces, the communication primarily based on events following the publisher-subscriber pattern results in a far more flexible solution. Taking this into consideration an event driven approach following the push model [Szyper98] has been chosen, no direct method calls are used.

The push model communication concept naturally suits the event-driven nature of the embedded application domain the framework has been primarily designed for. The representative configuration shown in Figure 3 is a good example. In most cases the GPS device calculates and sends new position information once in every second. This data is received by a serial port component, which raises an IncomingData event ("event" means conceptual events and not .NET events in this case). The GPS Device component receives this event as it is registered at the Serial Port component. It analyses the data and extracts the position information according to the communication protocol of the GPS device. The position information is sent to the Positioning component, which possibly checks GPS coordinate validity based on data received last from the odometer device and sends the noise filtered position to the Route Processor component. Next, the Route Processor checks if the vehicle has entered/left a close proximity of a stop and rises appropriate events. The Sign Control component is registered to stop change events and updates the electronic signs according to a specific sign communication protocol.

This approach has several advantages. First it inherently supports the broadcasting and multicasting of events. When the components are developed it is not known which other, not yet existing components will be interested in their events. This is not an issue if events are used. The subscription schema is defined in an XML configuration file processed by the framework at application startup. Even though this configuration file has to be edited manually now, an application with an intuitive user interface is being developed to help creating wiring definition.

Events can be implemented as sending and receiving *message objects*. In this case the parameter of a callback (event handler) method is always an object that can be perceived as a message encapsulating the type and the parameters of the event. This approach enables the logging of these messages to a log file during in the field operation, which can be played back in simulation mode later on. Multithreading combined with message queues yields a solution that can handle communication with slow hardware in a separate thread and also can hide threading issues from the programmer. Furthermore, messages can be easily serialized and sent via sockets making communication transparent across process or machine boundaries.

However, there are some liabilities. Realizing calls as sending messages makes calls requiring result more difficult to handle, as the correspondence between a particular request and a particular response has to be handled explicitly by the programmer.

Implementation

In the navigation framework inter-component events are implemented as sending and receiving message objects. In fact, messages are parameters of .NET delegates. When a .NET delegate is fired, the registered objects receive the message as the parameter of the event handler method. The type of event is encoded by the type of the message parameter. Therefore, there is no need to define a separate .NET event member for each event type, new message types can be introduced without modifying the interface of the related classes. The set of services offered and the events published by a component can be represented by different command and status type messages, respectively. There is a class hierarchy of messages with the IMessage interface as the root. Figure 5 shows a few message types involved in GPS communication.

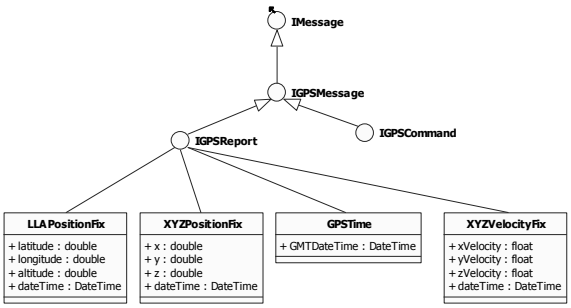


Figure 5. A part of the message class hierarchy

Each component has at least one Message Port object at the component boundary as mentioned earlier. This acts as a communication gateway between the logic part and other components. It has an OnNewMessage event handler method with an IMessage parameter, which is a reference to the message object received. The OnNewMessage method is registered (via .NET delegates) at other components and is called when there is a new incoming event for the component. The registration of this event handler at the event source components is performed by the framework at system startup. It is the responsibility of the component developer to "translate" incoming events to appropriate method calls into the component logic. The component developer also has to decide which events originated by the logic part should be transformed into messages and then forwarded to other components through the Message Port.

The core framework objects constituting the infrastructure services handle messages only through the IMessage interface, so they do not need to be recompiled when new components with new message types are introduced in the framework.

Messages are arranged into a class (or type, as interfaces are also involved) hierarchy. A few examples of different message types are the following: GPS related messages (e.g. position and time), position messages (either GPS or calculated from odometer and compass data), route processing status messages for detected off-route and late arrival conditions.

Filtering incoming events based on the type of the incoming event is possible in the Message Port. The following code snippet illustrates a component that is interested in GPS report messages only and ignores all other messages:

```
protected override void NewMessage(IMessage
message)
{
   if (message is IGPSReport)
        NewGPSReport(message);
}
```

The NewMessage method is an abstract method defined in the MessagePort base class. The OnNewMessage event handler calls the NewMessage operation for each incoming event. This way the component developer is forced to override the NewMessage method in his/her MessagePort derived class and handle the events appropriately.

The next piece of code illustrates how the MessagePort "derived" object translates the incoming message to a method call into the component logic.

```
protected override void NewMessage(IMessage
message)
{
   if (message is LLAPositionFix)
   {
        theComponentLogic.InLLAPositionPacket
        (message as LLAPositionFix);
   }
        …
}
```

If in case of a specific component the same object is the target of all incoming events it is possible to free up the programmer from writing any dispatching code. The name of the method to be called can be derived from the name of the type of the message object parameter according to a certain naming convention. First the existence of the method should be checked. If the method exists then it can be called with the message as a parameter. The next code fragment realizes this simple algorithm:

```
protected override void NewMessage(IMessage
message)
{
   string methodName = "On" +
        message.GetType().Name;

   MethodInfo minfo=
   theCompLogic.GetType().GetMethod(met
   hodName);

   if (minfo!=null)
        minfo.Invoke(theCompLogic,
        new object[] {message});
}
```

Mainly the receiving aspect of communication has been discussed so far. With respect to sending events to other components the developer of a component can call the dispatchMessage(IMessage message) method of the MessagePort class to send events to other components.

5. COMPONENT WIRING

The communication model of the framework supporting the publisher-subscriber pattern does not provide by itself a comprehensive solution. If the definition of component wiring - connecting subscribers to publishers - is awkward, then building specific target applications remains difficult.

Therefore, the configuration of components with the wiring information can be defined in an XML configuration file. Each component has a section within this file with a unique name, the assembly

name containing the code of the component, the class name of the Configuration Port object of the component, the enabled/disabled state, the configuration data specific to the component, and a ComponentConnections section with the list of the names of the components whose messages the component subscribes to. The next section shows a fraction of a sample configuration file.

```
...
<Component>
   <UniqueName>Route Processor</UniqueName>
   <Assembly>RouteProcessor</Assembly>
   <Class>RouteProcessor.RouteProcConfPort</
      Class>
   <Enabled>True</Enabled>
   <ComponentConnections>
      <ComponentConnection>Positioning
         Component</ComponentConnection>
      <ComponentConnection>Main
         UI</ComponentConnection>
      </ComponentConnections>
   <AutomaticPathStart>True
      </AutomaticPathStart>
</Component>
<Component>
   <UniqueName>Positioning
        Component</UniqueName>
   <Assembly>PositioningComponent</Assembly>
   <Class>PositioningComponent.PositioningCo
        mponent ConfPort</Class>
   <Enabled>True</Enabled>
   <ComponentConnections>
      <ComponentConnection>GPS Device</
        ComponentConnection>
   </ComponentConnections>
</Component>
...
```

The framework processes the configuration file at startup. The next code sample shows how simple .NET reflection makes the dynamic loading of assemblies and the creation of objects, given only the name of the class as a string:

```
…
ComponentListElement cle;
Assembly assembly = Assembly.LoadFrom(
  cle.assemblyName + ".dll");

Type type = assembly.GetType(
        cle.className);

cle.componentRef =
(ComponentConfManager)Activator.CreateInsta
        nce(type);

cle.isLoaded = true;
```

The ComponentListElement is a simple class holding the name of the class to be instantiated, the name of the source assembly and a reference to the created object.

6. SYSTEM STARTUP

The navigation framework instantiates a singleton Component Manager object at application startup, which takes care of most of the component loading, instantiation, configuration and cleanup tasks.

The Component Manager processes the system configuration file, instantiates the Configuration Port object of the enabled components based on their assembly and class name, and stores a reference to them in a running component list. Having the Configuration Port object instantiated its CreateSubcomponents method is called. This method has to be overridden by the component developer to instantiate the objects internal to the component. In the next step the Component Manager calls the Configure method on the Configuration Port object of the component (passing the XML path to the section of the configuration file belonging to the component) so that the component can configure itself. The same steps are performed for each component in turn. At this point the Component Manager registers the OnNewMessage event handler method of the subscriber Message Port objects at the event source Message Port objects according to the component wiring schema defined in the configuration file. Now the runtime configuration has been set up and the communication between components is enabled. The framework calls the Start method of the Configuration Port objects, which can be overridden by the component developer giving a chance to perform startup tasks specific to components.

7. THREADING

Most components perform tasks that do not take a long processing time and therefore can run in the main thread of the application without involving a thread context switch. Most *port type* components (e.g. serial ports) however have to perform long running blocking operations, for example reading data from a device. These operations can not be performed in the main thread as it would halt all other components running here. Consequently, port type components inherently have to utilize multithreading. However, most application and component developers do not have experience in handling multithreading issues (applying mutual exclusion locks properly and avoiding dead locks). For these reasons the framework has built in threading support for port type components. Each port type component starts a separate thread to perform blocking operations and hides it from the component developer.

The key issue is how to dispatch events to components connected to the port from within the *main thread* (for instance when new data is received). If the events *originated from the external thread* of the port can be routed by the port to itself in a way that they are *triggered to other components from within the main thread*, then from this point the intra-thread event dispatching mechanism (discussed in section 4) can be used. The key question is how to

"inject" the call into the main thread for the specific port object. .NET delegates form the bases of the solution as they encapsulate both the target objects and the method to be called.

Each thread (including the main thread) creates a synchronized message queue in its TLS (Thread Local Storage) memory area to hold references to messages sent by other threads. Figure 6 outlines the solution.

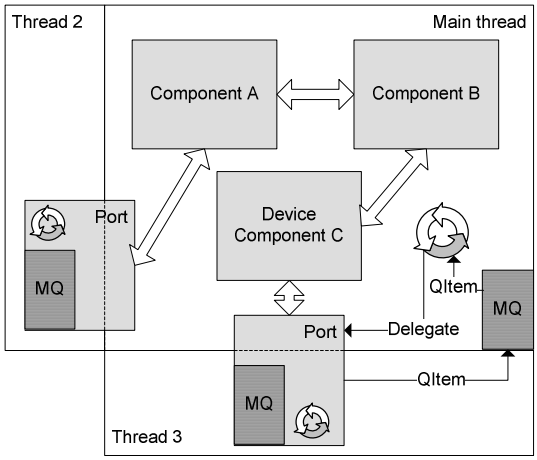


Figure 6. Threads and message queues

When new data is received by the port, an object of class QueueItem is created (abbreviated as QItem in Figure 6) by the external thread. This class encapsulates the delegate to be fired with its parameter, which is always of the type IMessage.

```
public class QueueItem
{
   public NewMessageEvent messageDelegate;
   public IMessage message;
};
```

The *messageDelegate* member is set up by the component, so that when the delegate is fired a specific method (dispatchMessageHandler) of the *sender* component is called. The *message* member is set to the message object parameter, for instance to and object of class IncomingData in case of a serial port component that has just received new data. The newly created QueueItem object is put into the message queue of the main thread. The main thread taking out the QueueItem object from its queue fires its *messageDelegate* member, passing the *message* member as a parameter:

```
queueItem.messageDelegate(queueItem.message)
```

This results in calling the dispatchMessageHandler event handler method of the *same* Message Port object from within the *main thread*. The dispatchMessageHandler dispatches the event to all registered components exactly the same way as sending events to components residing in the same thread, making threading fully transparent for the *receiving* component. Sending messages across thread boundaries is also made transparent for the *sender* component. The developer of a port type component calls the dispatchMessage(message) method in the MessagePort derived object to send messages to other components. This method is inherited from the MessagePort base class and is implemented differently for port type components, which use a message queue, and the regular components, which use direct invocation for message dispatching.

The scenario is very similar with respect to the flow of events in the opposite direction. If data is sent by a higher level component to a port type component, the event with the message parameter is transparently dispatched from the main thread to the message queue of the port, and from the queue to the appropriate handler function.

A different approach would be using .NET asynchronous delegates. Calling a delegate asynchronously results in executing the method encapsulated by the delegate by a thread pulled from a thread pool. This solution however would require the called method to take care of the synchronization by using appropriate locks when accessing shared data, and threading issues could be not hidden by the framework.

It should be mentioned that the .NET Framework has built in support for message queuing based on the COM+ MSMQ facility, made available under the Messaging namespace. The MSMQ targets enterprise applications, and is less suitable for performance sensitive embedded systems. The message queue implementation of the navigation framework uses the System.Collections.Queue class and the synchronization for the queue is provided by the System.Threading.Monitor class, together yielding a simple and efficient implementation.

8. SIMULATION

One of the most appealing benefits of message based inter-component communication is that messages can be serialized to a log file during in the field operation. This log file can be played back later on in simulation mode enabling the full reconstruction of system behavior. This feature can be really useful for instance in the fine tuning phase of developing dead reckoning algorithms, or when an accident occurs and the exact scenario has to be played back with an enhanced user interface representation (e.g. with the vehicle symbol animated on a digital map). Only messages created by *port type* component are logged as these are the components that communicate with the outside world. Each message is logged with the

name of the component that just generated the message. During simulation the simulation manager object reads the log file, extracts the messages and forwards them to that particular component which originally generated the message. According to this fundamental design concept applications can be started either in normal or in simulation mode. Most components are unaware of the actual mode, only port type components have to able to be switched to normal or simulation mode. In simulation mode the port type components are supposed to disable their hardware connections and dispatch data sent by the simulation manager.

9. SAMPLE APPLICATION

The infrastructure services of the framework have been implemented in the .NET Framework. Besides, the following components, which can be used as building blocks for developing applications have been developed so far: Serial Port, GPS Processor (handling Trimble TSIP protocol), Positioning and Route Processing. The Route Processing component encapsulates relatively complex component logic: it is fully capable to determine the relative position of the vehicle according to the route number set by the driver, can trigger next stop announcements and similar events. A map framework using an appropriate digital map database has also been developed within the ESRI MapObjects map component framework, which was built into the user interface component to visualize vehicle location, vehicle status and the current route with all the stops along that route. To demonstrate system capabilities an application has been composed using these components. Figure 7 illustrates the application:

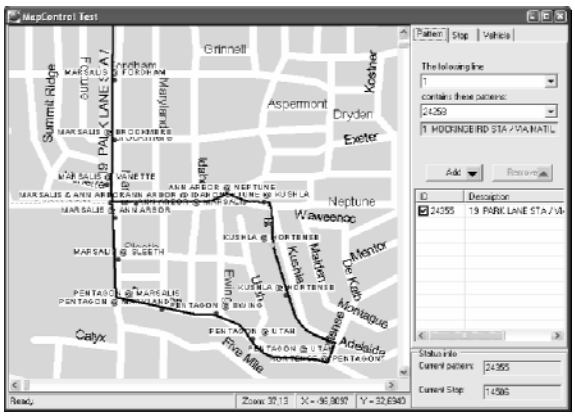


Figure 7. An application built utilizing the framework

10. CONCLUSION

A framework effectively supporting the development of complex, highly integrated, easily extendible applications in event-driven environments has been presented in this paper. As a primary application field navigation on-board computer systems have been discussed. When creating application building blocks developers can easily encapsulate system logic into components. Once the components have been developed, creating specific target applications from these components is straightforward. A communication mechanism has been elaborated following the push model: events are dispatched by the framework from the source component to each component connected to the source according to the component wiring schema. The wiring itself is defined in a configuration file, along with the individual settings specifically required by the given component. Future work will include hiding objects common to all components (Message Ports and Configuration Ports) from the component developers and enable them to provide these as .NET attributes. This will further simplify the component development process.

11. ACKNOWLEDGEMENTS

The author would like express his thanks to István Bihari, Balázs Osztatni and Milan Trenovszki for their valuable contribution in designing and implementing both the framework and the building block components.

12. REFERENCES

[Bened03] Z. Benedek: Intelligent Integrated Fixed Route Transportation Systems, Wesic Conference, 2003

[Bened04] Z. Benedek, S. Juhász: Intelligens Járműfedélzeti Rendszerek, Elektrotechnika folyóirat (Intelligent On-board Computer Systems, Journal of Electronics), 2004

[Doucet02] An Environment for Dynamic Component Composition for Efficient Co-Design, In Proc. Design Automation and Test in Europe, 2002

[Dougl99] S. Douglas et al.: Pattern-Oriented Software Architecture, Volume 2, John Wileys & Sons, Ltd, 1999

[Richt02] Jeffrey Richter: Applied Microsoft .NET Framework Programming, Microsoft Press, 2002

[Szyper98] C. Szypersky: Component Software – Beyond Object-Oriented Programming, Addison-Wesley, New York, 1998

[Muller01] Muller, P.O. et al.: Components @ work: component technology for embedded systems, Euromicro Conference, 2001

[Zhao97] Y. Zhao: Vehicle Location and Navigation Systems, Artech House, Inc, 1997

[Web01] ITS Online, <http://www.itsonline.com>

[Web02] The United States Department of Transportation, <http://www.dot.gov>

Compiling Scheme programs to .NET Common Intermediate Language

Yannis Bres, Bernard Paul Serpette, Manuel Serrano
Inria Sophia-Antipolis
2004 route des Lucioles - BP 93, F-06902 Sophia Antipolis, Cedex, France
{Yannis.Bres,Bernard.Serpette,Manuel.Serrano}@inria.fr

ABSTRACT

This paper presents the compilation of the Scheme programming language to .NET platform. .NET provides a virtual machine, the Common Language Runtime (CLR), that executes bytecode: the Common Intermediate Language (CIL). Since CIL was designed with language agnosticism in mind, it provides a rich set of language constructs and functionalities. Therefore, the CLR is the first execution environment that offers type safety, managed memory, tail recursion support and several flavors of pointers to functions. As such, the CLR presents an interesting ground for functional language implementations.

We discuss how to map Scheme constructs to CIL. We present performance analyses on a large set of real-life and standard Scheme benchmarks. In particular, we compare the performances of these programs when compiled to C, JVM and .NET. We show that .NET still lags behind C and JVM.

1. INTRODUCTION

Introduced by Microsoft in 2001, the .NET Framework has many similarities with the Sun Microsystems Java Platform [9]. The execution engine, the Common Language Runtime (CLR), is a stack-based Virtual Machine (VM) which executes a portable bytecode: the Common Intermediate Language (CIL) [8]. The CLR enforces type safety through its bytecode verifier (BCV), it supports polymorphism, the memory is garbage collected and the bytecode is Just-In-Time [1,17] compiled to native code.

Beyond these similarities, Microsoft has designed the CLR with language agnosticism in mind. Indeed, the CLR supports more language constructs than the JVM: the CLR supports enumerated types, structures and value types, contiguous multidimensional arrays, etc. The CLR supports tail calls, i.e. calls that do not consume stack space. The CLR supports closures through delegates. Finally, pointers to functions can be used although this leads to unverifiable bytecode. The .NET framework has 4 publicly available implementations:

- From Microsoft, one commercial version and one whose sources are published under a shared source License (Rotor [16]). Rotor was released for research and educational purposes. As such, Rotor JIT and GC are simplified and stripped-down versions of the commercial CLR, which lead to poorer performances.
- Ximian/Novell's Mono Open Source project offers a quite complete runtime and good performances but has only a few compilation tools.
- From DotGNU, the Portable.Net GPL project provides a quite complete runtime and many compilation tools. Unfortunately, it does not provide a full-fledged JIT [20]. Hence, its speed cannot compete with other implementations so we will not show performance figures for this platform.

1.1 Bigloo

Bigloo is an optimizing compiler for the Scheme (R^5RS [7]) programming language. It targets C code, JVM bytecode and now .NET CIL. In the rest of this presentation, we will use BiglooC, BiglooJvm, and Bigloo-.NET to refer to the specific Bigloo backends. Benchmarks show that BiglooC generates C code whose performance is close to human-written C code. When targeting the JVM, programs run, in general, less than 2 times slower than C code on the best JDK implementations [12].

Bigloo offers several extensions to Scheme [7] such as: modules for separate compilation, object extensions *à la* Clos [3] + extensible classes [14], optional type annotations for compile-time type verification and optimization.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
.NET Technologies'2004 *workshop proceedings,*
ISBN 80-903100-4-4
© UNION Agency - Science Press, Plzen, Czech Republic

Bigloo is itself written in Bigloo and the compiler is bootstrapped on all of its three backends. The runtime is made of 90% of Bigloo code and 10% of C, Java, or C# for each backend.

1.2 Motivations

As for the JVM, the .NET Framework is appealing for language implementors. The runtime offers a large set of libraries, the execution engine provides a lot of services and the produced binaries are expected to run on a wide range of platforms. Moreover, we wanted to explore what the "more language-agnostic" promise can really bring to functional language implementations as well as the possibilities for language interoperability.

1.3 Outline of this paper

Section 2 presents the main techniques used to compile Bigloo programs to CIL. Section 3 enumerates the new functionalities of the .NET Framework that could be used to improve the performances of produced code. Section 4 compares the run times of several benchmark and real life Bigloo programs on the three C, JVM and .NET backends.

2. COMPILATION OUTLINE

This section presents the general compilation scheme of Bigloo programs to .NET CIL. Since CLR and JVM are built upon similar concepts, the techniques deployed for these two platforms are close. The compilation to JVM being thoroughly presented in a previous paper [12], only a shallow presentation is given here.

2.1 Data Representation

Scheme polymorphism implies that, in the general case, all data types (numerals, characters, strings, pairs, etc.) have a uniform representation. This may lead to boxing values such as numerals and characters, i.e., allocating heap cells pointing to numbers and characters. Since boxing reduces performances (because of additional indirections) and increase memory usage, we aim at avoiding boxing as much as possible. Thanks to the Storage Use Analysis [15] or user-provided type annotations, numerals or characters are usually passed as values and not boxed, i.e. not allocated in the heap any more. Note that in the C backend, boxing of integers is always avoided using usual tagging techniques [6]. In order to save memory and avoid frequent allocations, integers in the range [-100 ... 2048] and all 256 characters (objects that embed a single byte) are preallocated. Integers are represented using the `int32` type. Reals are represented using `float64`. Strings are represented by arrays of `byte`s, as Scheme strings are mutable sequences of 1 byte characters while the .NET built-in `System.String`s are non-mutable sequences of wide characters. Closures are instances of `bigloo.procedure`, as we will see in Section 2.3.3.

2.2 Separate Compilation

A Bigloo program is made of several modules. Each module is compiled into a CIL class that aggregates the module definitions as static variables and functions. Modules can define several classes. Such classes are compiled as regular CIL classes (see §2.3.4). Since we do not have a built-in CIL assembler yet, we print out each module class as a file and use the Portable.Net assembler to produce an object file. Once all modules have been separately compiled, they are linked using the Portable.NET linker.

2.3 Compilation of functions

Functions can be separated in several categories:

- Local tail-recursive functions that are not used as first-class values are compiled as loops.
- Non tail-recursive functions that are not used as first-class values are compiled as static methods.
- Functions used as first-class values are compiled as real closures. A function is used as a first-class value when it is used in a non-functional position, i.e., used as an argument of another function call or used as a return value of another function.
- Generic functions are compiled as static methods and use an *ad hoc* framework for resolving late binding.

2.3.1 Compiling tail-recursive functions

In order to avoid the overhead of function calls, local functions that are not used as values and always called tail-recursively are compiled into CIL loops. Here is an example of two mutually recursive functions:

```
(define (odd x)
   (define (even? n)
      (if (= n 0) #t (odd? (- n 1))))
   (define (odd? n)
      (if (= n 0) #f (even? (- n 1))))
    (odd? x))
```

These functions are compiled as:

```
.method static bool odd(int32) cil managed {
  .locals(int32)
        ldarg.0      // load arg
  odd?: stloc.0      // store in local var #0
        ldloc.0      // load local var #0
        ldc.i4.0     // load constant 0
        brne.s loop1 // if not equal go to loop1
        ldc.i4.0     // load constant 0 (false)
        br.s end     // go to end
  loop1: ldloc.0     // load local var #0
        ldc.i4.1     // load constant 1
        sub          // subtract
  even?: stloc.0     // store in local var #0
        ldloc.0      // load local var #0
        ldc.i4.0     // load constant 0
        brne.s loop2 // if not equal go to loop2
        ldc.i4.1     // load constant 1 (true)
        br.s end     // go to end
```

```
  loop2: ldloc.0      // load local var #0
         ldc.i4.1     // load constant 1
         sub          // subtract
         br.s odd?    // go to odd?
  end:   ret          // return
}
```

2.3.2 Compiling regular functions

As a more general case, functions that cannot be compiled to loops are compiled as CIL static methods. Consider the following Fibonacci function:

```
(define (fib n::int)
  (if (< n 2) 1 (+ (fib (- n 1)) (fib (- n 2)))))
```

It is compiled as:

```
.method static int32 fib(int32) cil managed {
         ldarg.0        // load arg
         ldc.i4.2       // load constant 2
         bne.s loop     // if not equal go to loop
         ldc.i4.1       // load constant 1
         br.s end       // go to end
  loop:  ldarg.0        // load arg
         ldc.i4.1       // load constant 1
         sub            // subtract
         call int32 fib::fib(int32)
         ldarg.0        // load arg
         ldc.i4.2       // load constant 2
         sub            // subtract
         call int32 fib::fib(int32)
         add            // add
  end:   ret            // return
}
```

Note also that if their body is sufficiently small, these functions might get inlined (see [13]).

2.3.3 Compiling closures

Functions that are used as first-class values (passed as argument, returned as value or stored in a data structure) are compiled to closures.

The current closure compilation scheme for the JVM and .NET backends comes from two *de facto* limitations imposed by the JVM. First, the JVM does not support pointers to functions. Second, as to each class corresponds a file, we could not afford to declare a different type for each closure. We estimated that the overload on the class loader would raise a performance issue for programs that use closures intensively. As an example of real-life program, the Bigloo compiler itself is made of 289 modules and 175 classes, which produce 464 class files. Since we estimate that the number of real closures is greater than 4000, compiling each closure to a class file would multiply the number of files by more than 10.

In JVM and .NET classes corresponding to Bigloo modules extend `bigloo.procedure`. This class declares the arity of the closure, an array of captured variables, two kind of methods (one for functions with fixed arity and one for functions with variable arity), and the index of the closure within the module that defines it. In order to have a single type to represent closures, all the closures of a single module share the same entry-point function. This function uses the index of the closure to call the body of the closure, using a `switch`. Closure bodies are implemented as static methods of the class associated to the module and they receive as first argument the `bigloo.procedure` instance.

The declaration of `bigloo.procedure` is similar to:

```
class procedure {
  int index, arity;
  Object[] env;
  virtual Object funcall0();
  virtual Object funcall1(Object a1);
  virtual Object funcall2(Object a1, Object a2);
  ...
  virtual Object apply(Object as);
}
```

Let's see that in practice with the following program:

```
(module klist)
(define (make-klist n) (lambda (x) (cons x n)))
(map (make-adder 10) (list 1 2 3 4 5))
```

The compiler generates a class similar to:

```
class klist: procedure {
  static procedure closure0
    = new make-klist(0, 1, new Object[] {10});
  make-klist(int index, int arity, Object[] env) {
    super(index, arity, env);
  }
  ...
  override Object funcall1(Object arg) {
    switch (index) {
      case 0: return anon0(this, arg);
      ...
    }
  }
  ...
  static Object anon0(procedure fun, Object arg) {
    return make-pair(arg, fun.env[0]);
  }
  static void Main() {
    map(closure0, list(1, 2, 3, 4, 5));
  }
}
```

2.3.4 Compiling Generic Functions

The Bigloo object model [14] is inspired from Clos [3]: classes only encapsulate data, there is no concept of visibility. Behavior is implemented through generic functions, called generics, which are overloaded global methods whose dispatch is based on the dynamic type of their arguments. Contrarily to Clos, Bigloo only supports single inheritance, single dispatch. Bigloo does not support the Clos Meta Object Protocol.

In both JVM and CLR, the object model is derived from Smalltalk and C++: classes encapsulate data and behaviour, implemented in methods which can have different visibility levels. Method dispatch is based on the dynamic type of objects on which they are applied. Classes can be derived and extended with new data slots, methods can be redefined and

new methods can be added. Only single inheritance is supported for method implementation and instance variables, while multiple inheritance is supported for method declarations (interfaces).

Bigloo classes are first assigned a unique integer at run-time. Then, for each generic a dispatch table is built which associates class indexes to generic implementations, when defined. Note that class indexes and dispatch tables cannot be built at compile-time for separate compilation purposes. When a generic is invoked, the class index of the first argument is used as a lookup value in the dispatch table associated with the generic. Since these dispatch tables are usually quite sparse, we introduce another indirection level in order to save memory.

Whereas C does not provide direct support for any evolved object model, JVM or CLR do and we could have used the built-in virtual dispatch facilities. However, this would have lead to serious drawbacks. First, as generics are declared for all objects, they would have to be declared in the superclass of all Bigloo classes. As a consequence, separate compilation would not be possible any more. Moreover, this would lead to huge virtual function tables for all the Bigloo classes, with the corresponding memory overhead. Finally, the framework we chose has two main advantages: it is portable and it simplifies the maintenance of the system. For these reasons, the generic dispatch mechanism is similar in the C, JVM and .NET backends.

2.4 Continuations

Scheme allows to capture the continuation of a computation which can be used to escape pending computations, but it can also be used to suspend, resume, or even *restart* them! If in the C backend, continuations are fully supported using `setjmp`, `longjmp` and `memcpy`, in JVM and CLR, the stack is read-only and thus cannot be restored. Continuation support is implemented using structured exceptions. As such, continuations are first-class citizens but they can only be used within the dynamic extent of their capture.

One way to implement full continuation support in JVM and CLR would be to manage our own call stack. However, this would impose to implement a complex protocol to allow Bigloo programs to call external functions, while this is currently trivial. Moreover, we could expect JITs to be far less efficient on code that manages its own stack. Doing so would thus reduce performances of Bigloo programs, which seems unacceptable for us. Therefore, we chose not to be fully R^{5}RS compliant on this topic.

3. .NET NEW FUNCTIONALITIES

In this section we explore the compilation of Scheme with CIL constructs that have no counterpart in the JVM.

3.1 Closures

If we consider the C implementation of closures as a performance reference, the current JVM and .NET implementations have several overheads:

- The cost of body dispatching depending on closure index (in the C backend pointers to functions are directly available).
- An additional indirection when accessing a captured variable in the array (in the C backend, the array is inlined in the C structures representing the closures).
- The array boundaries verification (which are not verified at run-time in the C compiled code).

The CLR provides several constructs that can be used to improve the closure compilation scheme: delegates, declaring a new class per closure, and pointers to functions [18]. We have not explored this last possibility because it leads to unverifiable code.

3.1.1 *Declaring a new type for each closure*

Declaring a new type for each closure, as presented in §2.3.3, would get rid of the indexed function call and enables inlining of captured variables within the class instead of storing them in an array. However, as we have seen, each JVM class is stored in its own file and there are more than *4000* closures in the compiler. Hence, we could not afford to declare a new class for each closure in the JVM backend: loading the closures would be too much of a load for the class loader.

This constraint does not hold in the .NET Framework as types are linked at compile-time within a single binary file. However, loading a new type in the system is a costly operation: metadata have to be loaded, their integrity verified, etc. Moreover we noted that each closure would add slightly more than 100 bytes of metadata in the final binary file, that is about more than 400Kb for a binary which currently weights about 3.8MB, i.e. a size increase of more than 10%.

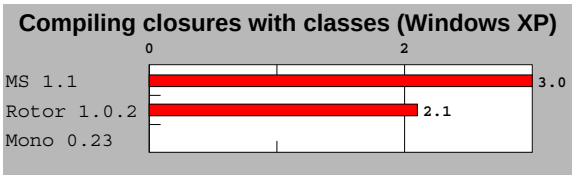


Fig. 1: Declaring a class per closure. This test compares the performance of two techniques for invoking closures: *declaring a type per closure* and *indexed functions*. Lower is better.

We have written a small benchmark program that declares 100 modules containing 50 closures each. For

each module, the program calls 10 times each 50 closures in a row. All closure functions are the identity, so this program focuses on the cost of closure invocations. Figure 1 shows that such a program always runs at least twice slower when we define a new type for each closure (Mono crashes on this test). Note that if the closures were invoked more than 10 times, these figures would decrease as the time wasted when loading the new types would be compensated by the acceleration of closure invocations. However, declaring a new type for each closure does not seem to really be a good choice for performances.

3.1.2 Using Delegates

The CLR provides a direct support for the Listener Design Pattern through Delegates which are linked lists of couples <*object reference*, *pointer to method*>. Delegates are a restricted form of pointers to functions that can be used in verifiable code while real pointer to functions lead to unverifiable code. Declaring delegates involves two steps. First, a delegate is declared. Second, methods whose signature match its declaration are registered. This is illustrated by the following example:

```
delegate void SimpleDelegate( int arg );
void MyFunction( int arg ) {...}
SimpleDelegate d;
d = new SimpleDelegate( MyFunction );
```

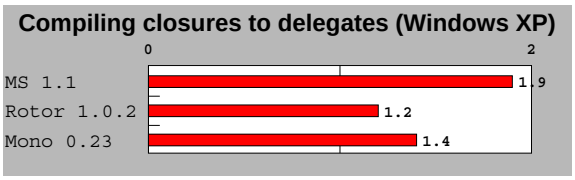


Fig. 2: Compiling closures to delegates. This test compares the performance of two techniques for invoking closures: *delegates* and *indexed functions*.

Figure 2 shows that our closure simulation program also runs slower when using delegates as surrogate pointers to functions instead of the indexed call. Such a result is probably due to the fact that delegates are linked lists of pointers to methods where we would be satisfied by single pointers to methods.

3.2 Tail Calls

The R^{5}RS requires that functions that are invoked tail-recursively must not allocate stack frames. In C and Java, tail recursion is not directly feasible because these two languages does not support it. The trampoline technique [2,19,5,11] allows tail-recursive calls. Since it imposes a performance penalty, we have chosen not to use it for Bigloo. As such, the Bigloo C and JVM backends are only partially compliant with the R^{5}RS on this topic.

In the CIL, function calls that precede a return instruction can be flagged as tail-recursive. In this case,

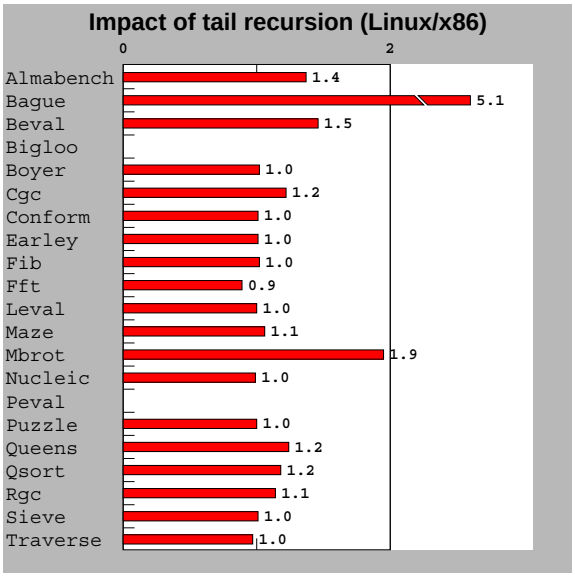


Fig. 3: This test measures the impact of tail recursion on .NET executions. Scores are relative to Bigloo.NET, which is the 1.0 mark. Lower is better.

the current stack frame is discarded before jumping to the tail-called function. The CLR is the first architecture considered by Bigloo that enables correct support of tail-recursive calls. For the .NET code generator, we have added a flag that enables or disables the CIL .tail call annotation. Hence, we have been able to measure, as reported Figure 3, the impact of tail calls on the overall performance of Bigloo programs (see §6 for a brief description of the benchmarks used). As demonstrated by this experiment, the slowdown imposed by flagging tail calls is generally small. However, some programs are severely impacted by tail recursion. In particular, the Bague program runs 5 times slower when tail recursion is enabled! This toy benchmark essentially measures function calls. It spends its whole execution time in a recursion made of 14 different call sites amongst which 6 are tail calls. This explains why this program is so much impacted by tail recursion.

The tail-call support of the .NET platform is extremely interesting for porting languages such as Scheme. However, since tail calls may slow down performance, we have decided not to flag tail calls by default. Instead we have provide the compiler with three options. One enabling tail-calls inside modules, one enabling them across modules, and a last one enabling them for all functions, including closures.

3.3 Precompiling binaries

With some .NET platforms, assemblies (executables and dynamic libraries) can be precompiled once for all. This removes the need for just-in-time compiling assemblies at each program launch and enables

heavier and more expensive program optimizations. Precompiled binaries are specifically optimized for the local platform and tuned for special features provided by the processor. Note that the original portable assemblies are not replaced by optimized binary versions. Instead, binary versions of assemblies are maintained in a cache. Since .NET assemblies are versioned, the correspondance between original portable assemblies and precompiled ones is straightforward. When an assembly is looked up by the CLR, preference is then given to a precompiled one of compatible version, when available.

Even if precompiling binaries is a promising idea for realistic programs such as Bigloo and Cgc (a simple C-like compiler), we have unfortunately measured no improvement using it. Even worse, we have even noticed that when precompiled these programs actually run slower!

4. PERFORMANCE EVALUATIONS

We have used a large set of benchmarks for estimating the performance of the .NET CLR platform. They are described in Figure §6, which also describes the platform we have used for these experiments. For measuring performance, we have used Mono .NET because it is the only implementation that is available on all main operating systems and because it delivers performance comparable to that of Microsoft CLR (when ran on Windows).

4.1 Bigloo vs C#

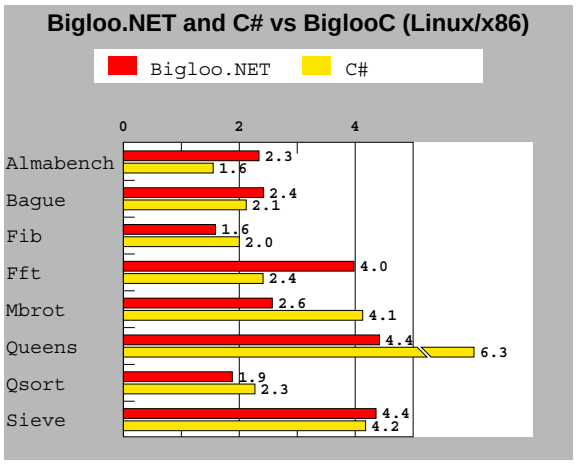


Fig. 4: This test compares the performance of Bigloo.NET vs C#. Scores are relative to BiglooC, which is the 1.0 mark. Lower is better.

To assess the quality of the CIL code produced by the Bigloo.NET compiler, we have compared the running times of the Bigloo generated code vs. regular human-written code in C# on a subset of our programs made of micro benchmarks that were possible to translate. For this experiment we use *managed* CIL code. That is, bytecode that complies the byte code verification rules of the CLR. Figure 4 shows that most Bigloo compiled programs have performances that are quite on par with their C# counterparts, but for Almabench and Fft. Actually the Bigloo version of these two benchmarks suffer from the same problem. Both benchmarks are floating point intensive. The Bigloo type inference is not powerful enough to get rid of polymorphism for these programs. Hence, many allocations of floating point numbers take place at run-time, which obviously slows down the overall execution time.

4.2 Platform and backend benchmarks

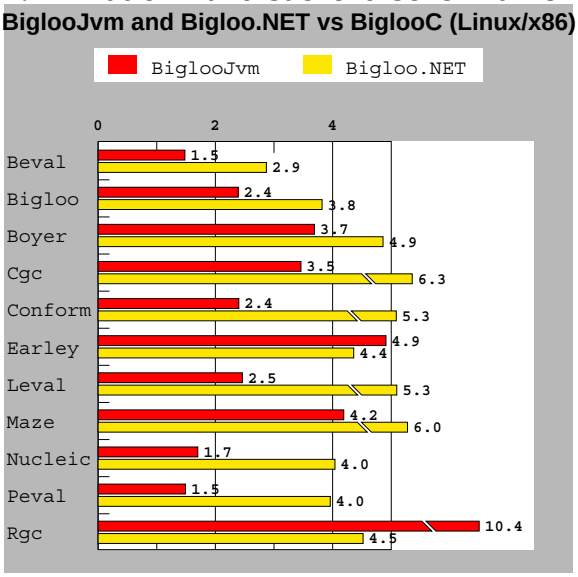


Fig. 5: This test compares BiglooJVM and Bigloo.NET. Scores are relative to BiglooC, which is the 1.0 mark. Lower is better.

Figure 5 shows the running times of several real-life and standard Scheme benchmarks for all three Bigloo backends. Since we are comparing to native code where no verification takes place, we have decided to measure the performance of *unmanaged* CIL bytecode and JVM bytecode that is not conform to the JVM bytecode verifier. (Figure 7 presents figures for *unmanaged* and *managed* CIL bytecode.)

In general, Bigloo.NET programs run from 1.5 to 2 times slower than their BiglooJvm counterpart. The only exceptions are Earley and Rgc for which Bigloo-.NET is faster. These two programs are also the only ones for which the ratio BiglooJvm/BiglooC is greater than 4. Actually these two programs contain patterns that cause trouble to Sun's JDK1.4.2 JIT used for this experiment. When another JVM is used, such as the one from IBM, these two programs run only twice slower than their BiglooC counterpart.

The benchmarks test memory allocation, fixnum and flonum arithmetics, function calls, etc. For all these topics, the figures show that the ratio between Bigloo-

Jvm and Bigloo.NET is stable. This uniformity shows that BiglooJVM avoids traps introduced by JIT*ted* architectures [12]. The current gap between Jvm and .NET performance is most likely due to the youth of .NET implementations. After all, JVM benefits from 10 years of improvements and optimizations. We also remember the time where each new JVM was improving performance by a factor of two!

4.2.1 *Impact of the memory management*

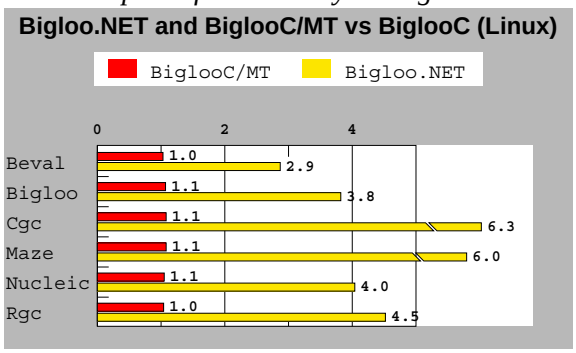


Fig. 6: This test measures the impact of multi-threading.

Both BiglooC (native) runtime system and Mono VM use the garbage collector developed by H-J Boehm [4]. However, as reported in Section 2.1, BiglooC uses traditional C techniques for minimizing the memory space of frequently used objects such as pairs or integers. In addition, BiglooC tunes the Boehm's collector for single threaded applications while Mono tunes it for multi-threading. In order to measure the impact of the memory management on performance, we have compiled a special BiglooC native version, called BiglooC/MT that used the very same collector as the mono one. As reported on Figure 6 BiglooC/MT is no more than 10% slower than BiglooC on real benchmarks. Therefore, we can conclude that memory management is not the culprit for the weaker performance of Bigloo.NET.

4.3 Related Work

Besides Bigloo, several projects have been started to provide support for Scheme in the .NET Framework. (*i*) Dot-Scheme [10] is an extension of PLT Scheme that gives PLT Scheme programs access to the Microsoft .NET Framework. (*ii*) Scheme.NET , from the Indiana University. (*iii*) Scheme.NET , from the Indiana University. (*iv*) Hotdog, from Northwestern University. Unfortunately we have failed to install these systems under Linux thus we do not present performance comparison in this paper. However, from the experiments we have conducted under Windows it appears that none of these systems has been designed and tuned for performance. Hence, they have a different goal from Bigloo.NET.

Beside Scheme, there are two main active projects for functional language support in .NET: (*i*) From Microsoft Research, F# is an implementation of the core of the CAML programming language.(*ii*) From Microsoft Research and the University of Cambridge, SML.NET is a compiler for Standard ML that targets the .NET CLR and which supports language interoperability features for easy access to .NET libraries. Unfortunately, as for the Scheme systems described above, we have failed in installing these two systems on Linux. Hence, we cannot report performance comparison in this paper.

5. CONCLUSIONS

We have presented the new .NET backend of Bigloo, an optimizing compiler for a Scheme dialect. This backend is fully operational. The whole runtime system has been ported to .NET and the compiler bootstraps on this platform. With the exception of continuations, the .NET backend is compliant to Scheme R^{5}RS. In particular, it is the first Bigloo backend that handles tail-recursive calls correctly. Bigloo.NET is available at: `http://www.inria.fr/mimosa/fp/-Bigloo`.

In conclusion, most of the new functionalities of the .NET Framework are still disappointing if we only consider performances as the ultimate objective. On the other hand, the support for tail calls in the CLR is very appealing for implementing languages that require proper tail-recursion. Currently .NET performance has not reached the one of Jvm implementation: Bigloo.NET programs run significantly slower than BiglooC and BiglooJvm programs. However there seems to be strong activity in the community of .NET implementors. Future will tell if next versions of .NET will bridge the gap with JVM implementations.

Bibliography

[1] Adl-Tabatabai, A. and Cierniak, M. and Lueh, G-Y. and Parikh, V. and Stichnoth, J. – **Fast, Effective Code Generation in a Just-In-Time Java Compiler** – Conference on Programming Language Design and Implementation, Jun, 1998, pp. 280–190.

[2] Baker, H. – **CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A <1>** – Sigplan Notices, 30(9), Sep, 1995, pp. 17-20.

[3] Bobrow, D. and DeMichiel, L. and Gabriel, R. and Keene, S. and Kiczales, G. and Moon, D. – **Common lisp object system specification** – special issue, Sigplan Notices, (23), Sep, 1988.

[4] Boehm, H.J. – **Space Efficient Conservative Garbage Collection** – Conference on Programming Language Design and Implementation, Sigplan Notices, 28(6), 1993, pp. 197–206.

[5] Feeley, M. and Miller, J. and Rozas, G. and Wilson, J. – **Compiling Higher-Order Languages into Fully Tail-Recursive Portable C** – Rapport technique 1078, Université de Montréal, Département d'informatique et r.o., Aug, 1997.

	Wall clock time in seconds					
Bench	Bigloo	BiglooJvm	BiglooJvm (vrf)	Bigloo.NET	Bigloo.NET (mgd)	Bigloo.NET (tailc)
Almabench	**5.54** (1.0 δ)	10.29 (1.85 δ)	*20.96* (3.78 δ)	8.72 (1.57 δ)	12.99 (2.34 δ)	12.01 (2.16 δ)
Bague	**4.71** (1.0 δ)	7.51 (1.59 δ)	7.61 (1.61 δ)	11.52 (2.44 δ)	11.42 (2.42 δ)	*58.81* (12.48 δ)
Beval	**5.98** (1.0 δ)	8.88 (1.48 δ)	9.17 (1.53 δ)	17.2 (2.87 δ)	24.16 (4.04 δ)	*25.2* (4.21 δ)
Bigloo	**19.2** (1.0 δ)	45.91 (2.39 δ)	46.34 (2.41 δ)	73.48 (3.82 δ)	*84.59* (4.40 δ)	*error*
Boyer	**8.43** (1.0 δ)	31.14 (3.69 δ)	30.61 (3.63 δ)	41.03 (4.86 δ)	*57.07* (6.76 δ)	41.9 (4.97 δ)
Cgc	**1.97** (1.0 δ)	6.82 (3.46 δ)	6.91 (3.50 δ)	12.4 (6.29 δ)	*19.26* (9.77 δ)	15.15 (7.69 δ)
Conform	**7.41** (1.0 δ)	17.82 (2.40 δ)	18.97 (2.56 δ)	39.4 (5.31 δ)	*48.44* (6.53 δ)	40.0 (5.39 δ)
Earley	**8.31** (1.0 δ)	40.86 (4.91 δ)	*41.91* (5.04 δ)	36.27 (4.36 δ)	40.61 (4.88 δ)	36.7 (4.41 δ)
Fib	**4.54** (1.0 δ)	7.32 (1.61 δ)	7.34 (1.61 δ)	7.27 (1.60 δ)	7.22 (1.59 δ)	*7.43* (1.63 δ)
Fft	**4.29** (1.0 δ)	7.8 (1.81 δ)	8.11 (1.89 δ)	15.26 (3.55 δ)	*17.1* (3.98 δ)	13.71 (3.19 δ)
Leval	**5.6** (1.0 δ)	13.8 (2.46 δ)	13.81 (2.46 δ)	29.91 (5.34 δ)	*35.11* (6.26 δ)	30.0 (5.35 δ)
Maze	**10.36** (1.0 δ)	43.5 (4.19 δ)	43.69 (4.21 δ)	62.18 (6.00 δ)	64.2 (6.19 δ)	*66.41* (6.41 δ)
Mbrot	**79.26** (1.0 δ)	199.26 (2.51 δ)	198.51 (2.50 δ)	205.9 (2.59 δ)	204.14 (2.57 δ)	*403.51* (5.09 δ)
Nucleic	**8.28** (1.0 δ)	14.1 (1.70 δ)	14.29 (1.72 δ)	33.53 (4.04 δ)	*37.85* (4.57 δ)	33.3 (4.02 δ)
Peval	**7.57** (1.0 δ)	11.28 (1.49 δ)	11.87 (1.56 δ)	30.01 (3.96 δ)	*32.47* (4.28 δ)	*error*
Puzzle	**7.59** (1.0 δ)	12.96 (1.70 δ)	13.03 (1.71 δ)	20.96 (2.76 δ)	*29.26* (3.85 δ)	21.0 (2.76 δ)
Queens	**10.47** (1.0 δ)	36.97 (3.53 δ)	37.75 (3.60 δ)	42.76 (4.08 δ)	46.37 (4.42 δ)	*53.33* (5.09 δ)
Qsort	**8.85** (1.0 δ)	13.26 (1.49 δ)	13.39 (1.51 δ)	16.93 (1.91 δ)	16.72 (1.88 δ)	*20.1* (2.27 δ)
Rgc	**6.94** (1.0 δ)	72.3 (10.41 δ)	*73.11* (10.53 δ)	31.43 (4.52 δ)	33.48 (4.82 δ)	35.96 (5.18 δ)
Sieve	**7.37** (1.0 δ)	25.44 (3.45 δ)	25.17 (3.41 δ)	31.01 (4.20 δ)	*32.19* (4.36 δ)	31.46 (4.26 δ)
Traverse	**15.19** (1.0 δ)	43.02 (2.83 δ)	43.24 (2.84 δ)	76.4 (5.02 δ)	*81.59* (5.37 δ)	74.21 (4.88 δ)

Fig. 7: Benchmarks timing on an AMD Tbird 1400Mhz/512MB, running Linux 2.4.21, Sun JDK 1.4.2, and Mono 0.30.

[6] Gudeman, D. – **Representing Type Information in Dynamically Typed Languages** – University of Arizona, Departement of Computer Science, Gould-Simpson Building, The University of Arizona, Tucson, AZ 85721, Apr, 1993.

[7] Kelsey, R. and Clinger, W. and Rees, J. – **The Revised(5) Report on the Algorithmic Language Scheme** – Higher-Order and Symbolic Computation, 11(1), Sep, 1998.

[8] Lidin, S. – **Inside Microsoft .NET IL Assembler** – *Microsoft Press*, 2002.

[9] Lindholm, T. and Yellin, F. – **The Java Virtual Machine** – *Addison-Wesley*, 1996.

[10] Pinto, P. – **Dot-Scheme A PLT Scheme FFI for the .NET framework** – Scheme workshop, Boston, MA, USA, Nov, 2003.

[11] Schinz, M. and Odersky, M. – **Tail call elimination of the Java Virtual Machine** – Proceedings of Babel'01, Florence, Italy, Sep, 2001.

[12] Serpette, B. and Serrano, M. – **Compiling Scheme to JVM bytecode: a performance study** – 7th Int'l Conf. on Functional Programming, Pittsburgh, Pensylvanie, USA, Oct, 2002.

[13] Serrano, M. – **Inline expansion: when and how** – Int. Symp. on Programming Languages, Implementations, Logics, and Programs, Southampton, UK, Sep, 1997, pp. 143–147.

[14] Serrano, M. – **Wide classes** – ECOOP'99, Lisbon, Portugal, Jun, 1999, pp. 391–415.

[15] Serrano, M. and Feeley, M. – **Storage Use Analysis and its Applications** – 1fst Int'l Conf. on Functional Programming, Philadelphia, Penn, USA, May, 1996, pp. 50–61.

[16] Stutz, D. and Neward, T. and Shilling, G. – **Shared Source CLI Essentials** – *O'Reilly Associates*, March, 2003.

[17] Suganama, T. et al. – **Overview of the IBM Java Just-in-time compiler** – IBM Systems Journal, 39(1), 2000.

[18] Syme, D. – **ILX: Extending the .NET Common IL for Functional Language Interoperability** – Proceedings of Babel'01, 2001.

[19] Tarditi, D. and Acharya, A. and Lee, P. – **No assembly required: Compiling Standard ML to C** – ACM Letters on Programming Languages and Systems, 2(1), 1992, pp. 161–177.

[20] Weatherley, R. and Gopal, V. – **Design of the Portable.Net Interpreter** – DotGNU, Jan, 2003.

6. APPENDIX: THE BENCHMARKS

Figure 7 presents all the numerical values on Linux 2.4.21/Athlon Tbird 1.4Ghz-512MB. Native code is compiled with Gcc 3.2.3. The JVM is Sun's JDK1.4.2. The .NET CLR is Mono 0.30. The JVM and CLR are multithreaded. Even single-threaded applications use several threads. In order to take into account the context switches implied by this technique we have preferred actual durations (wall clock) to CPU durations (user + system time). It has been paid attention to run the benchmarks on an unloaded computer. That is, the wall clock duration and the CPU duration of singled threaded C programs were the same.

Almabench (300 lines) floating point arithtmetic. **Bague** (105 l) function calls, fixnum arithtmetic, and vectors. **Beval** (582 l) the regular Bigloo Scheme evaluator. **Bigloo** (99,376 l) the bootstrap of the Bigloo compiler. **Boyer** (626 l) symbols and conditional expressions. **Cgc** (8,128 l) A simple C-to-mips compiler. **Conform** (596 l) lists, vectors and small inner functions. **Earley** (672 l) Earley parser. **Fft** (120 l) Fast Fourier transform. **Fib** (18 l) Fibonacci numbers. **Leval** (555 l) A Scheme evaluator using λ-expressions. **Maze** (809 l) arrays, fixnum operations and iterators. **Mbrot** (47 l) The Mandelbrot curve. **Nucleic** (3,507 l) floating point intensive compuations. **Peval** (639 l) A partial evaluator. **Puzzle** (208 l) A Gabriel's benchmark. **Queens** (131 l) tests list allocations. **Qsort** (124 l) arrays and fixnum arithmetic. **Rgc** (348 l) The Bigloo reader. **Sieve** (53 l) fixnum arithmetic and list allocations. **Slatex** (2,827 l) A LaTeX preprocessor. **Traverse** (136 l) allocates and modifies lists.

Teaching Compiler Development using .NET Platform

Andrey Terekhov
Microsoft Russia & CIS
Chapaevsky per., 14
125252, Moscow, Russia
terekhov@acm.org

Dmitry Boulychev, Anton Moscal, Natalia Voyakovskaya
St. Petersburg State University
Mathematical & Mechanical department
Universitetsky pr., 28, room 2360
198504, St. Petersburg, Russia
{db, msk, nat}@tercom.ru

ABSTRACT

We present our experience of teaching in the universities compiler development on the basis of .NET platform. We discuss typical problems of teaching compiler development and our approach to dealing with these problems. We consider applicability of .NET/Rotor in this context and share the lessons learned during preparation and delivery of this academic course.

Keywords

Compilers, teaching, .NET, Shared Source CLI, Rotor

1. INTRODUCTION

Compiler development is one of the oldest and the best researched topics in software engineering. It is a fundamental part of the universities computer science curricula. However, it is also one of the most difficult topics to teach. Students often find courses on compilers hard, because they have complex theoretical foundation and exercises require tedious coding. In most cases, the size of compilers that students have to write during the course exceeds anything they produced earlier. For these reasons, development of a course in compilers merits special consideration. The goal is to support early interest and understanding of the subject, and retain students' motivation throughout the course.

In 2001 we started rewriting an existing academic course on compiler development which ran in St. Petersburg State University since early 1970s. This course is offered to the students in the 3rd year of education and lasts for one academic semester (four calendar months). The course has been regularly updated every 5 to 10 years. At the beginning of our project it was based on the architecture of Intel microprocessors. At that time we have already had an experience of working with early releases of Visual Studio .NET and came to a conclusion that .NET represents a future-proof platform that could be used as a basis for the course on compiler development.

In March 2002 Microsoft announced its Shared Source Initiative (see http://sharedsourcecli.sscli.net), an open-source implementation of .NET that was informally code-named Rotor, and we launched a brief investigation on whether Rotor would be a better fit for the purposes of our course. It turned out that Rotor provides students with an excellent opportunity to become acquainted with a real-life compiler, as well as to get experience of working with the large (3.5+ million lines of code!) software code base while still at the university.

Simultaneously with Rotor's announcement, Microsoft Research issued a Request For Proposals aimed at supporting Rotor-based research and education projects. Our group was awarded one of the grants under this initiative. We have created a complete set of presentations and lecture notes for one-semester academic course on "Compiler Development for .NET Platform" in Russian and English languages, which is available at the Web-site of St. Petersburg State University (see http://www.iti.spbu.ru/eng/grants/Cflat.asp).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2004 *workshop proceedings,*
ISBN 80-903100-4-4
Copyright UNION Agency – Science Press, Plzen, Czech Republic

In this article we present the experience gained during preparation and delivery of this academic course. The article is organized as follows. In Section 2 we briefly discuss advantages and disadvantages of .NET platform from the point of view of supporting various programming languages. Section 3 presents typical problems of teaching compiler development and our approach to handling them. Section 4 contains a general overview of the course and a description of the deliverables that we have created during this project. In Section 5 we illustrate our approach to teaching compiler development using excerpts of the lectures. Finally, Section 6 summarizes our experiences so far and outlines some directions for further research.

2. WHY .NET?

One of the more popular directions of the last decade is *virtual machines* – a powerful concept, which abstracts away the differences between hardware platforms and thus enables portability of programs written in a particular programming language.

At the moment .NET is arguably the most promising of those virtual machines, because it was designed from the very beginning to support most of the existing programming languages, unlike previous efforts that were aimed at a single language. Thus .NET is a convenient platform for compiler development, which is more powerful than its predecessors, such as Java:

- In .NET there exists a special API for code generation (Reflection.Emit), while Java provides only file generation methods
- In .NET it is possible to pass a reference as a parameter and as an output value (in C# this options are represented by keywords **ref** and **out**). To emulate such behavior in Java one has to create a wrapper class that would be placed in a heap.
- .NET platform provides support for important encapsulation and abstraction mechanisms, such as properties and indexers. In Java this cannot be implemented directly, so one has to settle for the use of naming conventions, which the compiler does not verify.
- In .NET it is possible to generate unsafe (i.e., unverifiable) code. This could be useful, for instance, for the purposes of achieving runtime efficiency or integration of legacy systems. In Java this is possible only by calling programs written in other languages, such as C.

Naturally, .NET is not the perfect solution, and some of its advantages are based on subtle design trade-offs, which are especially visible during implementation of languages that do not correspond directly to the .NET model. Here are some features typical for various programming languages, but difficult to implement in a compiler to .NET platform:

- Multiple inheritance (Eiffel, C++)
- Nested procedures (Pascal, Algol 68)
- Parametric polymorphism (ML, Haskell)
- Constructors with user-defined names other than the name of the class (Pascal)
- Non-standard data types (for instance, consider the problems of supporting PICTURE data type used in Cobol and PL/I)

Finally, writing a compiler from almost any functional languages to .NET is somewhat problematic, because .NET is heavily biased towards traditional imperative languages. Such a compiler would lead to inefficiency of the generated code or would require generating unverifiable code.

Nevertheless, practice has shown that these problems are not crucial – there already exist dozens of compilers from various languages to .NET, and new compilers keep appearing. Writing a compiler for .NET platform as an exercise is relevant for the students ..NET continues to evolve – some of the above mentioned problems are already obsolete and others will probably get resolved in the upcoming releases of .NET (see .NET version 1.2, and research projects such as Gyro, see http://research.microsoft.com/projects/clrgen and ILX, see http://research.microsoft.com/projects/ilx).

3. ISSUES OF TEACHING COMPILER DEVELOPMENT

The following technical and psycholiogical issues need to be considered in preparation and delivery of a compiler development course [Chanon75, Appelbe79].

First of all, compiler courses deal with the complexity of the problem domain, especially with the abundance of mathematical theory. This presents more difficulty for students majoring in software engineering, since their curriculum is usually more practical than theoretical. In some cases, this problem leads to over-emphasis on theory at the expense of practical usefulness of the course; in other cases the course becomes all-embracing and overly time-

consuming for students. As a result, there exists a gap between compilers "as taught in the universities" and compilers "as written in the industry".

In order to overcome this problem, we tried to minimize the amount of theory by describing only those formalisms and theorems that are directly required for understanding the material of the course. Nevertheless, our course includes introduction to language and grammar theory, automata theory, data and control flow analyses, so purely theoretical material constitutes about one third of our course.

We also found it useful to separate the text of the lecture notes into "main text" and "digressions"., Main text contains theoretical explanations and description of universally accepted practices of compiler construction, while digressions are the advanced topics, such as practical tricks that are useful only under certain conditions or could be employed to overcome various limitations of the straightforward approaches[1]. This is useful for structuring the theoretical material of the course and presents the student with two different perspectives on compiler writing, from the computer science and software engineering points of view. Experienced students may also take advantage of this separation by concentrating only on those parts that are less well-known for them.

Secondly, for most students the size of compiler that they have to produce is much greater than all their previous projects at the university. In order to be successful, courses on compilers should run in the interactive mode and contain many possibilities for the student to get clarifications. One of the main methods to achieve this is to complement the lectures with the self-paced independent work by students, which, in our opinion, should be organized at regular hours in the university computer labs and should be supervised by either lecturer or assistant.

This course tries to teach not only the basics of compiler writing (so called "programming-in-the-small"), but also issues that are important for working with large code base (so called "programming-in-the-large"). We try to achieve both of these goals by first demonstrating the concepts of compiler writing using examples taken from a demo compiler of a simple language C-flat, which is a subset of C#, and then by illustrating advanced topics using examples taken from a full-blown compiler of C# that is available in Rotor. This approach shows the student the whole set of "under-the-hood" details of compilers that are usually too complicated for implementation in "toy languages" and are quite often omitted in academic courses on compilers:

- Possibility to illustrate various platform-dependent aspects, such as run-time support and generation of debugging information
- Demonstration of garbage collection, JIT-compilation and other system mechanisms
- System and auxiliary tools (assembler, disassembler, debugger etc.)
- Implementation of Foundation Class Library classes

Note that one cannot guarantee that the algorithms used in Rotor are equivalent to those used in Visual Studio .NET, since their goals are different – Rotor is designed to be as clear and understandable as possible, while .NET is striving to achieve maximum efficiency of the generated code. This makes Rotor good for teaching, but creates a risk that students will mechanically imply that the same algorithms work in industrial implementation and will rely on that false assumption in their work, so the lecturer should explicitly draw students' attention on this difference.

Finally, for some of the students understanding the target platform may be difficult, especially, at the code generation phase. The knowledge of .NET platform is not yet widespread, so it was a real problem of our course. We recommend starting the course with a two-lecture overview of .NET platform presented from the programming point of view. However, it might be a better idea to consider knowledge of .NET as a pre-requisite for this course. We have already started transition to this model, because now there is a separate course on .NET available for the students on elective basis earlier in their studies in St. Petersburg State University. This course was devised by one of the authors, Andrey Terekhov, and based on a well-known book [Richter02]. We believe that in the foreseeable future .NET will gain more popularity both in industry and academia and thus more universities will view this approach as a better alternative.

4. OVERVIEW OF THE COURSE

As a result of preliminary research and planning we came up with the following requirements to our

[1] The idea of separating material into "main text" and "digressions" was traditional for mathematical textbooks of the Soviet era. Typically, digressions represented reading that was not required for the students and were typeset in fine print.

course – the course should unite both theory and practice, complimenting both the theory-oriented text books, such as [Aho86, Muchnick97], and practice-oriented books, such as [Gough02]. The course should be based on .NET and particularly on examples taken from its open implementation, Rotor.

We wrote lecture notes and slides for this course based on the above requirements. The phase of active development lasted for about a year. As a result of this activity, we created the following 15 lectures:

1. Overview of .NET and Rotor
2. Overview of C#
3. Compiler Basics
4. Language Theory
5. Lexical Analysis
6. Syntax Analysis – Recursive Descent
7. LR(k) and LALR Grammars
8. Grammars and YACC
9. Semantic Analysis. Internal Representation
10. Memory Management
11. Optimization
12. Control Flow Analysis
13. Data Flow Analysis
14. Generation of CIL
15. Instruction Selection during Code Generation

Note that some of the lectures require more time for delivery than the usual one and a half hours that are typically allotted in Russian academic system, so this list represents logical division of the course into related topics rather than the recommended duration. We assume also that during the semester the students will additionally spend a comparable amount of time on review of source codes of C-flat and Rotor, and will independently implement a sample compiler according to the individual tasks set out by the lecturer.

The course material includes a sources and binaies of ademo compiler of a "toy language" called C-flat. The grammar of this language is intentionally simple – BNF grammar of C-flat takes less than 30 lines. Anton Moscal, one of the authors of the course, has produced the C-flat compiler.The course refers often to the compiler sources. We are currently trying to bootstrap C-flat compiler, i.e. we are trying to rewrite C-flat compiler in C-flat. This is a dual process, which requires both changing the compiler (i.e., using less powerful language constructions) and changing the language (i.e., expanding the set of allowed constructions).

In 2002-03 we made a pilot delivery of some of our lectures to the students of St. Petersburg State University. The lectures were well-received by the audience and generated a lot of feedback that we used to improve the contents of the course.

We also proposed topics for term work on the basis of the course to the students. One of the goals of these term assignments was to assess validity of our assumptions about students' knowledge prior to the course and the difficulty of course material. For instance, a 3rd year student was given a task to implement a C-flat compiler in C# in order to make sure that it is possible for a student to develop such a compiler during one semester. In another term project, a team of 4th year students was asked to develop a compiler from subset of Pascal to .NET that would be written in SML.NET using MLLex/MLYacc. Both of these projects were successfully completed, which suggests a strong evidence of importance of this course and relevance of its content.

At the moment we are considering several ideas on further development of the course. One of the ways to improve the course is to enlarge the scope by adding material on Mono project (see http://www.go-mono.com). Mono is another open-source implementation of .NET, which is interesting due to the fact that it uses different approaches to implementation of various aspects of .NET than Rotor, for instance:

- Mono C# compiler is written in C# and thus capable of bootstrapping itself. It might be argued that it is also more readable from the student's point of view than Rotor's compiler written in C++
- Garbage collection is implemented using Boehm's conservative garbage collector for C [Boehm88, Boehm93], which is a radically different approach to solving memory management issues
- Unlike Rotor C# compiler, Mono uses BURG [Pelegrì-Llopart88, Aho89, Fraser92] approach for instruction selection. This topic is briefly mentioned in the last lecture of our course, and thus Mono presents a good opportunity to illustrate this theory with a real-life example

5. AN EXAMPLE OF LECTURE MATERIAL

To demonstrate some of the ideas behind this course, we will briefly walk through the material of the lecture on memory management and garbage collection. This lecture relies heavily on examples from Rotor source code that are mostly adopted from the book [Stutz03], which we recommend as a supplementary reading to students.

We start the lecture with asserting that memory management is an extremely important and resource-consuming problem in programming. According to a classical textbook [McConnell93] in typical C projects memory management consumes up to 50% of the time dedicated to coding and debugging. As an example we consider Rotor's C# compiler that is written in C++ and thus is based on manual memory tracking and disposal by the programmer. In order to perform this task correctly, developers of C# compiler had to come up with a number of auxiliary structures, for instance, below we enumerated all the places where Rotor stores information about an object instance:

- Memory for object instances is stored in a garbage-collected heap (excluding SyncBlock, which is stored inside the execution engine itself)
- Method table of the object is placed in the "frequently used" heap of its application domain; in the meantime EEClass, FieldDescs and MethodDescs of the object are placed in "rarely used" heap
- Native code, generated by JIT-compiler, is placed in the code heap and is shared by all application domains
- All other stuff related to object (for instance, stubs generated for this object) are stored in a separate memory region of the execution engine

Clearly, manual tracking of all these elements is a tedious task and thus modern programming languages and platforms tend to rely on automatic memory management instead.

Then we provide an overview of existing methods of memory management. The students should already have this knowledge from the course on fundamentals of programming languages, but we believe that it is always helpful to provide a brief refresher on this topic.

We proceed to a detailed discussion of garbage collection scheme that is used in Rotor. This is important for students because it enables them to connect theoretical description from the previous part of lecture with the concrete implementation. The main focus is on practical consequences of the theory that we have just discussed and on the fact that programmers always have to deal with tricks and heuristics in order to increase efficiency of the implementation.

First, we emphasize the general approach to garbage collection in Rotor and explain the reasoning behind choosing particular garbage collection methods. We mention that Rotor uses hybrid scheme of garbage collection with two generations:

- Generation 0 is compacted by copying — this pays off since the majority of the objects goe away before the first GC, so not a great deal of copying takes place
- Generation 1 is collected by mark & sweep — copying would not be advantageous here, since few objects are dying in this generation

There are many interesting details:

- Copying is always performed to a new heap (to be more precise, to a new segment of the heap, see below); this process is not overly expensive because it takes place separately in different generations
- Large objects are collected in a separate heap *without copying* (see below)
- Completely different scheme exists in Rotor for garbage collection of remote objects (see sscli/clr/src/bcl/system/runtime/remoting)

Note that the issue of GC for remote objects could be also discussed in a separate lecture "Remoting in .NET", which should be a part of a separate academic course on .NET platform (this is the case for St. Petersburg State University, where these courses run in parallel).

Then we illustrate the implementation details by going from the top. We point to the main entry point of garbage collection – `GCHeap::GarbageCollect()`. We explain that it is called whenever a garbage collection is needed and enumerate possible scenarios for that to happen – for instance, if memory runs out, or an application domain is being unloaded, or finalizers have just

completed, or if there was an explicit call by the programmer, during exit from the process and during debugging from the profiler. After the call to this function, all threads are suspended except the thread that performs the GC – this is achieved by call to the following function:

```
SuspendEE(GCHeap::SUSPEND_FOR_GC);
```

Note that prior to that all threads should reach their "GC safe" state (this is a good place to explain what are the characteristics of this state). Then the control is passed to the main function, **gc_heap::gc1()**. It works as follows – an attempt of copying garbage collection in the zero generation is made (as a result, all live objects are moved to the first generation). If garbage collection is required for the first generation as well (this is almost always the case), then mark-and-sweep is performed in it.

After these high-level explanations, we walk the students through the actual Rotor code that performs these tasks and explain the implementation details. This is quite easy to do, because the code itself is properly commented and readable, not even to mention the detailed explanations provided in the book [Stutz03]. For instance, below is the code that we use to illustrate the first stage of GC, `gc_heap::copy_phase()`:

- Live objects are found by recursive search and copied into the elder generation:

```
// Promote objects referred to by cross-generational pointers
    copy_through_cards_for_segments (copy_object_simple_const);
    copy_through_cards_for_large_objects (copy_object_simple_const);
// Promote objects found on the stack or in the handle table
    CNameSpace::GcScanRoots(GCHeap::Promote, condemned_gen_number,
max_generation, &sc, 0);
    CNameSpace::GcScanHandles(GCHeap::Promote, condemned_gen_number,
max_generation, &sc);
// Promote any object referred to from the finalization queue
    finalize_queue->GcScanRoots(GCHeap::Promote, heap_number, 0);
```

- References to these objects are updated:

```
// Relocating cross generation pointers
    copy_through_cards_for_segments (get_copied_object);
    copy_through_cards_for_large_objects (get_copied_object);
// Relocating objects on the stack or in the handle table
    CNameSpace::GcScanRoots (GCHeap::Relocate, condemned_gen_number,
max_generation, &sc);
   CNameSpace::GcScanHandles(GCHeap::Relocate, condemned_gen_number,
max_generation, &sc);
// Relocating finalization data
    finalize_queue->RelocateFinalizationData (condemned_gen_number, __this);
```

Table 1. Rotor code used to illustrate the copying stage of garbage collection

After discussion of garbage collection we also briefly mention *code pitching*. This is just one of the examples of the topics that are difficult for the students to grasp (it does not come easily to the students that not only program data, but also the generated code could be treated as garbage) and yet is easily demonstrated using the Rotor source code. As this is more or less an aside detail, we illustrate only the general idea of code pitching and leave the implementation details for students' independent study. The scheme of this part goes as follows:

- When the size of the heap for the compiled code exceeds some predefined maximum, the whole contents of the buffer is thrown away and all return addresses on the stack are replaced with address of thunk that causes re-compilation of methods
- To make a decision on throwing the code away, this process takes into account a lot of parameters (size of native code, ratio of native code to IL, time of JIT-compilation of the method etc.)
- See sscli/clr/src/vm/ejitmgr.cpp

We believe that it is important to make the students study the source code of real-life compilers, because it provides a much better way to learn how to write the code and acquaints the students with all the intricate details before they encounter them in their professional work after graduation.

6. ACKNOWLEDGEMENTS

This work was financially supported by the Rotor grant by Microsoft Research. We would also like to thank Vladimir Pavlov (eLine Software Inc.), Dmitry Malenko (Dnepropetrovsk National University, Ukraine) and anonymous reviewers for their valuable comments that helped to improve this paper.

7. CONCLUSIONS

We presented an academic course on compiler development that is based on .NET and Rotor. In this course we tried to bridge the gap between compilers "as taught in the universities" and "real-world compilers" by demonstrating both theoretical and practical perspectives on compiler development. From our point of view, .NET can be successfully used as a platform for education and research in various areas of computer science and software engineering, such as compilers, programming languages and component architectures. We also found out that Rotor is an especially interesting platform for education because it enables students to get acquainted with typical problems of working with industrial large-scale software projects.

8. REFERENCES

[Appelbe79] B. Appelbe "Teaching Compiler Development", In Proceedings of the 10th SIGCSE Technical Symposium on Computer Science Education, 1979, pp. 23-27

[Aho86] A.V. Aho, R. Sethi, J. D. Ullman "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986, 500 pp.

[Aho89] A.V. Aho, M. Ganapathi, S.W.K. Tjiang "Code Generation Using Tree Matching and Dynamic Programming", ACM Transactions on Programming Languages and Systems", Vol. 11, No. 4, 1989, pp. 491-516.

[Boehm88] H.-J. Boehm, M. Weiser "Garbage collection in an uncooperative environment", Software Practice & Experience, Vol. 18, No. 9, 1988, pp. 807-820.

[Boehm93] H.-J. Boehm "Space efficient conservative garbage collection", In Proceedings of the Conference on Programming Language Design and Implementation, 1993, pp. 197-206.

[Chanon75] R. Chanon "Compiler construction in an undergraduate course: some difficulties", ACM SIGCSE Bulletin, Vol. 7, No. 2, 1975, pp. 30-32.

[Crowe02] M.K. Crowe "Compiler Writing Tools Using C#", see http://cis.paisley.ac.uk/crow-ci0/

[Fraser92] C.W. Fraser, R.R. Henry, T.A. Proebsting "BURG – Fast Optimal Instruction Selection and Tree Parsing", SIGPLAN Notices, Vol. 27, No. 4, 1992, pp. 68-76.

[Gough02] J. Gough "Compiling for .NET Common Language Runtime", Addison-Wesley, 2002

[McConnell93] S. McConnell "Code Complete", Microsoft Press, 1993

[Muchnick97] S. Muchnick "Advanced Compiler Design and Implementation", Morgan Kaufmann, 1997, 856 pp.

[Pelegrì-Llopart88] E. Pelegrì-Llopart, S.L. Graham "Optimal Code Generation for Expression Trees: An Application of BURS Theory", Proceedings of the Conference on Principles of Programming Languages, 1988, pp. 294-308.

[Richter02] J. Richter “Applied .NET Framework Programming”, Microsoft Press, 2002

[Stutz03] D. Stutz, T. Neward, G. Shilling "Shared Source CLI Essentials", O'Reilly, 2003

Experience Integrating a New Compiler and a New Garbage Collector Into Rotor

Todd Anderson Marsha Eng Neal Glew
Brian Lewis Vijay Menon James Stichnoth

Microprocessor Technology Lab, Intel Corporation
2200 Mission College Blvd., Santa Clara, CA, 95054, U.S.A.
james.m.stichnoth@intel.com

ABSTRACT

Microsoft's ROTOR is a shared-source CLI implementation intended for use as a research platform. It is particularly attractive for research because of its complete implementation and extensive libraries, and because its modular design allows different implementations of certain components (*e.g.*, just-in-time compilers). Our group has independently developed a high-performance just-in-time (JIT) compiler and garbage collector, and wanted to take advantage of ROTOR as a platform for experimenting with these components. In this paper, we describe our experiences integrating these components into ROTOR, and evaluate the flexibility of ROTOR's modular design toward this goal.

We found the just-in-time (JIT) compiler easier to integrate than the garbage collector because ROTOR has a well defined interface for the former but not the latter. However, the JIT integration required changes to ROTOR to support multiple JITs, which included implementing a new code manager and supporting a second JIT manager. We detail the changes to our just-in-time compiler to support ROTOR's calling conventions, helper functions, and exception model. The garbage collector integration was complicated by the many places in ROTOR where components make assumptions about the garbage collector's implementation. It was also necessary to reconcile the different assumptions made by our garbage collector and ROTOR about object layout, virtual-method table layout, and thread structures.

Keywords

CLI, Java, virtual machine, just-in-time compilation, dynamic compilation, garbage collection, calling conventions, software interfaces

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
.NET Technologies'2004 workshop proceedings,
ISBN 80-903100-4-4
Copyright UNION Agency - Science Press, Plzen, Czech Republic.

1. INTRODUCTION

ROTOR, Microsoft's Shared Source Common Language Infrastructure [7, 9], is an implementation of CLI (the Common Language Infrastructure [6]) and C# [5]. It includes a CLI execution engine, a C# compiler, various tools, and a set of libraries suitable for research purposes (it omits a few security and other commercially important libraries). As such, it provides a basis for doing research in CLI implementation, and Microsoft is encouraging this use of ROTOR.

Our group has been doing research for a number of years on the implementation of managed runtime environments for Java and CLI on Intel platforms. As part of this effort, we developed a high-performance just-in-time compiler (JIT), called STARJIT [1], that can compile both Java and CLI applications, and a high-performance garbage collector (GC), called GCV4. Because ROTOR provides a complete platform for CLI experimentation, we set out to integrate STARJIT and GCV4 with ROTOR on the IA-32 architecture. This paper describes our experience and presents our observations on the suitability of ROTOR as a research platform.

STARJIT and GCV4 were originally developed for use with our virtual machine, ORP (the Open Runtime Platform [3]). ORP was originally designed for Java and later adapted to support CLI as well. One of ORP's key characteristics is its modularity: ORP interacts with JITs and GCs almost exclusively through well-defined interfaces.[1] We hoped the use of these interfaces by STARJIT and GCV4 would simplify our integration. ROTOR also has a well-defined JIT interface, but not one for GCs. Some of ROTOR's interfaces are defined directly in terms of internal details of the VM, but others are more abstract, using structures like handles and separating the VM cleanly from other components. Using these abstract interfaces, JITs such as STARJIT can be built independently of ROTOR itself, and loaded as DLLs (dynamically-linked libraries) at runtime.

Our ultimate goal is to see how well STARJIT's and GCV4's optimizations apply to CLI and what further

[1]The only exceptions to ORP's well-defined interfaces are its assumptions about the layouts of performance-critical data structures including object headers, vtables (virtual-method tables), and some GC information stored in vtables.

optimizations for CLI can be developed. StarJit includes advanced optimizations such as guarded devirtualization, synchronization optimization, Class Hierarchy Analysis (CHA [4]), runtime-check elimination (null pointer, array index, and array-store checks), and dynamic profile-guided optimization (DPGO). GcV4 performs parallel sliding compaction to maximize application throughput. StarJit and GcV4 can collaborate to insert prefetching based on dynamic profiles of cache misses [2]. All these optimizations are important to managed languages like Java and C#.

Overall, we found the JIT integration more straightforward because the Rotor JIT interface is well defined. In contrast, integrating the collector required many intricate changes, and these are interspersed throughout the Rotor source code. In both cases, however, we found our work complicated by missing functionality. We begin with some background about the integration effort, then describe in detail what was required for the JIT and the GC.

2. INTEGRATION OVERVIEW

A key goal of our JIT and GC integration efforts was to minimize changes to Rotor's code base. We also wanted to avoid making extensive changes to our own StarJit and GcV4 code bases.

2.1 JIT-Related Modifications

Rotor divides the compilation and management of compiled code into three components: JITs, JIT managers, and code managers. JITs compile CLI bytecodes into native code. JIT managers allocate and manage space for compiled code, data, exception-handler information, and garbage-collection information. Code managers are responsible for stack operations involving the frames of compiled code that they manage. The Rotor design is general, and there is no reason why it cannot support multiple JITs, multiple JIT managers, multiple code managers, JITs that share JIT and code managers, *et cetera*. Currently, Rotor has one JIT, two JIT managers, and one code manager.

To implement a JIT, JIT manager, or code manager, one writes a class that implements the appropriate interface. The JIT interface is designed for implementation in DLLs. It also hides the details of Rotor's types for classes, methods, fields, *et cetera*, with the use of handles such as `CORINFO_CLASS_HANDLE`, `CORINFO_METHOD_HANDLE`, and `CORINFO_FIELD_HANDLE`. On the other hand, Rotor's JIT manager and code manager interfaces use Rotor's internal data structures directly and so are difficult to place in DLLs.

We found that most of the StarJit integration effort centered around the JIT interface, which is defined in `corjit.h` and `corinfo.h`. These files define a number of interface classes, all of whose names begin with the letter `I` (*e.g.*, `ICorClassInfo`). The JIT must implement the interface class in `corjit.h` and can communicate with the VM using the interface classes in `corinfo.h`. To date, we have succeeded in using only these interface functions for method compilation, and Rotor modifications have not yet been necessary.

However, certain StarJit optimizations will require extensions to this interface. For example, CHA requires the JIT to examine the currently loaded class hierarchy to detect whether a particular method in a class has been overridden by a subclass. While Rotor's JIT interface allows exploration *up* the class hierarchy, it currently does not allow exploration *down* the class hierarchy, precluding CHA.

Because of our past experience building new JITs, we implemented support for multiple JITs in Rotor. This approach allows several different JITs to be present in the system at the same time. For each new method, the VM calls the first JIT to compile the method. If the JIT is unsuccessful the VM calls the next JIT and so on until one JIT reports success. In our implementation, we give StarJit the first opportunity to compile a method. StarJit has "method table" code that allows the user to specify which methods StarJit should compile; other methods are rejected, and compiled by FJIT. As a result, if a bug is encountered, we can gradually reduce the set of methods compiled by StarJit until we locate the single method that caused the problem. This technique of debugging a new JIT with the use of a robust backup JIT proved invaluable in our integration effort.

2.2 GC-Related Modifications

Unlike for JITs, there is no clean interface in Rotor for a garbage collector to communicate with the rest of the system. The Rotor GC is responsible for both object allocation and garbage collection, and also interacts with the threading subsystem. As such, more extensive modifications of Rotor were required for integrating GcV4.

Garbage collection problems can be notoriously difficult to debug, since a problem introduced during a collection may not manifest itself until much later. For debugging such problems, we found it useful to use built-in Rotor functionality for forcing collections at more regular intervals. Rotor has a `GCStress` parameter that can be given various settings. One setting we found especially useful forces a collection every time an object is allocated. This setting often causes garbage collection problems to show up soon after they occur, when the information needed to debug them is still available.

3. JIT COMPILE-TIME INTERFACE

As previously mentioned, a major part of the StarJit integration is adapting StarJit to Rotor's JIT interface. This adaptation includes providing the function to compile a method for Rotor and adapting StarJit to use the set of functions that Rotor provides for querying classes, fields, methods, *et cetera*.

While the StarJit integration is still under development, we have successfully compiled and run enough programs that we believe the integration is nearly complete. Despite some initial difficulty understanding the semantics of a few of Rotor's JIT interface functions, our experience has been predominantly positive. This section discusses our experience and notes the few problems we found.

3.1 Supporting the JIT Compile-Time Interface

StarJit already includes an internal interface, `VMInterface`, that it uses to isolate itself from any particular VM. The ORP version of StarJit, for example, is built with an ORP-specific implementation of this interface. The main part of our effort was spent implementing a Rotor-specific implementation of `VMInterface`.

`VMInterface` includes about 160 methods. The majority of these methods resolve classes and get information about methods, fields, and other items during compilation. One `VMInterface` method returns the address of the different runtime helpers, and is described in detail in the next section. Various StarJit optimizations are supported by other `VMInterface` methods that, for instance, return a method's "heat" (an indication of the amount of execution time spent in the method) for profile-based recompilation.

Most of the `VMInterface` implementation for Rotor was straightforward. However, the `VMInterface` implementation is not yet complete—we have not implemented specialized support for optimizations that are not presently enabled.

To support StarJit's requirements, we found two cases where it was necessary to define new data structures in the `VMInterface` implementation to augment the corresponding Rotor information. In the first case, Rotor does not provide a way for JITs to get a handle (`CorInfoHandle`) for a primitive class. Since StarJit preloads the primitive classes at startup, we defined a new `RotorTypeInfo` data structure to represent types that include enough information to describe primitive types. When, for example, StarJit passes a `RotorTypeInfo` to the `VMInterface` method `typeHandleIsUnboxed`, the latter can recognize if the `RotorTypeInfo` represents a primitive class, and in that case return `true`. In the second case, StarJit needs the type of the `this` argument for many methods. In Rotor, this type is not derivable from the signature information (`CORINFO_SIG_INFO`) for a method. Our solution is to represent a method's signature using a structure that contains both a `CORINFO_SIG_INFO` and a `CORINFO_METHOD_HANDLE`. This technique is similar to the `OpType` tuple class used in Rotor's built-in FJIT.

3.2 Experience

In summary, we found Rotor's compile-time JIT interface (`ICorJitInfo`) generally well-designed. However, some information needed for optimizations is missing. It was also necessary to work around some limitations such as the inability of a JIT to get handles for primitive classes. We have the impression that `ICorJitInfo` is narrowly defined to provide just the functionality needed for FJIT. While this makes the interface simple, it complicates adding new, more optimizing JITs to Rotor.

The `ICorJitInfo` class inherits from a number of abstract superclasses that each define functions in various areas of compile-time information (e.g., methods, modules, fields) and areas of runtime information (e.g., helper functions and profiling data). We expect to add support for our optimizations by adding a new superclass. This will contain, for example, methods to get class hierarchy and profile-based recompilation information.

The lack of documentation about Rotor's internals was another obstacle. While the book, Shared Source CLI Essentials [9], is a great help, too often we resorted to experimentation to discover what Rotor functions to use. To be more widely successful as a VM intended for research, Rotor needs better documentation.

4. JIT RUNTIME INTERFACE

This section describes the runtime support needed to integrate StarJit into Rotor. Besides the compile-time cooperation described earlier, StarJit and Rotor must also cooperate at runtime. For example, although StarJit generates code for managed methods, StarJit relies on Rotor for such VM-specific issues as object allocation. Similarly, the Rotor VM handles stack unwinding and root-set enumeration but it relies on the JIT to interpret a given stack frame.

4.1 Helper Calls

The JIT-compiled code of StarJit and Rotor's FJIT both rely on helper calls to perform VM-specific operations (*e.g.*, to allocate objects, throw exceptions, do `castclass` or `isinst` operations, and acquire or release locks) and, in some cases, to perform common complex operations (*e.g.*, 64-bit operations on a 32-bit architecture). Rotor provides a mechanism to query for helpers in its `ICorInfo` interface. StarJit's Rotor-specific `VMInterface`, in turn, maps StarJit helpers to Rotor ones. During our integration work, we encountered several issues specific to helper calls. In most cases, we were able to solve these issues within the Rotor-specific `VMInterface` layer.

The first issue we encountered involved the different calling conventions used by ORP and Rotor. StarJit had been hardwired to use the ORP conventions when calling VM helper functions as well as other managed code. To modify StarJit to use Rotor's calling conventions, an `#ifdef` was used to control the conventions it employs.

A second issue we discovered involved differences in both the required parameters and their order for different helpers. For example, ORP's `rethrow` helper requires the exception as a parameter but Rotor's does not. In addition, ORP and Rotor's `castclass` helpers have the object and type descriptor in different orders. We considered the use of wrapper stubs to convert between one set of conventions and the others. However, these wrappers complicate stack unwinding and incur additional performance overheads. We instead modified StarJit to use Rotor's conventions.

There are a couple of differences between Rotor and StarJit related to type-specific helpers. A number of helpers, including the ones for object allocation, type checks, and interface table lookups, involve types that are known at compile time. In these cases, Rotor returns different helpers for different types, based on a type passed in at compile time. Accordingly, we modified the helper function lookup in StarJit's `VMInterface` to require a type for all type-related helpers. For any VM (*e.g.*, ORP) where the type is not required, that VM's StarJit`VMInterface` imple-

mentation ignores the type. There are also differences in exactly which of several type-related data structures are passed at compile time or runtime to these helpers. We abstracted this detail into `VMInterface` so that the VM-specific code can give StarJit the correct data structure to pass.

Another challenge involved helpers that StarJit expected that were not provided by Rotor. In most cases, these were helpers for 64-bit integer operations (*e.g.*, shifts) not provided by Rotor. In these cases, the helper could easily be implemented within the Rotor-specific `VMInterface`. Some other cases reflect a more serious mismatch between StarJit and Rotor. For example, Rotor provides an `unbox` helper that performs the necessary type check on a reference and then unboxes it. In StarJit, however, the type check and the actual unbox are broken into separate operations at an early point with the hope of statically removing the type check via optimization. StarJit expects a helper to perform the unbox-specific type check but generates a simple address calculation to do the actual unboxing. Rotor, on the other hand, only provides a helper to perform the entire unbox. For now, we use the `castclass` helper instead to perform the unbox type check. However, this approach fails when the unboxed reference is a boxed enumeration type and will have to be corrected.

Finally, there are a number of helpers that Rotor provides that are not currently invoked by StarJit. Some of these additional helpers are provided only to simplify portability (without them, Rotor's FJIT would need IA-32-specific and PowerPC-specific assembly sequences). Other helpers assist in debugging, while still more support additional functionality such as remoting. Up to this point, none of the applications that we have tried to execute with StarJit have needed the additional functionality provided by these helpers. However, in the future, we plan to extend ORP's `VMInterface` to enable StarJit to query the VM and discover which of these additional helper functions must be called.

4.2 Code and JIT Managers

As part of our implementation of multiple JIT support, we found we needed to use the other JIT manager in Rotor. We could not use a second instance of FJIT's JIT manager because its implementation uses global variables to, for example, map program counters to methods and to manage memory. Two instances would have conflicting uses of these variables.

Another part of the runtime interface concerns stack walking activities such as root-set enumeration, exception propagation, and stack inspection. The Rotor design, like many other VMs, divides this task into one part that loops over the stack as a whole and another part that deals with individual stack frames. The loop part is in the VM proper and rightly so. Conversely, processing an individual stack frame depends upon the JIT's stack conventions (*e.g.*, the location of callee saves registers and where local and temporary variables of reference type are located) and therefore requires the JIT's cooperation. In Rotor, all processing of individual stack frames is done by the code manager.

The code manager that comes with Rotor makes many assumptions about JIT-compiled code:

- The code for each method is expected to consist of a prologue, followed by the body, followed by an epilogue.
- Multiple epilogues and epilogues interspersed in the body are not allowed.
- The prologue and epilogue are precisely defined code sequences, no deviations are allowed.
- Only `ebp` and `esi` are saved and available for use; `ebp` is used as a frame pointer, while `esi` is always a valid object reference (but possibly NULL). Registers `ebx` and `edi` may not be used.
- The security object is at address `ebp-8`.
- JITs give root-set information to the JIT manager in the form of an *info block*, which the JIT manager then passes to the code manager during root-set enumeration. This information is expected to match the particular structure of Rotor's JIT.

These assumptions of Rotor's code manager fundamentally conflict with those of StarJit. We therefore decided to write our own code manager. This code manager has to be part of the VM, but we decided to try emulating Rotor's interaction with the JIT by having this new code manager simply convert all its calls into calls to a runtime manager placed in the same DLL as the matching JIT. We defined an interface along the lines of `corjit.h`, and allow a DLL to export a runtime manager as well as a JIT. This approach was mostly straightforward.

However, the parameters passed to different code manager methods are inconsistent. For example, the method `UnwindStackFrame` gets an `ICodeInfo` object, which can be used to identify the method and some of its attributes, but `FixContext` does not. Also, these methods need to know the current values of registers for the frame that they are unwinding, fixing up, or enumerating the roots of, and there are different types of contexts for `FixContext` versus `UnwindStackFrame` and most of the other methods. We decided to reflect these inconsistencies in the external interface. Since StarJit's runtime interface is more uniform and requires the method handle for the method of the frame, we used the info block to pass the missing information from compile time to run time.

Another minor point is that `UnwindStackFrame` is sometimes called with the context `esp` equal to the address just above the arguments of the out-going call and sometimes equal to the lowest address of the outgoing arguments. In general, there is no way to tell which of the two cases holds. This situation is fine if frame pointers are used; the context `ebp` can be used to find everything in the frame. However, requiring frame pointers on IA-32 reduces the number of usable registers from 7 to 6. For now, we have modified StarJit to use frame pointers.

4.3 Exception Handling

Another significant difference between ROTOR and STARJIT concerns the details of exception propagation. Here, the differences stem directly from the characteristics of CLI and Java. In CLI, there are exception handlers, filters, finally blocks, and fault blocks. Each of these is a separate block of bytecode from the region being protected, and control cannot enter these blocks except through the exception mechanism. Conversely, in Java, there are only exception handlers and these protect a region of bytecode. When an exception is caught in Java, control is transfered to a handler address which can be anywhere in the method's bytecode.

Since STARJIT was developed against the interfaces of ORP, which originally supported Java and was later adapted to also support CLI, STARJIT's design reflects the Java exception mechanism. First, STARJIT implements finally and fault blocks by catching all exceptions and then rethrowing them. This behavior is close to but not exactly that required by the CLI specification, although it is correct for code compiled from C#. Second, there is a particular bytecode for leaving an exception handler and returning to the "main" code (a `leave`). ROTOR requires the JIT at such a bytecode to call the runtime helper `EndCatch`. This helper cleans up stack state generated by the VM for exception handling and ensures that finally blocks are called. We modified STARJIT to call this helper since ORP does not have a corresponding helper. Finally, ROTOR needs an exception handler to be compiled to a contiguous region of native code and it needs to know the start and end addresses of that region. STARJIT knows the start address, but not the end address, and might rearrange blocks so that a handler is no longer contiguous. We do not have a solution for this problem yet. For now, we give a zero end address—this causes ROTOR to compute incorrect handler nesting depths, but otherwise seems to have no ill effect.

4.4 Experience

ROTOR should include better support for multiple JITs. We had to modify ROTOR to try more than one JIT. We also had to add a second JIT manager, and to write our own code manager. ROTOR would be better if JIT managers and code managers could be packaged with JITs in a separate DLL, and if these could interact with the VM through an abstract interface such as those in `corjit.h` and `corinfo.h`. Furthermore, the code manager functions should have a more consistent set of parameters to make for a more uniform interface.

Our experience integrating STARJIT with ROTOR also led to changes in STARJIT. For example, STARJIT's `VMInterface` had to be generalized to better support requests for type-specific helpers. We also found that STARJIT should allow calling conventions to be specified by the VM. Currently, we use `#ifdef`s in STARJIT's source code to control calling conventions, but this makes the code hard to maintain and the resulting code less flexible. If STARJIT queried the VM about the calling conventions to use, it could adapt itself dynamically to the needs of the VM. Also, the design of a clean and flexible runtime helper interface is an interesting problem, and one we would like to address.

5. GC INTEGRATION

ROTOR does not have an explicit, cleanly-defined GC interface that resembles its JIT interface. ROTOR also does not support the dynamic loading of garbage collectors from DLLs. As a result, to integrate our GCV4 garbage collector into ROTOR, we added GCV4 directly to the ROTOR VM code base. Much of our effort involved adapting GCV4 to run in ROTOR and reconciling the different assumptions made by ROTOR and GCV4. This section discusses our experience integrating this collector, including the issues we encountered and our solutions.

Probably the most significant issue we found was that ROTOR exposes the implementation of its collector to other components in the system. For example, ROTOR's `GCHeap` class reveals that ROTOR uses a generational collector that treats large objects differently than small ones, and allows clients to query whether an object is part of the ephemeral generation. Much of this is likely to change if ROTOR's GC is replaced with another GC. As another example, the ROTOR VM uses knowledge about the collector's implementation to allow JITs to emit optimized code. The VM's function `JIT_TrialAlloc::EmitCore` can be called by JITs to emit code for the allocation fast path for many types of objects. That code assumes intimate knowledge of the GC's data structures and object-allocation strategies.

The ROTOR VM requires that the garbage collector export a number of functions. We modified ROTOR to invoke GCV4's functions instead of the corresponding ROTOR ones. The ROTOR VM now calls the GCV4 initialization function and object allocator instead of the ROTOR equivalents. We also modified ROTOR's thread constructors and destructors to keep GCV4 up-to-date with respect to thread existence. Finally, we modified `JIT_TrialAlloc::EmitCore` to no longer make assumptions about the collector's data structures. The code it generates currently directly invokes the "slow" allocation function, which we modified to call GCV4's allocator. We intend to add fastpath allocation back into the generated code, but we hope to develop an interface that will allow this to happen in a generic way to support other collectors in the future.

Similarly, GCV4 expects the VM to supply a number of functions. One especially important function, used at the start of a garbage collection, requests that the VM stop all threads and enumerate all roots. Since stopping (and restarting) threads in ROTOR requires a very specific sequence of events, we reused much of the existing ROTOR code for this purpose. We also reused the two `CNameSpace` methods `GcScanRoots` and `GcScanHandles` to do root-set enumeration by passing them our own GCV4 callback function instead of ROTOR's one.[2]

[2]In fact, `CNameSpace::GcScanHandles` ignores its callback parameter, but we modified the two functions it calls to invoke our callback function instead.

5.1 Integration Issues and Solutions

In the course of our integration, we found a number of conflicts between the assumptions made by GCV4 and ROTOR about the layout of several key data structures. These are listed below along with our solutions.

- *Object Layout.* Since GCV4 was originally developed for ORP, GCV4 expected objects to use ORP's memory layout. Moreover, GCV4 assumed that each object began with a pointer to the vtable, followed immediately by ORP's multi-use `obj_info` field. This field holds synchronization, hash code, and garbage collection state, and so resembles ROTOR's "sync block index." However, ROTOR places other object data at a four byte offset while ROTOR expects the sync block index to be at a four byte *negative* offset from the start of an object. Realistically, too many parts of ROTOR depend on this layout to change it. Also too many parts of ROTOR use the sync block index in ways incompatible with GCV4's use of the `obj_info` field, so mapping `obj_info` to the sync block index is not a solution.

  Our solution was to place the ORP `obj_info` field before each object, at a negative eight offset from the object's vtable pointer. This offset does not conflict with any part of ROTOR's object layout. As a result, no ROTOR component is aware of the extra field.

- *Vtable Layout.* GCV4 assumed that the first 4 bytes of each vtable is a pointer to a structure containing GC-related information that indicates, for example, whether the object contains pointers and if so, the offset of each pointer. The start of ROTOR's `MethodTable` structure contains the component size (for array objects and value classes), the base size of each instance of this class, and a pointer to the corresponding class structure (`EEClass`). There are many places in ROTOR that assume specific offsets to these fields, so changing the field layout would raise many problems.

  We also could not store the pointer at a negative offset from the start of the vtable. That would interfere with ROTOR's `CGCDesc` and `CGCDescSeries` structures, which are stored before the vtable if the class contains pointers. These structures are used by FJIT as well as ROTOR's collector, so we could not use that space for our pointer.

  We solved this by reserving space in ROTOR's `MethodTable` class at a sufficiently high offset to avoid conflicts with ROTOR's fields.

- *Thread Layout.* GCV4 assumed that a portion of each thread's data structure is storage reserved for its use, which is is an essential part of ORP's object allocation and garbage collection strategies. However, ROTOR does not have an analogous field in its thread data structure. Our solution was to add the extra storage at the end of the thread objects.

5.2 Experience

ROTOR should include a GC interface that resembles its JIT interface. That is, ROTOR should use functions to abstract the interactions between the collector and other ROTOR components. These functions would hide details about the collector's implementation and help to make explicit the assumptions it makes. Such an interface would make it easier to modify the collector and to experiment with new implementations without affecting other components. For example, ROTOR's GC interface could include a function that returns the offset of its sync block index. This would avoid other components assuming a fixed constant for that value. Our experience with ORP's GC interface has been strongly positive, and it has allowed us to use several different collector implementations without changing its VM or JITs.

To enable easier GC experimentation, it would help if ROTOR's GC could be dynamically loaded like its JITs. New collectors could be plugged in to ROTOR including ones tailored for particular needs, such as when an application needs high throughput more than short GC pause times. Changing ROTOR to dynamically load its GC would also help to minimize assumptions made by the VM or other components.

6. STATUS AND FUTURE WORK

When we started our integration work, we wondered how suitable ROTOR would be as a research platform, that is, how difficult would it be to add our optimizations and what changes to ROTOR would be needed to support them. Our plans were initially to add STARJIT and GCV4, then later implement in ROTOR a number of optimizations such as our synchronization techniques, prefetching, and DPGO. This paper described the approaches we took to integrate STARJIT and GCV4, and our experience with that effort.

The STARJIT integration was straightforward except for a few issues. While most of the needed changes were within STARJIT, we found that we had to modify ROTOR to add support for multiple JITs and to add a new code manager for STARJIT. We also needed to support another JIT manager in ROTOR. This is because we could not create another instance of FJIT's JIT manager since its implementation depends on global variables. Although ROTOR allows JITs to be loaded dynamically, and communicates with those JITs using its abstract JIT interface, ROTOR does not allow JIT or code managers to be loaded dynamically. Adding new code or JIT managers requires modifying ROTOR itself, although abstract interfaces for these managers could be added to ROTOR without much trouble. Later, we expect to add support for some of the more sophisticated STARJIT optimizations such as DPGO by augmenting ROTOR's JIT interface with a new abstract superclass that defines the required functions.

We found that adding a new garbage collector to ROTOR was much more difficult than integrating a new JIT. ROTOR does not have a clean interface for GCs that resembles its JIT interface. Its `GCHeap` class, for example, exposes details about the GC's implementation that are used by several other parts of the system

including FJIT, so adding a different implementation required changing those parts. We tried to minimize the changes to ROTOR, but a number of changes were needed, for example, to have ROTOR call functions in the GC interface that GCV4 exports. Both ROTOR and GCV4 make assumptions about the layout of objects and virtual-method tables, so it was necessary to modify our GCV4 implementation to place the fields that GCV4 needs (such as one used to hold a forwarding pointer during collections) in locations that do not conflict with fields required by ROTOR.

Our work integrating STARJIT and GCV4 with ROTOR is ongoing. We can run a number of test programs and are currently getting our modified ROTOR to work with the C# version of the SPEC JBB2000 [8] benchmark. Our plans for STARJIT include adding support for pinned objects and full support for CLI exceptions (such as filters), as well as support for our optimization technologies such as DPGO and prefetching. Similarly, we will add support to GCV4 for managed pointers. We are optimistic about being able to complete this work and look forward to exploring other opportunities for improving ROTOR's performance.

7. REFERENCES

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7(1), February 2003. Available at `http://intel.com/technology/itj/2003/volume07issue01/art02_starjit/p01_abstract.htm`.

[2] A.-R. Adl-Tabatabai, R. Hudson, M. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *SIGPLAN Conference on Programming Language Design and Implementation*, Washington, DC, USA, June 2004.

[3] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003. Available at `http://intel.com/technology/itj/2003/volume07issue01/art01_orp/p01_abstract.htm`.

[4] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, Aug. 1995. Springer-Verlag (*LNCS* 952).

[5] ISO/IEC 23270 (C#). ISO/IEC standard, 2003.

[6] ISO/IEC 23271 (CLI). ISO/IEC standard, 2003.

[7] Microsoft. Shared source common language infrastructure. Published as a Web page, 2002. See `http://msdn.microsoft.com/net/sscli`.

[8] Standard Performance Evaluation Corporation. SPEC JBB2000, 2000. See `http://www.spec.org/jbb2000`.

[9] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly, Mar. 2003.

Active C#

Raphael Güntensperger Jürg Gutknecht

ETH Zürich
Clausiusstrasse 59
CH-8092 Zürich

guentensperger@inf.ethz.ch gutknecht@inf.ethz.ch

ABSTRACT

Active C# is a variant of Microsoft's C# that enhances the basic language with a direct support for concurrency and a new model for object communication. The C# compiler of the Shared Sources Common Language Infrastructure (SSCLI) served as a basis to extend the compiler. Modifications mainly concern the enhancement of C# with an active object concept and a novel communication paradigm based on formal dialogs.

Keywords

Active C#, Programming Languages, Concurrency, Formal Dialogs, Active Objects, AOS, SSCLI

1. BACKGROUND

The roots of Active C# can be found in a ROTOR project partially funded by Microsoft Research [Gu]. The concept of active objects and their synchronization comes from Active Oberon [Gk], a successor of the Oberon Language and from the Active Object System [Mu], an internally developed operating system microkernel. This paper presents a consolidation and enhancement of an experimental language concept introduced in the aforementioned ROTOR project.

2. OVERVIEW

From a historical perspective, we can easily recognize an evolution of the object concept from purely passive *data records* to re-active, *functional entities*. In our language experiment, we evolve the object concept another step further by adding *encapsulated behavior* and *communication capabilities*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2004 *workshop proceedings,*
ISBN 80-903100-4-4
Copyright UNION Agency – Science Press, Plzen, Czech Republic

Active C# is an extension of C# which mainly includes two new technologies: *active objects* and *formal dialogs*.

Both technologies support the seamless integration of threading into the programming model, with the aim of increased acceptance and use of concurrency in programs. The idea is that programmers do not need to call the underlying threading framework directly anymore but can still add concurrency to their programs simply by making appropriate use of the programming model.

Active Objects

An active object is an instance of a class with encapsulated behavior, running one or more separate threads.

In Active C#, this idea is supported by *activities*. a new kind of class members. An activity is a method with an empty parameter list and void result, run as a separate thread. Any number of activities are allowed in a class.

Two kinds of activities exist: unnamed and named. An *unnamed activity* automatically starts after object instantiation and is executed only once per instance, where a *named activity* must be started explicitly and can be executed any number of times. The `static` modifier is also allowed for both kinds of activities and, if chosen, the activity is bound to the type of the object rather than to its instance. This implies that a

static unnamed activity is started when the class is loaded and that these activities run in the static context of the class.

Formal Dialogs

Formal dialogs are a vehicle that allows advanced, syntax-controlled communication between objects, notably between remote objects. Thus, a formal dialog serves as a communication interface from the outer world to an object.

Assuming that some object *b* provides a syntactic specification of dialog D and that object *a* wants to communicate with *b*, this is how it works:

- *a* instantiates dialog D with *b*
- *b* creates a separate thread of control, acting as a symbolic channel of communication between *a* and *b*
- *a* and *b* communicate over a symbolic channel according to the syntactical specification of D

In Active C#, dialog interfaces and implementations are represented by keyword enumerations and parser activities respectively. For example:

```
dialog D { u, v, w }
// keywords

class B {
  …
  activity d: D { … }
  // parser
}
```

3. ACTIVITIES

An activity is defined inside a class and follows this syntax:

["static"] "activity" [name] "{" statements "}".

We recall that unnamed activities are launched automatically at instance creation time. In contrast, named activities must be started explicitly by calling an overloading of the `new` operator:

"new" QualIdent.

By concept, activities always run to their end, and there is no explicit option of aborting an activity. Of course, each activity can still decide to finish its work early depending on some state condition.

Activities are inherited from base classes and run in parallel with activities defined in the derived class. Inherited anonymous activities are therefore started automatically at each instance creation of a derived class.

The automatic starting of threads belonging to anonymous activities is handled by our modified compiler at the end of either the instance constructor or the type constructor.

Synchronization

Normally, object activities run in full concurrency. However, sometimes a certain precondition is needed for continuation. In line with our goal of simplifying concurrent programming and avoiding explicit calls to the threading framework, we use a direct

```
await (condition)
```

statement instead of signals for the reactivation of waiting activities. `condition` is an arbitrary Boolean expression representing the condition to be waited for.

To ensure proper synchronization, the await statement must occur in a context that is locked with respect to the enclosing object and it must refer to a purely object-local condition. In Active C#, we use an overloading of the `lock` statement for this purpose:

```
lock { }
```

whose semantics is given in Figure 1, where *context* refers either to the current object (`this`) or its type descriptor (`typeof(Class)`) and `Monitor` refers to the corresponding type of the .NET Framework's threading library.

```
Monitor.Enter(context);
try {
        // statements
}
finally {
        Monitor.PulseAll(context);
        Monitor.Exit(context);
}
```

Figure 1. The `lock` construct decomposed

The `lock` statement simplifies both the specification of context-locking actions and the implementation of the Active C# compiler. Assuming that object state-changes occur within context-locked sections only, it is reasonable to

- map `await (condition)` to
  `while (!condition)`
  `  Monitor.Wait()`
- generate a
  `Monitor.PulseAll (context)`
  at the end of each lock block

However, interestingly, this is not sufficient. Another `PulseAll (context)` is necessary right before

an unsatisfied await statement suspends its thread for the first time[1]
In summary, all this leads to the decomposition of the await statement shown in Figure 2.

In principle, time-oriented conditions could be handled by await statements of the form

```
await (t >= T)
```

within some *Timer* object. However, for convenience, a special *passivate* statement is provided for this purpose. This is its form and semantics:

```
passivate (duration);
```

where the `duration` parameter specifies the number of milliseconds the current thread is to be suspended. The passivate statement can occur at any places in the code and takes any integer expression as argument.

4. DIALOGS

Our dialog model is based on *formal grammars* that constitute some kind of contract between caller and callee. An element of such a grammar is called a *token*. Each token basically specifies a data type and a direction. Our implementation of formal dialogs associates two buffers with each communication. Tokens sent by the caller are stored in the *input buffer* to be processed by the callee. Conversely, tokens sent by the callee are stored in the output buffer to be processed by the caller. Technically, both buffers are instances of `System.Dialog.DialogBuffer` and implemented as self-expanding ring-buffers.

Encoding and decoding

Because dialogs are designed to be used in remote environments as well, an encoding must be specified and agreed upon for each token type, and a *codec* must be plugged into the sender and receiver program respectively. This system works because the token buffers act as FIFO-queues and therefore allow their contents to be treated as a byte stream.

The current codec supports the C# built-in types `int`, `long`, `float`, `double`, `bool`, `char`, `string`, `byte`, `byte[]`, the new Active C# type `keyword` and an `escape` type used in some formal grammars.

[1] Before suspending a thread after checking the condition of its await statement at all later times, no signal is necessary, because this thread had no possibility to change any condition in the meantime.

```
bool waitingAlready = false;
while(!condition)
{
        if(!waitingAlready)
        {
                Monitor.PulseAll(ref);
                waitingAlready = true;
        }
        Monitor.Wait(ref);
}
```

Figure 2. Decomposition of the await statement

Dialog specification and implementation

A *dialog specification* is an element of a namespace (on the same level as classes and interfaces) and has the following syntax:

> [accmod] ”dialog” DialogTypeName keywords.
> accmod = private | internal | protected | public.
> keywords = ”{” [{ keyword ”,” } keyword] ”}”.

This declaration defines the *dialog type*, including the list of keywords of the underlying grammar. User defined dialog types are always implicitly derived from `System.Dialog.Dialog`, a predefined type that specifies the dialog accessors (see next section) and some references to internal ingredients of a running dialog, such as its buffers.
All keywords are of the new built-in type `keyword`, mapped to the enumeration type `System.Dialog.Keyword`. Their values are used by the sender and receiver, which guarantees an efficient transfer of keyword tokens.
Note that dialog types have a comprehensive character and provide the following infrastructure:

- An enumeration type for keywords
- An interface for a dialog implementation
- The data structure to control a running dialog

A *dialog implementation* is a named activity that implements the corresponding dialog specification. The syntax is familiar from interface implementation:

> [“static”] “activity” ActivityName
> “:” DialogType “{“ statements “}”.

Note that, in the case of activities, a formal syntax consistently replaces the argument list occurring in method declarations. We will use the C# attribute concept to bind a formal syntax to a dialog declaration. An automatic parser generator, which we are implementing in a related project, may read this syntax to produce an appropriate parser.

Dialog operators

In Active C#, four dialog-related operators exist: `new, ~, !, ?` and `??`. In turn, their meaning is create a new dialog instance, close a dialog instance, send a token and receive a token in blocking and unblocking mode respectively.
Not surprisingly, the Active C# compiler and runtime depend on powerful library support for the implementation of dialogs, especially for remote dialogs (see the corresponding section below). We already mentioned the types `System.Dialog.Dialog` and `System.Dialog.DialogBuffer`.
These are the library methods that correspond to the Active C# operators:

- **`constructor`** instantiates a dialog and returns a reference to the instance
- **`close`** explicitly discards a dialog and stops its associated thread
- **`send`** takes an object, encodes it and passes the encoded data to the input buffer
- **`receive`** tries to decode the output buffer and returns an object

The `receive` accessor can be called in two modes. In *blocking* mode, control is given back to the caller only after a complete object has been received, where in the *non-blocking* mode the accessor immediately returns control, however with a possible `null` return value if not enough bytes were available to decode a complete object at the time of invocation.
Two variants `put` and `get` of `send` and `receive` are used within the callee class. They take the dialog reference directly from the thread context and are in-lined by the compiler directly into the parser code. While the accessor methods work with the general `object` type, the compiler automatically casts the received object to the type of the target variable.

Dialog lifecycle control

Activities are launched in Active C# simply by calling their name, qualified by a reference to the object instance or class name (in the case of static activities). In the special case of *dialog* activities, a reference to the launched activity is needed in sending and receiving operations. For this reason, an overloading of the new operator is provided:

ref = ”new” TypeOrRef ”.” ActivityName.

where `TypeOrRef` is the name of the type for a static dialog or a reference to the callee respectively. Note that the reference returned by `new` refers to one specific instance of a dialog and is necessary to specify the context of the communication. Internally (that is, on the callee side) it is registered relative to the activity thread descriptor and loaded in a local variable at the beginning of each method which might potentially make use of it[2], thus the programmer does not have to refer to it explicitly. In this way, the reference to the current dialog instance is available even across method calls. The caller can discard the current instance of a dialog explicitly by calling its destructor:

”~” ref.

Any further access to this dialog would raise an exception.
When the dialog activity terminates regularly, the corresponding thread is discarded and no further communication is possible, although the reference to the dialog instance remains valid.

Communication

The *send* and *receive* operators are designed to take generic arguments of type `object`. Received objects are type-checked and cast back to their actual type. Table 1 shows the communication syntax. *d* denotes a reference to the current dialog and *obj* is the token to be exchanged. On the callee side, the reference to the dialog is implicit.
The use of separate buffers for input and output allows a full-duplex data-flow. The buffer size is increased automatically on demand but can be limited on desire. If the input buffer is full, the next send operation blocks.

Action	By client	In parser context
Send	d!obj;	!obj;
Receive (blocking)	d?obj;	?obj;
Receive (non-blocking)	d??obj;	??obj;

Table 1: Active C# communication syntax

An example

The communication mechanism supported by Active C# really shines when it comes to “stateful” dialogs such as, for example, negotiations. An upgraded version of John Trono’s Santa Claus concurrency exercise [Tr] may illustrate this.

The original version goes like this: Santa Claus sleeps at the North Pole until awakened by either all of the nine reindeer, or by a group of three out of ten

[2] Each method which contains at least one send or receive statement is marked appropriately

```
c = new Coordinator.CoordElves;
while (true) {
   passivate(Christmas.Rnd());
   c!CoordElvesDialog.join;
   c?msg;
   if (msg == CoordElvesDialog.wait)
      if ((Christmas.Rnd() % 3) == 0)
         c!CoordElvesDialog.release;
      else c!CoordElvesDialog.join;
   }
 }
```

Figure 3. Behavior of an elf

elves. He performs one of two indivisible actions: If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on vacation. If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys. A waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santa's time is extremely valuable, marshalling the reindeer or elves into a group must not be done by Santa.

The following complication now adds an element of negotiation: If complete groups are waiting for Santa when an elf desires to join, she should be given the option of withdrawing and walking away. Also, if one and the same elf desires to join excessively often, the coordinator should reject her.

While the translation of the original Santa scenario into an elegant C# program is easy, the negotiation added provides a bigger challenge, mainly because no appropriate language construct is readily available. However, using the dialog construct of Active C#, the following solution of uncompromised elegance is straightforward.

Figure 3 shows the behaviour of an elf while Figure 4 depicts the coordinator activity which is the dialog partner of the elf. Note the negotiation which takes place between the two participants.

It is perhaps interesting to compare our full C# program in the Appendix with Ben-Ari's carefully crafted solution [Be] in Ada95 [Ad], albeit without the complication of negotiation.

Remote dialogs

Up to this point, we have concentrated our discussion on dialogs in local contexts, which allows us to refer to callee objects and dialog instances directly via memory references. However, the communication concept is by no means limited to local environments. The two basic upgrades needed to enable remote dialogs are:

```
while (true) {
  ?msg;
  if (eBuild <= groupNo + 2)
    !CoordElvesDialog.reject;
  else {
    if (eGo < eBuild) {
       !CoordElvesDialog.wait;
       ?msg; }
    if (msg ==
       CoordElvesDialog.join) {
       lock { groupNo = eBuild;
          eSize++;
          if (eSize ==
             Christmas.reqElves)
           {  eSize = 0; eBuild++;}
          await (eGo > groupNo);
       }
       !CoordElvesDialog.release;
    }
  }
}
```

Figure 4. Behavior of the elf coordinator

- Use GUIDs instead of memory references for the identification of both the callee object and the current dialog
- Adjust the supporting dialog libraries to make them work on top of some suitable transport layer

See [Gu] for more details.

Summary

We have presented an enhanced variant of C# called Active C#, featuring a new kind of class members called *activity*. Activities provide a uniform tool for two different purposes: specification of active behavior of objects and implementation of dialogs. The rationale behind is a new object model centered around interoperating active objects, in contrast to passive objects that are remote-controlled by threads. Important advantages of the new model are integrated threading and compatibility with remote object scenarios.

While our first experiments with active objects were based on our proprietary language Active Oberon (one activity per object, no dialogs), the ROTOR Shared Source initiative and the availability of the C# compiler in source form (written in C++) allowed us to go a significant step further. The resulting Active C# compiler is fully functional and available [Ac].

Acknowledgement

We gratefully acknowledge the opportunity of developing and implementing our ideas, given to us by Microsoft Research in the context of the ROTOR project.

REFERENCES

[Ac] Active C# Compiler, http://www.cs.inf.ethz.ch/~raphaelg/ACSharp/

[Ad] Intermetrics, Inc., Ada 95 Reference Manual, ISO/IEC 8652:1995.

[Be] M. Ben-Ari, How to solve the santa claus problem. Wiley & Sons, 1997.

[Gk] J. Gutknecht, Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon, JMLC 97, p. 207-220.

[Gu] Güntensperger Raphael, Jürg Gutknecht: Activities & Channels, IEE Proceedings, Volume 150, October 2003.

[Ho] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[Mu] Muller Pieter Johannes: The Active Object System – Design and Multiprocessor Implementation, Diss. ETH No. 14755, 2002.

[Re] P. Reali, Structuring a Compiler with Active Objects, JMLC 2000, p. 250-262.

[Tr] John A. Trono: A New Exercise in Concurrency, ACM SIGCSE Bulletin, Volume 26, #3, September 1994.

Appendix

Sample Active C# Program: Santa Claus++ (Original by John Trono [Tr])

Santa Claus sleeps at the North Pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves. He performs one of two indivisible actions: If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on vacation. If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys. A waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santa's time is extremely valuable, marshalling the reindeer or elves into a group must not be done by Santa.

Complications: If complete groups are waiting for Santa when an elf desires to join, she should be given the option of withdrawing and walking away. Also, if one and the same elf desires to join excessively often, the coordinator should reject her.

```
using System;
using System.Dialog;

namespace SantaClaus
{

    // dialog declarations
    dialog CoordReindeerDialog { join, release }
    dialog CoordElvesDialog { join, reject, wait, release }
    dialog ActivSantaDialog { deliver, consult, done }

    class Reindeer {
        // an unnamed instance activity -> starts when object is instantiated
        activity {
           object msg;
           // create a new dialog instance
           CoordReindeerDialog c = new Coordinator.CoordReindeer;
           while (true) {
               passivate(Christmas.Rnd()); // wait for a random time
               c!CoordReindeerDialog.join; // send the keyword 'join'
               c?msg; // receive whatever is sent
           }
        }
    }

    class Elf {
        activity {
            keyword msg; // a variable of the special type 'keyword'
            CoordElvesDialog c = new Coordinator.CoordElves;
            while (true) {
```

```
            passivate(Christmas.Rnd());
            c!CoordElvesDialog.join;
            c?msg;
            // Note: automatic casting to the target type is done
            // by the compiler
            if (msg == CoordElvesDialog.wait)
              // the elf has to decide by her own what she wants to do now…
              if ((Christmas.Rnd() % 3) == 0) c!CoordElvesDialog.release;
              else c!CoordElvesDialog.join;
        }
    }
}

class Santa {
    const int consultTime = 10, deliverTime = 20;

    static activity ActivSanta : ActivSantaDialog {
        keyword msg;
        while (true) {
            ?msg;
            if (msg == ActivSantaDialog.deliver) {
               Console.WriteLine("Santa delivering toys");
               passivate(deliverTime);
            }
            else {
               // if it is not 'deliver' it must be 'consult'
               Console.WriteLine("Santa consulting");
               passivate(consultTime);
            }
            !ActivSantaDialog.done;  // send the keyword 'done'
        }
    }
}

class Coordinator {
    static int rGo = 0, rBuild = 0, rSize = 0;
    static int eGo = 0, eBuild = 0, eSize = 0;

    static activity CoordReindeer : CoordReindeerDialog {
        object msg;
        int groupNo;
        while (true)
        {
            ?msg;
            // sections with state changes and await statements must be locked
            lock {
                groupNo = rBuild; rSize++;
                if (rSize == Christmas.reqReindeer) {
                    // this group is full, prepare to build a new one
                    rSize = 0; rBuild++; }
                // wait until this group of reindeers comes back
                // from delivering
                await (rGo > groupNo);
            }
            !CoordReindeerDialog.release;
        }
    }

    static activity CoordElves : CoordElvesDialog {
        keyword msg;
        int groupNo = -9999;
        while (true)
        {
            ?msg;
            // an elf is not allowed to join too often
```

```
                if (eBuild <= groupNo + 2) !CoordElvesDialog.reject;
                else {
                    if (eGo < eBuild) {
                        // complete groups are already waiting for santa
                        // let the elf decide to join or to leave
                        !CoordElvesDialog.wait; ?msg;
                    }
                    if (msg == CoordElvesDialog.join) {
                        lock {
                            groupNo = eBuild; eSize++;
                            if (eSize == Christmas.reqElves) {
                                // this group is full, prepare to build a new one
                                eSize = 0; eBuild++; }
                            await (eGo > groupNo);
                        }
                        !CoordElvesDialog.release;
                    }
                }
            }
        }


        static activity {
            object msg;
            ActivSantaDialog c = new Santa.ActivSanta;
            while (true)
            {
                lock {
                    await ((rBuild > rGo) || (eBuild > eGo));
                }
                if (rBuild > rGo) {
                    c!ActivSantaDialog.deliver;
                    c?msg;
                    // the state change of this variable has to appear in a locked
                    // section in order to be recognized by an await statement
                    lock { rGo++; }
                }
                else {
                    c!ActivSantaDialog.consult;
                    c?msg;
                    lock {eGo++; }
                }
            }
        }
    }

    public class Christmas {
        public const int nofReindeer = 9, reqReindeer = 9;
        public const int nofElves = 10, reqElves = 3;
        static Random rnd = new Random();

        public static int Rnd () { return rnd.Next(1000); }

        static void Main() {
           for (int i = 0; i < nofReindeer; i++) new Reindeer ();
           for (int i = 0; i < nofElves; i++) new Elf ();
           new Santa ();
        }
    }
}
```

Alternative protection systems for OO Environments: Capability-Based Protection and the SSCLI-Rotor

Darío Álvarez Gutiérrez
Dept. of Informatics,
University of Oviedo
c/ Calvo Sotelo s/n
33007, Oviedo, Spain
darioa@uniovi.es

María Ángeles Díaz Fondón
Dept. of Informatics,
University of Oviedo
c/ Calvo Sotelo s/n
33007, Oviedo, Spain
fondon@uniovi.es

Iván Suárez Rodríguez
Dept. of Informatics,
University of Oviedo
c/ Calvo Sotelo s/n
33007, Oviedo, Spain
banisr@telecable.es

ABSTRACT

Protection (access control) is a crucial issue in modern software systems. There are many different protection mechanisms, including Access Control Lists and the Code Access Security included in .NET. Capabilities are other well-known protection mechanism that has many merits. This paper describes a form of capability-based protection specially suited for Object-Oriented environments based on OO Virtual Machines that compares favorably with the .NET CAS mechanism in many contexts. The implementation of this protection model into the Microsoft SSCLI-Rotor implementation of the .NET platform is shown (RotorCapa), involving modification of core VM structures and behaviour of instructions. Besides other benefits, the early performance results of the RotorCapa system compared with .NET CAS protection are very encouraging, as it does not suffer from the exponential degradation of performance imposed by the security stack walking mechanism of .NET.

Keywords

Security, protection, access control, capabilities, performance, virtual machine, code access security, SSCLI, Rotor, .NET security.

1. INTRODUCTION

A protection mechanism (access control mechanism) is a security measure in computer systems that restricts access from a piece of code (subject) to a resource (object). An example is the well-known Access Control List protection mechanism: its variants are used in operating systems such as Unix.

Object-Oriented environments based on OO Virtual Machines (Java and .NET being prominent examples) need a protection mechanism, too. Subjects are here objects (instances of a class), and the resource to protect is also an object (calls to a method of an instance of other class). .NET is shipped with a protection mechanism called Code Access Security [Wat02], part of a more comprehensive security system. The mechanism is based on a form of stack introspection [Wal97] (stack walking of the internal VM stack holding security information).

But there are other protection mechanism, such as capabilities. Our research focus on the application of capability-based protection to object-oriented environments. We have implanted capability-based protection into the SSCLI-Rotor (the RotorCapa system[1]). SSCLI-Rotor [Stu03] is Microsoft's Shared Source implementation of the Common Language Infrastructure (.NET).

This paper describes briefly our model of capability-based protection and its advantages (in general and compared to the .NET CAS mechanism). Then, the implementation of this model into the SSCLI-Rotor is presented with more detail, involving modifications to the core of the Rotor VM (Just in Time Compiler, object layout, addition and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2004 *workshop proceedings*,
ISBN 80-903100-4-4
Copyright UNION Agency – Science Press, Plzen, Czech Republic

[1] RotorCapa development was supported by Microsoft Research through the RFP Rotor Awards. Home page is at http://www.di.uniovi.es/~darioa/rotorcapa/ and at the SSCLI Community Site http://rotorcapa.sscli.net/

modification of instructions, etc.). Some early and encouraging performance results are presented in the next section. Another section draws some conclusions about the SSCLI-Rotor as a research platform gained while developing the implementation. The paper ends with a comparison with related work, and the conclusions and future work section.

2. CAPABILITY-BASED PROTECTION

Pure Capabilities [Den66] are a well-known protection mechanism that can be used to implement a comprehensive set of flexible security policies. A capability is basically a ticket which names an object (resource) and a set of permitted operations on that object (permissions) (Figure 1). The only requirement for an object (subject or client) in order to use another object (object or server) is to hold a capability pointing to the server object with adequate permission to use the intended operation. Consequently, an object will hold just the minimum protection information relevant to it: the rights to just the objects it will use.

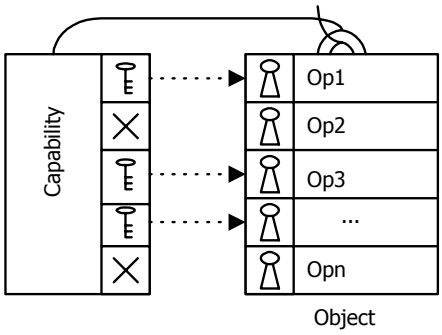


Figure 1. A capability.

Capabilities for OO Environments based on OO Virtual Machines

A big advantage of capabilities over other protection mechanisms such as the before mentioned access control lists, stack introspection, etc. [Wal97] is that they can be smoothly and easily integrated with the object model.

In our version of capabilities [Dia99], the protection information (permissions) can be integrated with object references in the machine, and the mechanism

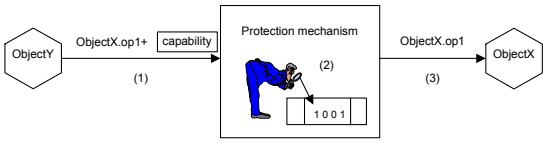


Figure 2. Checking permissions in a capability.

for testing the permissions can be integrated with the method call process (Figure 2. If the reference does not hold a permission for the method called in the destination object, the call fails, and an exception is raised).

Modifications to instructions (and structures) dealing with references must be also done accordingly. Just a new instruction to restrict the permissions a given capability is holding (to follow the principle of least privilege) is needed.

In fact, there are no conceptual changes to the Object Model, and the protection can be (and should be) seen as another property of the Object Model (encapsulation, inheritance, ...and protection).

We have previously worked with this model with our own OO environment with OO VM [Alv98] and have found advantages [Dia99] such as:

- Flexibility and adaptability
- High performance
- Integration with the object model
- Fine granularity of protection
- Reduced Trusted Computing Base, as a simple mechanism is implemented with a small code.
- More Hardened Systems, as the principle of least privilege can be followed with no restrictions.
- Compatibility with existing applications, as capabilities are used as normal references in applications.
- Scalability. Managing capabilities for thousands of objects is not a problem, as they are managed and stored as normal references in the objects themselves.

Capabilities have some drawbacks, most notably revocation problems, although there are solutions such as facades and reference monitors in case they were needed.

3. WHY CAPABILITIES FOR .NET?

The .NET security model is very complete and has many advantages. It includes a Code Access Security mechanism. So, why using other protection mechanism?

Capabilities in general have its own merits. They are clearly superior to Access Control Lists in terms of confinement [Har02]. Besides, there are some points in .NET security for some applications where our model of capabilities is a best fit:

- **Complexity**. The .NET security system is comprehensive and thus complex: evidence, policies, permission sets, stack walking mechanisms... For many applications that just need the base form of protection (such as the one provided by capabilities) this is overkill:
  - **Footprint and overhead**. The code, data, and runtime overhead needed by the .NET security system is present, although just a fraction of its power is used.
  - **Big Trusted Computing Base**. For the same reason, the trusted computing base of the system is big, and the probability of security bugs increases.
- **Access to source code needed**. To add protection to a given class, access to the source code of the class (to demand permissions) is needed, and the code to represent the permissions has to be created, too. With capabilities, any binary object can be protected anytime without effort (it just requires setting permission bits in the references).
- **Protection at the level of the class, not at the level of individual instances**. Since permissions are assigned based (roughly speaking) on the class of a client instance, not on an instance-by-instance basis. With capabilities, permissions are assigned on a reference-by-reference basis. Two objects of the same class can hold different permissions when calling a third object.

And finally, another reason is that .NET can be used just as a platform to research on other protection mechanism.

4. SSCLI-ROTOR IMPLEMENTATION OF CAPABILITIES: STRUCTURAL CHANGES

Since we had previous experience implementing capabilities in a VM, we expected to follow a similar path to implement capabilities into the SSCLI-Rotor 1.0. However, due to the constraints and the architecture of SSCLI-Rotor, we had to resort to a different approach, which is described in this and the following section.

Representation of capabilities in objects

Each (reference) attribute in an object can have a set of access permissions attached. A capabilities table holding these permissions is attached to every object (in the OBJECT structure), with an entry for each reference held in the object (Figure 3). A lazy-creation strategy is used so that objects that do not use protection (i.e. do not apply the operation to restrict permissions to a reference) do not have this necessary protection overhead.

The same is done for array references.

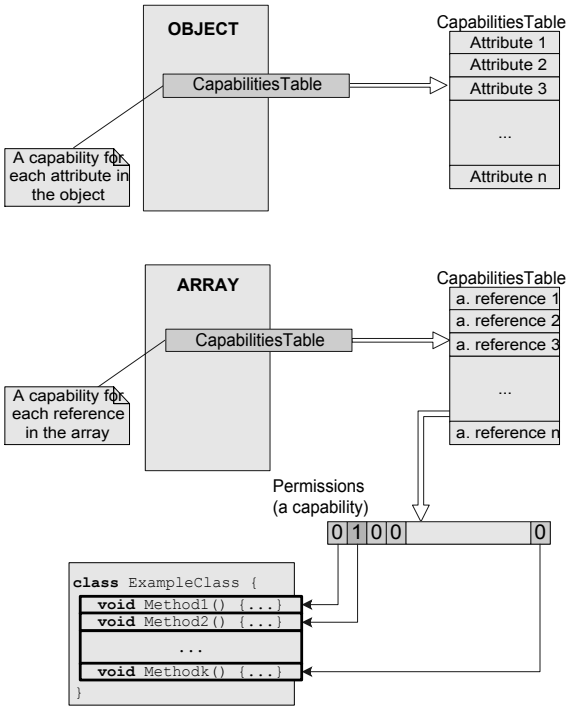


Figure 3. Representating capabilities in objects and arrays.

Support in behavioural structures

Implementing capabilities also needs support in structures used for behavioural purposes (execution): the execution stack (activation records) and the operation (evaluation) stack.

An activation record (stack frame) for a method can have references to other objects in local variables and method attributes (parameters). These references can have an associated set of permissions. A scheme using capabilities tables similar to the one used with objects is applied.

The permissions for these references are stored using one capabilities table for variables and one for attributes. These tables are organized into stacks that grow in parallel with the activation records (main stack).

The operation stack also holds references that can have permissions (for example, a reference to an object and references to object parameters are stacked prior to a method call to the first object. These references have associated permissions. A capabilities stack that mimics this operation stack holds the permissions for the references.

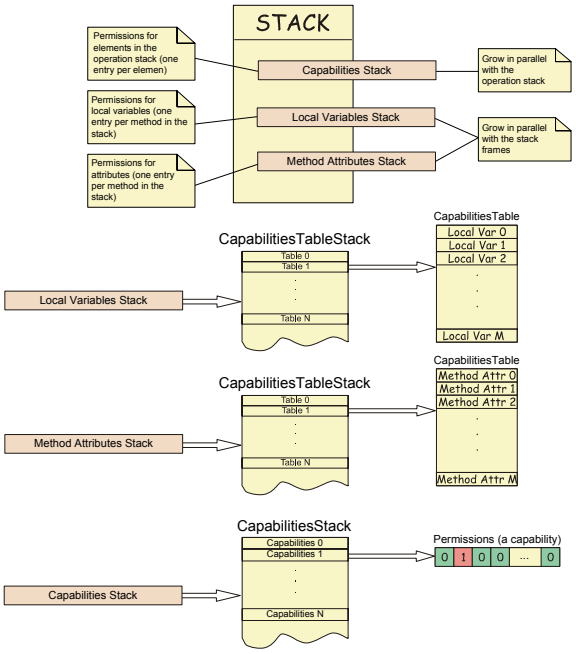


Figure 4. Representing capabilities in behavioural structures.

5. SSCLI-ROTOR IMPLEMENTATION OF CAPABILITIES: BEHAVIOURAL CHANGES

New instruction: Restrict <method>

This new instruction acts upon a reference (top of the stack), and denies access to the method specified, restricting the set of methods that can be called using the reference.

This is the primitive operation for security. Initially, the creator of an object holds a reference with all the permissions. The creator object can duplicate this reference, restrict some methods, and then pass the reference to others for secure computation (the set of available operations for these objects is restricted).

Call and callvirt now check permissions

The other pillar of capability-based protection is that method calls to an object should only be allowed if the reference (capability) used for making the call has the permission (bit) for the method set active.

Thus, **call** and **callvirt** instructions are modified accordingly. The instruction check that the reference to the object called (top of stack) has an asserted permission for the method being called (the bit for the method is "1" in the implementation). If the reference does not hold a permission (bit "0") a protection exception is raised.

Modifications to many other instructions

Although capabilities only affect the semantics of the "call" instruction (now a security exception might be thrown), MANY other instructions are indirectly affected. With capability-based protection, ALL references, including local variables, references in the stack, etc., have an attached set of permissions (conceptually, that is the philosophy of capability-based protection). The behaviour of the instructions that deal with references must take this into account, "manually" copying, deleting, etc. the set of permissions when dealing with references, as represented in structural and behavioural structures as shown before.

Some of the instructions that had to be modified are:

- Creation of new objects: **newobj** (when a new object is created a reference is returned. This reference has an associated set of permissions, initially set to "1" for the creator).
- Storing from the stack: **starg**, **stlocs**, **stfld**, **stsfld** (when storing a reference from the stack, the permissions associated to the reference must also be copied to the destination reference)
- Loading: **ldarg**, **ldloc**, **ldnull**, **ldelem.x**, **ldelem.ref**, **ldfld**, **ldsfld**, **ldsflda**, **ldstr** (symmetrically, permissions associated to a reference must be copied when the reference is pushed in the operation stack)
- Various: **dup**, **isinst**, **box**, **unbox**

Example of CLI code with capabilities

The following is a small example of the use of capabilities in the SSCLI (the **restrict** instruction):

```
...
// An object is created and a reference
// (capability) is left on the stack
newobj instance
       void Test::.ctor()
// A method is restricted in the
// capability in the top of the stack
restrict instance
       void Test::Message()
// Now the method is invoked using the
// reference in the top of the stack
// The reference can be stored, cloned,
// passed as an argument to other
// objects, etc.
callvirt instance
       void Test::Message()
// The call will not succeed and an
// exception is raised at this point,
// as the reference used has not the
// permission to call "Message" set
...
```

6. PERFORMANCE

Preliminary tests were made, comparing the SSCLI-Rotor capabilities system (RotorCapa) with a "normal" SSCLI-Rotor with the security system active. Access control to a method was checked by a very simple test program.

The test program involved a class with a given method. An instance of the class was created and then the method was repeatedly called using the reference to the object created. To protect the call in SSCLI-Rotor, a .NET permission protecting the method was created and granted to the original class. In RotorCapa, the permission for the method in the reference remained set to achieve the same effect.

The same test program was run with different stack depths before calling the method.

As the .NET security mechanism relies on stack walking, it was expected that the time taken by SSCLI-Rotor to execute the same program would increase with the stack size, as and domain intersections and stack walks are getting longer. The exponential degradation of performance shown in figure 5 confirms this.

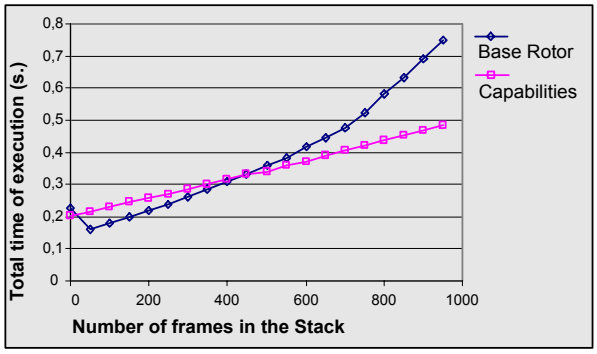


Figure 5. Performance of RotorCapa and SSCLI-Rotor (simple test).

RotorCapa, on the other hand, does not rely on stack walks. The mechanism always checks that a given permission is set on a reference or not, and this, ideally, is independent of stack size, and the number of permissions needed in the system. Thus, ideally, the time taken by RotorCapa should be constant. Actually, the figures show a linear increase of the time (much better than exponential). This can be related to the way method calls are handled and a non-optimal implementation of capabilities in this first version.

With modest stack sizes, RotorCapa performance is very similar to SSCLI-Rotor performance.

These performance results are very encouraging. On the one hand performance is equal or better than SSCLI-Rotor, and our implementation is barely optimized in this version. The test program was very simple. In fact, there is only one domain and one custom permission used. With more domains, and more custom permissions (as would be the case with real applications), the burden of stack walks, domain intersections and searches of permissions should be more apparent, and the performance of the capabilities system would become even more obvious.

7. PROS AND CONS OF THE SSCLI-ROTOR AS A RESEARCH PLATFORM

A very important bonus of working with SSCLI-Rotor is that we can now make further research on capabilities without having to resort build a complete VM to test the mechanism.

.NET is a system that has many "real" applications already done, and these applications can be directly used in SSCLI-Rotor (and therefore in RotorCapa). Thus, we can use real applications to research the cost of capability-based protection. To build similar applications in number and complexity for a custom system is something that is out of the picture. Obviously, the code to use capabilities must be added to the applications, but this is an incremental and relatively small process.

However, the implementation of capabilities in the SSCLI-Rotor source code was not as easy as we would have liked. Base Rotor security structures and code are deeply intermingled with the core of the Rotor VM. We did not try (yet) to delete them and avoid its overload. The nature of our project and the architecture of Rotor obliged us to "touch" almost every part of the Rotor VM: JIT, dozens of helpers for the JIT (assembly generation), memory layout, stack, threads, metadata, etc.

8. RELATED WORK

Capabilities as a protection mechanism are almost as old as computers, and many projects have used this protection mechanism, especially in the OS area. With the recent spread of security breaches in commercial Access Control List-based Operating Systems, there is a renewed interest in them, as shown in the EROS operating system [Har02], or the JX [Gol02] Operating System.

Capabilities were also used in object-oriented systems, for example in the Hidden Capabilities model [Hag96]. They were also used in OO systems based in Virtual Machines, such as the J-Kernel [Haw98] project for the Java platform.

All these projects use the basic philosophy of capabilities for protection. However, the specific variants differ in many aspects with our approach, mainly in how protection is smoothly integrated with the object model and the virtual machine structures and mechanisms in our system.

With respect to .NET and the SSCLI-Rotor, there are some projects that use the SSCLI to test or implement different protection mechanisms, such as the implementation of the Delegent authorisation system for SSCLI-Rotor [Ris03], but none (as far as we now) related to capabilities.

9. CONCLUSIONS AND FUTURE WORK

Protection (access control) is a crucial issue in modern software systems. There are many different protection mechanisms, including Access Control Lists and the Code Access Security included in .NET.

Capabilities are other well-known protection mechanism that has many merits. We have developed a form of capability-based protection specially suited for Object-Oriented environments based on OO Virtual Machines. Our system compares favorably with the .NET CAS mechanism in many contexts, as it is much simpler, with a smaller overhead, footprint, and trusted computing base, does not require access to the source code of a class (or additional coding) in order to protect it, and grants protection at the level of individual instances instead of at the level of classes.

We have successfully implanted this capability-based protection mechanism into the Microsoft SSCLI-Rotor implementation of the CLI (.NET) standard (the RotorCapa system). This involved modifications to structural and behavioural structures of the VM to represent the permissions associated to the capabilities, as well as modifications to the implementation of instructions that deal with references.

As expected, one advantage of capability-based protection is visible in the early performance results of the RotorCapa system. The performance is at least as good or much better than the .NET CAS figures, that show an exponential degradation of performance with stack size. Since the test program was very simple and the RotorCapa version in not much optimized, it is expected that the results would be better with test conditions similar to the ones had with real applications.

SSCLI-Rotor has proved to be a good platform for research, as we did not have to build a complete commercial-like VM to test our protection mechanism. Besides, we had a direct access to the vast array of existing applications created for the .NET platform. However, the nature of our work, and because of the Rotor architecture, involved modifying the source code of the many of the parts of the core Rotor system (and that was not as easy as expected).

Future work will be precisely in the area of performance testing. In a first phase, we will develop a more comprehensive benchmark of test programs, to exercise different elements of the system, and to represent conditions more similar to actual applications. In a second phase, we will instrument real .NET applications that are readily ported to Rotor, to measure the performance of the capability-based protection mechanism in real production conditions.

10. REFERENCES

[Alv98] Álvarez Gutiérrez, D., Tajes Martínez, L., Álvarez García, F., Díaz Fondón, M.A., Izquierdo Castanedo, R., and Cueva Lovelle, J.M. An Object-Oriented Abstract Machine as the Substrate for an Object-Oriented Operating System. Lecture Notes in Computer Science, 1357, pp. 537-544, 1998.

[Den66] Dennis, J., and van Horn, E. Programming Semantics for Multiprogrammed Computations, Comm. of ACM 9, No 3, 1966.

[Dia99] Díaz Fondón, M.A., Álvarez Gutiérrez, D., García-Mendoza Sánchez, A., Álvarez García, F., Tajes Martínez, L., and Cueva Lovelle, J.M. Integrating Capabilities into the Object Model to Protect Distributed Object Systems. Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99), IEEE Computer Society Press, pp. 374-383, 1999.

[Gol02] Golm, M., Felser, M., Wawersich, C., Kleinöder, J. A Java Operating System as the Foundation of a Secure Network Operating System. Technical Report TR-I4-02-05, University of Erlangen-Nuremberg, 2002.

[Hag96] Hagimont, D., Mosière, J., Rousset de Pina, X., and Saunier, F. Hidden Capabilities. 16th International Conference on Distributed Computing Systems, May 1996.

[Har02] Hardy, N., and Shapiro, J. EROS: A Principle-Driven Operating System from the Ground Up. IEEE Software, pp 26-33, January 2002.

[Haw98] Hawblitzel, C., Chang, C., Czajkowski, G., Hu, D., and von Eicken, T. Implementing Multiple Protection Domains in Java. 1998 Usenix Ann. Tech. Conf., Louisiana, June 1998.

[Ris03] Rissanen, E. Server based application level authorisation for Rotor. IEE Proceedings Software, 150(5), pp 291-295, October 2003.

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

[Wal97] Wallach, D.S., D. Balfanz, D. Dean, and Felten, E.W. Extensible Security Architecures for Java. 16th Symp. on Operating Systems Principles, Saint-Malo, France, 1997.

[Wat02] Watkins, D. An Overview of Security in the .NET Framework. January 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/netframesecover.asp

The .NET CF Implementation of GecGo
- A middleware for multihop ad-hoc networks -

Peter Sturm, Daniel Fischer, Volker Fusenig, Thomas Scherer
University of Trier
Department of Computer Science
D-54286 Trier, Germany
{sturm,fischer,fusenig,scherer}@syssoft.uni-trier.de

ABSTRACT

The goal of the GecGo middleware is to provide all the services required by self-organizing distributed applications running on multihop ad-hoc networks. Because of the frequent as well as unreliable and anonymous communication between accidental neighbors observed in these networks, applications have to adapt continuously to changes in the mobile environment and the GecGo middleware offers the required tight coupling. Additionally, GecGo addresses specifically the issue of "en passant" communication, where moving neighbor devices may interact only for short periods of time. In this paper, the architecture and basic concepts of the GecGo middleware are discussed and a prototype implementation of GecGo using the Microsoft Windows CE 4.2 .NET operating system for mobile devices and the .NET Compact Framework is presented.

Keywords
Middleware, Mobile Networks, Multihop Ad-Hoc Networks (MANET), Mobile Applications, Self-Organization

1. INTRODUCTION

The emerging capabilities of modern mobile devices with respect to CPU power, wireless communication facilities, and battery capacity are the foundation of future multihop ad-hoc networks. The frequent as well as unreliable and anonymous communication between accidental neighbors observed in these mobile networks makes their successful deployment a challenging task. With the absence of any reliable backbone network, all mobile devices have to participate altruistically in a distributed execution environment with some kind of epidemic message delivery. Self-organization is the most promising design principle in order to manage these networks successfully and efficiently. As a consequence, any decision of a mobile device must be based on local as well as on current neighborhood knowledge and common goals must be achieved by means of synergy.

Any fundamental communication pattern in such a network exhibits an *en passant* characteristic. Two devices are within communication range for a short period of time and while they pass each other, they might cooperate and exchange certain data. In most cases, being within communication range with a given device is purely accidental and the probability to meet this device again in the near future is fairly low. During this en passant communication, applications and middleware must agree fast on which entities should change the hosting device in order to get closer to their final destination. The required decisions depend on a number of factors, among others the importance of the moving entity, the size of the entity compared to an estimation of the remaining interaction period, and the future direction of the neighbor with respect to the final destination.

These stringent conditions for distributed applications in multihop ad-hoc networks aggravate the need for the continuous adaption to a dynamically changing environment. As a consequence, this requires a very tight coupling between the mobile applications and the middleware. Many high-level mechanisms that are common in traditional system software and middleware that trade transparency vs. performance are therefore inadequate. The goal of the GecGo middleware (Geographic Gizmos) is to offer this tight interaction with application components and to provide all the necessary services required by self-organizing systems running on multihop ad-hoc networks. The first prototype of this middleware has been implemented on Microsoft Windows CE 4.2 .NET using the .NET Compact Framework.

In the next section, the fundamental concepts and the basic functionality of the GecGo middleware are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2004 workshop proceedings,
ISBN 80-903100-4-4
Copyright UNION Agency – Science Press, Plzen, Czech Republic

introduced. The functionality of the GecGo middleware is discussed in section 3. The architecture and implementation issues of the .NET prototype are discussed in section 4. The paper ends with an overview on related work and a conclusion.

2. GECGO CONCEPTS

The conceptional structure of the middleware and its four basic abstractions are depicted in figure 1. Any mobile or stationary device participating in the GecGo runtime environment is represented by a `DeviceGizmo` and the code of GecGo applications is derived from the base class `CodeGizmo`. Every code has its residence in form of a device. Depending on the distributed execution model, this residence remains fixed or it might change over time (mobile agents). For application code with a fixed residence, GecGo provides the abstraction of mobile state (`StateGizmo`) that might change the hosting device instead of the code. Since end-to-end messages between devices may remain on a device for a longer period of time in case no suitable neighbor is found, they also exhibit a more state-like nature. As a consequence, messages are represented in GecGo as special cases of a `StateGizmo`.

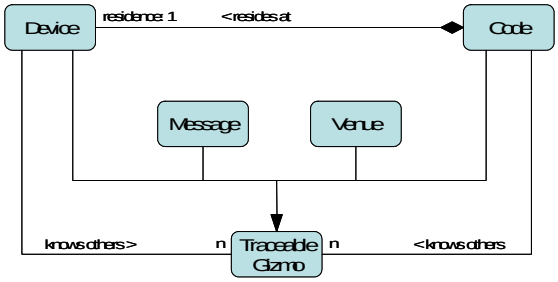


Figure 1: Main GecGo Abstractions

The fourth abstraction is defined by the `VenueGizmo` which ties a logical place or an event to a well-defined set of geographic coordinates and time slots, e.g. a several week long lecture on distributed systems with changing rooms and time slots. `VenueGizmos` are virtual in the sense, that they bear no computational resources per se. Instead they rely on the devices that are within a given distance from the venue center. Entities with a venue as their destination will first try to reach a device at the venue. As long as they have no other destination, they will try to remain at the venue possibly by changing the hosting device.

GeoTraces

All major abstractions in GecGo are derived from a fundamental data type `TraceableGizmo` (see figure 2). Any subtype of this class is traceable in time and space by means of a `GeoTrace`. These traces keep accounts on events in the past, they reflect the present situation, and they store estimates about future events. The actual information stored in the trace of a gizmo is defined by its type and consists of a set of so-called `Gepots` (pieces of time and geographic data). Also the depth and the level of detail of the `GeoTrace` depends on resource considerations and the actual type of gizmo. For example, `DeviceGizmos` keep track about where they have been in the past, at what time as well as why and they may also store information about previous neighbor devices. The present informs about the current position of the device and the actual neighborhood. The future trace might contain estimates where the device will be in the future, e.g. students will be in certain future lectures with a high probability.

Traces of `StateGizmos` will be more resource-limited. They will store at least the final destination as part of the future trace. `VenueGizmos` are even more restricted, since they represent only virtual entities within the GecGo environment. As such, the trace of the venue is identical to the time schedule of the event associated with this venue. Additional data that might be important to run the venue must be stored by the hosting devices that are currently within the vicinity of the venue center.

All devices are required to update their traces continuously over time. With the goal to keep the number of Gepots in the past to a reasonable minimum, the information stored in the present of the trace will be shifted into the past, e.g. when a mobile device starts moving again. The traces of devices are also the primary source for changes in the traces of other currently hosted state and code gizmos.

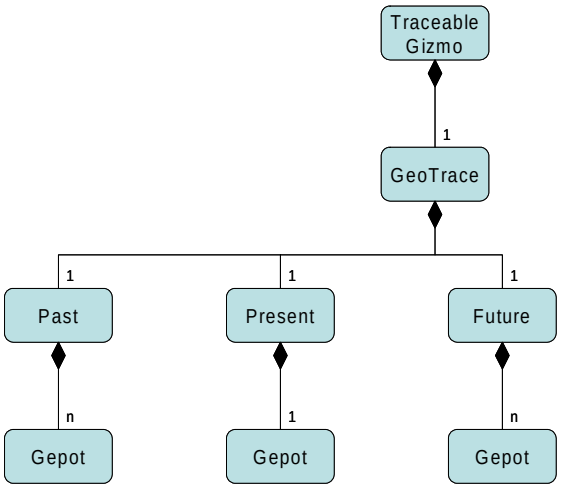


Figure 2: Basic Type "Traceable Gizmo"

From a conceptual point of view, the main function of the GecGo middleware enables traceable gizmos to move towards new destinations. The most common type of movement allows for mobile state to reach a given mobile device or to get into the vicinity of a certain venue, e.g. to implement the marketplace communication pattern for multihop ad-hoc networks as presented in [2]. In this context, a marketplace is a application-defined geographic area with an expected

high density of mobile devices (e.g. a lecture hall). Entities of a mobile application move between mobile devices of the users and the marketplace using a geographic routing protocol. By concentrating applications on specific geographic areas, marketplaces increase the probability that corresponding entities of the application get in contact with each other.

Covered by the concepts of GecGo are also the movement of mobile devices to reach a given venue (the special case of a navigation system, of course with the physical help of the human device owner) and movement of mobile code between devices as a means to implement mobile agent systems.

3. GecGo Architecture

The basic architecture of the GecGo middleware consists of two gizmo management domains (see also figure 3):

- the `Lobby` for all the gizmos that are in transit and haven‘t reached their final destination yet
- the `Residence`, with gizmos that are intended to stay at this device for a longer period of time

Movement of gizmos from the lobby to the residence and vice versa will be performed with the aid of the porter service. Primarily, the porter is responsible for securing the identity of incoming gizmos and for providing the resources requested.

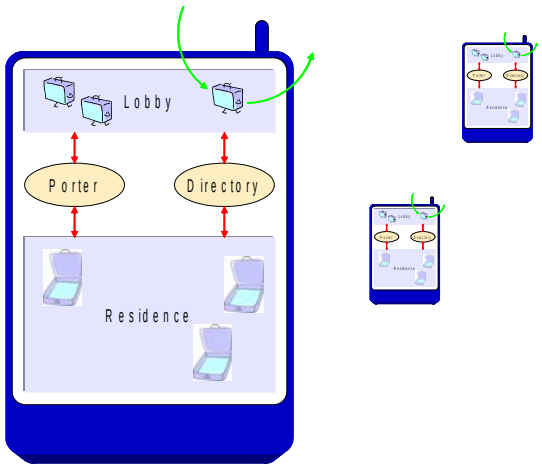


Figure 3: GecGo Device Architecture

The Directory Service

A central directory service keeps track on any changes in both management domains. The directory itself has a hierarchical structure with leaves at the gizmo level (see figure 4). Applications may query the directory with wildcards to locate the required information. Most of the attributes of a gizmo entry are application-specific. For this purpose, the directory allows the definition of arbitrary key/value pairs as part of a gizmo leave. Additional attributes inside the directory tree are defined by the GecGo middleware and primarily serve infrastructure purposes such as the number of gizmos actually stored in the lobby or the list of neighbor devices moving in a southern direction.

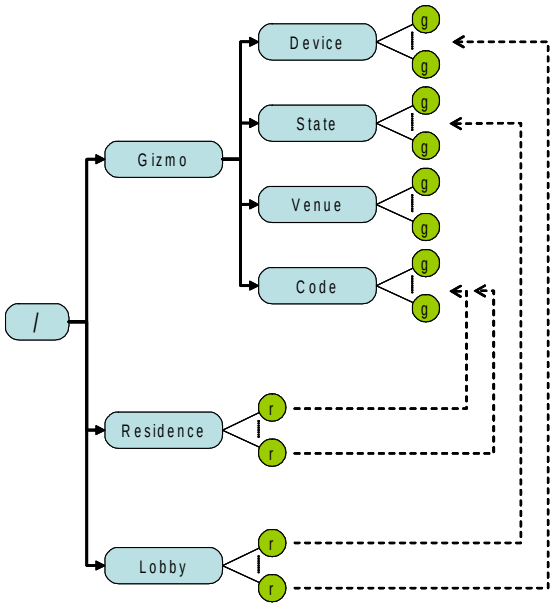


Figure 4: The Directory Tree

As depicted in figure 4, the directory tree consists of three major branches. The branch named Gizmo contains all the information and attributes about any gizmo located on the given mobile device. Additionally, entries about specific gizmo types that are known by a mobile application but where no instance is available on the device yet can be inserted by application components. In conjunction with the asynchronous notification mechanism described below, applications can react appropriately on future events.

The remaining two branches below the root of the tree keep track on the gizmos located in the residence or the lobby. Primarily, these branches contain references to specific gizmos that are part of these domains. The residence branch holds references to code gizmos that are installed on the mobile device or that entered the device via the porter in case of a mobile agent environment. The lobby has references to the gizmos awaiting a device change in order to reach their final destination. Most of these gizmos are of type `StateGizmo`, although again in a mobile agent environment, the lobby could also be occupied by code gizmos that might change the host or enter the residence through the porter. Also part of the lobby branch are references to device gizmos that represent the current neighborhood of the mobile device. This information about the current neighborhood is also available as part of the device’ present `GeoTrace`.

Gizmos in the lobby and the residence may query for the existence of certain gizmos and they may register a callback to be informed about specific events. For this purpose, every gizmo has an application defined unique name that serves as the key for the directory service. Possible events supported by the middleware are:

- arrival of a new gizmo with a specified type in the lobby of a device
- departure of a gizmo from the lobby
- movement of gizmos between the lobby and the residence
- instantiation and deletion of new gizmos by means of code gizmos in the residence

Communication with direct Neighbors

Singlehop communication services for gizmos with devices in the immediate neighborhood are provided by the middleware kernel. Therefore the middleware has to know all available devices within singlehop communication range. Information about these devices are periodically updated in the lobby branch of the directory tree as references to device gizmos. These entries store the following attributes:

- an unique device id, which is used to resolve the current ip address of the device
- the port which is used for gizmo transmissions
- a flag indicating if the device currently accepts incoming gizmos
- the future GeoTrace of the mobile device in order to ease the decision which gizmos to move during the en passant communication

The required information is exchanged among mobile devices by broadcasting a beacon in regular time intervals using an UDP based communication protocol. These beacons contain all the aforementioned information to ease gizmo exchange. The protocol is also being used to transmit other kinds of messages such as termination signals or requesting a certain gizmo. With the device information—containing ip address and port of a potential communication partner—the middleware is able to establish a TCP connection to transfer a gizmo.

Mobile State vs. Mobile Code

A central decision in mobile ad-hoc networks addresses the issues on mobile code vs. mobile state. Mobile agents are an interesting technology for wireless and mobile networks with far reaching implications on system security and code integrity. The GecGo middleware covers mobile agents in its model by accepting a changing residence for mobile code. This functionality is currently not part of the .NET implementation of the GecGo middleware platform. Besides technical reasons, this decision is primarily driven by a number of unsolved problems with respect to the limited resources on a mobile device, the larger amount of data required to move mobile code including its execution state transparently from one device to another, and the need to authenticate and secure code execution.

Instead of mobile code, the GecGo middleware actually offers so-called *mobile state*, which requires the application components to cooperate non-transparently in packaging and unpacking execution state into and from mobile state gizmos. The middleware offers several functions and services to ease this task for the application code. In contrast to mobile code, applications must be installed explicitly by the user of a device, before state gizmos for a given application can be received and processed on their final destination. Of course no application code must be installed on devices that are only intermediate hosts for state gizmos.

An Routing Gizmo Example

To illustrate the dynamics inside the GecGo middleware, an example for a multihop routing service is elaborated in more detail. In this scenario, a routing gizmo—a subclass of a code gizmo—implementing a version of a geographic routing protocol is available as a pre-installed GecGo application. In this constellation, the routing gizmo is responsible for the end-to-end communication in the ad-hoc network on the basis of exchanging state gizmos between neighboring devices as provided my the middleware.

The routing gizmo registers a callback with the event that will be invoked in case a new neighbor device is within communication range. The event causes the routing gizmo to examine the beacon information of this neighbor in order to decide which state gizmo to transfer. As mentioned above, part of the beacon information are geographical coordinates about future positions of the neighbor device. This position information is the primary source for the routing decision, e.g. if a venue in the future trace of the neighbor device is identical to the future venue of a candidate state gizmo or if the trajectory of the neighbor device is targeting towards the destination of the state gizmo.

In a first scenario we assume that a state gizmo *g*, which is currently part of the lobby on a device *d1*, wants to reach a venue gizmo *v*. Each time such a new state gizmo with different destination enters the lobby of a device, the enter-event triggers the execution of specific callback in the routing gizmo in order to store its destination venue. Suppose now that another device *d2* is passing by *d1*. In that case the routing gizmo decides with respect to the used routing protocol (e.g. greedy routing) whether *g* moves from *d1* to *d2* or not. This procedure is repeated on different devices until *g* arrives on a device close to its destination venue.

In another routing scenario, gizmos residing on two different mobile devices want to exchange information directly, for example a mobile application on device *d1* wants to send a state gizmo *g* to another application on a specific device *d2*. Since a routing path from *d1* to *d2* among multiple mobile devices

cannot be maintained because of the dynamics in multihop ad-hoc networks, *d1* must determine the position of *d2* in some other way. One solution to this problem is, to hash the identification of *d2* to a specific geographic position within a given area (e.g. the university campus). This hash function is identical on every participating device and consists of a classical cryptographic hash to achieve a statistically unified distribution and a subsequent mapping of this randomized identifier to a geographic position. The calculated geographic position will be the target for the state gizmo in the same manner as in the example above. In return the device *d2* itself is able to compute the same geographic position and it might issue additional requests to collect the information or to periodically send updates about its position to this venue.

4. .NET IMPLEMENTATION

A first prototype version of GecGo has been implemented using the Microsoft Windows CE 4.2 .NET operation system and the .NET Compact Framework. The middleware has been written in C#. At the time of writing, the middleware has a size of approximately 3000 lines of C# source code. The GecGo executable itself has a size of 84 KBytes at runtime without any mobile applications installed. The target mobile devices are Compaq IPAQs H5550 with 128 MByte main memory and integrated WLAN communication facilities. The middleware can also be executed on ordinary notebooks supporting the full version of the .NET framework. The porter service as described in section 3 is not part of the current implementation. The main reason for this is the concentration on mobile state instead of mobile code. As a consequence, all mobile applications are currently installed explicitly on any participating mobile device with no need for the porter services.

The GecGo middleware architecture consists of five major classes:

- `Middleware`: This class serves as the main access point for mobile applications. The class also handles incoming state gizmos and enables the execution of new applications.
- `Beacon`: The UDP beacon required for the discovery of devices within communication range is sent periodically by an instance of this class.
- `UDPListener`: Primarily, this class handles incoming beacons of other devices. It also acts on the receipt of UDP-based requests by neighbor devices to prepare for the transmission of another gizmo.
- `TCPServer`: This class is used on the receiving side to transmit complete gizmos from one device to another. The device willing to send a gizmo takes up the client role.
- `GecGoDirectory`: This class defines the interface to all directory-related functions of GecGo.

The single instance of class `Middleware` implements the graphical user interface of GecGo. The user can browse the mobile applications installed on the device and select individual applications to be executed. For this purpose, the Middleware maintains a hash table to derive the reference to the corresponding assembly via its name. Each time an application is started, the entry point of the assembly—a public static method called `run(IMiddleware mid)`—is executed. In this call, the argument `mid` of type `IMiddleware` defines all the functions that are available to mobile applications. In the prototype version of GecGo, the following methods and properties are provided:

- `SendToAllNeighbors(Gizmo g)`
- `SendGizmo(Gizmo g, Device r)`
- `RegisterApplication(Assembly a)`
- `IGecGoDirectory gd`

The property `gd` returns a reference to the directory service of the middleware through the interface `IGecGoDirectory`. This interface encapsulates all the directory functions available to the mobile application. Among others, the directory service currently defines the following methods:

- `InsertGizmo(Gizmo g)`
- `DeleteGizmo(Gizmo g)`
- `RegisterEvent (Delegate f, string regExp)`
- `UnregisterEvent(Delegate f)`
- `GetGizmos(string regExp)`

The delegate mechanism of C# is used to implement the asynchronous callback mechanism of the GecGo directory service. For example, if a gizmo inside the residence wants to be notified upon the arrival of gizmos of a given type T in the lobby, it registers a delegate with the event `/Lobby/T/<Enter>` and the middleware will call back each time such a gizmo enters the device. The .NET events may also be used to implement asynchronous notifications between different gizmos.

The different communication tasks of the middleware are handled by 3 dedicated threads, which are created through instances of the classes `Beacon`, `TCPServer`, and `UDPListener`. Any device maintains information about its current neighborhood through the listening UDP thread which receives any beacon messages issued by the Beacon thread of nearby devices. The detailed information about the neighbor will be continuously reflected by the structure of the directory service and may—for example—trigger the transmission of state gizmos in the lobby of a device. In this case, an UDP request is sent to the potential next host. If this device is willing to act as a host, a TCP connection will be established and the selected gizmo will be transmitted. The decision to rely on TCP connections for the transmission of gizmos even if the size of the gizmo would fit into a single WLAN frame only holds for the prototype version of GecGo.

While implementing the thread-based communication part of the middleware, several limitations of the .NET compact framework became obvious. Several management functions of the full .NET framework such as asynchronous thread termination are missed in the compact version of .NET. As a consequence, more complex synchronization techniques must be implemented to manage the active number of threads. Also the interaction between beaconing and dealing with the receipt of UDP messages would benefit from additional functionality around UDP sockets, e.g. to add a time-out value to a blocking receive on an UDP socket.

Before sending, any gizmos are serialized. The binary format of a serialized gizmo follows the TLV principle (Type, Length, Value). In the current version of GecGo it is assumed that a gizmo is defined by the values of the non-static class members, so that there are no interface implementations needed to marshall objects. Thus, the serialized form is a sequence of these values. Each member is described by a type, a name and a value triple. Non-primitive values are marshalled recursively in the same way and arrays are characterized by the enumeration of their values. By using the reflection mechanisms of the .NET Compact Framework the member informations and values are queried independently from their visibility. The reason for the explicit implementation of a gizmo serialization is the missing support of the `BinaryFormatter` and the `SoapFormatter` classes of the .NET framework in the compact version. The main reason for this were size and performance considerations which might hold for today's mobile devices. But in anticipation of future heterogeneous and mobile computing environments, there will be a increasing need for a standardized serialization mechanism besides accessing web services.

Threads in GecGo

A first set of experiments was targeted towards the Microsoft system software in order to determine the costs induced in a multi-threaded implementation of the middleware running on top of Windows. Observations at the beginning of the implementation phase lead to the initial decision, to implement a first middleware version with a limited number of threads only, because thread instantiations, deletions, and context switches appeared to be to costly on the IPAQ devices. In order to verify this observation, a simple C# test program has been implemented which instantiates a variable number of threads. Each of these threads yields the CPU in a while loop for a given period of time (2 minutes). The calculated time for a single context switch is depicted in figure 5 for 2 to 256 threads in a single address space (application domain). The same source code has been compiled for 2 different release platforms: (1) a Windows console application with the full support of the .NET foundation classes (Version 1.1) and (2) a smart device application using the .NET compact framework only. Both executables have been executed on a 1900 MHz mobile Intel Pentium 4. As the graphs in figure 5 indicate, the time for a single context switch increases slowly with the number of threads, starting with 937 ns (.NET CF executable) and 915 ns (full .NET executable) for two threads. The absolute number for 256 threads are 3361 ns (.NET CF executable) and 3438 ns (full .NET executable).

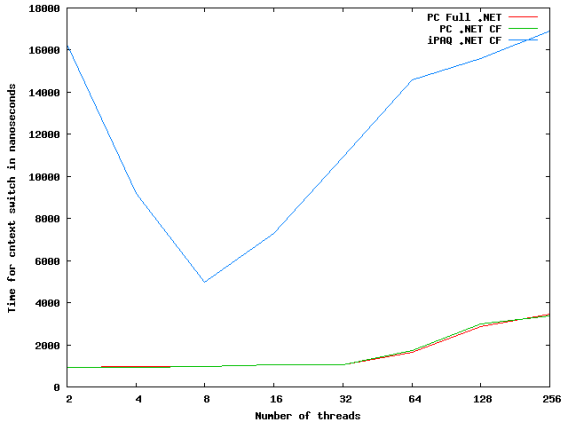


Figure 5: Duration of context switches

The absolute times for the iPAQ .NET CF executable running on a 400 MHz Intel Xscale processor are 16.28 microseconds for 2 threads and again 16.889 microseconds for 256 threads. A little surprising was a significant minimum of 4.99 microseconds for 8 threads. Provided that the time of a single context switch with no required change in address space is determined primarily by the CPU speed, there is a factor of 4.75 in CPU speed (1900 MHz to 400 MHz), compared to a factor of 17.38 (worst case) and 5.17 (best case) in context switch time.

Measuring Gizmo Exchange

It is crucial for the successful execution of distributed applications in multihop ad-hoc networks, that a sufficiently large number of gizmos can be exchanged during en passant communication. In another set of experiments, this number of transferred gizmos between mobile devices within communication range is determined. The test programs for these experiments are written in C# as mobile applications using the functionality of the GecGo prototype. Varying parameters are the size of state gizmos, the number of participating devices, and the average mobility of specific devices.

The number of gizmos with increasing size transferred successfully between two iPAQs using the first prototype version of GecGo are depicted in table 1. As expected, the number of gizmos transmitted decreases with increasing gizmo size. Obviously, these numbers are still fairly low compared to the available throughput of nominal 11 Mbps offered by the WLAN adapters of the mobile devices. But since

the prototype version of GecGo has not yet been optimized and performance tuned, hopefully there will be space left for improvements. Especially, the frequency of beacon messages, the impact on power consumption, as well as the fine tuning of execution paths during the exchange of gizmos has not been investigated in more detail yet. In order to estimate the possible increase in throughput, it is also intended to measure the achievable number of bytes transmitted over a plain TCP connection in an identical scenario between several iPAQs in an accompanying experiment.

Size of state gizmo (Bytes)	Transmitted gizmos per second	Throughput (Bytes/s)
0	11.3	-
512	2.12	1083
1024	1.6	1638
2048	0.87	1774
4096	0.45	1843
8192	0.22	1775
16384	0.11	1757

Table 1: Transmitted gizmos between 2 devices

Nevertheless, this version of the GecGo prototype can already be used for first experiments with mobile applications: assuming that two devices are within communication distance for at least 50 seconds—provided that two mobile devices with a communication range of 50 meters travel with a speed of about 1 meter per second—they can exchange e.g. about 100 gizmos of size 512 bytes.

The same experimental program can also be used for more than two devices in order to determine the influence of neighboring devices on gizmo exchange. Due to the spread spectrum modulation of wireless communication, the interference between pairs of communicating devices appears to be fairly low. With 4 iPAQs, where gizmos are exchanged among pairs of mobile devices, the number of successfully transmitted gizmos is identical to the 2 iPAQ scenario (2.12 and 2.13 gizmos per second in case of 512 byte size).

First GecGo Applications

The GecGo middleware platform is currently used to implement several example applications, to gain experience with the abstractions provided by the middleware and to improve the platform architecture and functionality. We started with the development of a simple e-learning application: a peer-to-peer quiz for students to assist in the preparation of examinations. The basic idea is to enable participating students to issue interesting examination questions. These questions are propagated by the GecGo middleware to the corresponding venue that has been assigned to the specific course. Participants interested in examination questions for a given course will issue a request that too will be propagated to the corresponding venue where it remains for some period of time to collect new items. This collection of new questions will be realized by means of additions to the initial mobile state gizmo. Eventually, the request will move back to the sending owner and any results will be presented to the user. Additionally, the application enables students to rate and to order a set of questions from a didactical point of view. Rates and orders are again sent to the venue to be accessible to other participants.

The implementation of additional mobile applications for ad-hoc networks using GecGo is planned for the next 9 months: a mobile auction system and a self-organizing electronic rideboard in an university environment [1,5]. These applications have been investigated already on a simulated basis [3,6] and as prototypes running on a java-based middleware called SELMA [4], the predecessor of GecGo.

5. Related Work

Traditional middleware systems such as CORBA, Microsoft DCOM or Java RMI are not suitable for mobile ad-hoc networks because they rely on central infrastructure like naming services and assume the reachability of all network nodes. These assumptions cannot be matched by mobile multihop ad-hoc networks. Additionally, traditional middleware approaches are too heavyweight for mobile devices. Many adaptions have been made to apply them in mobile settings such as OpenCORBA [7] or Next-GenerationMiddleware [8]. These extensions provide mechanisms for context awareness, but cover mainly infrastructure networks and one-hop mobile communications.

An increasing number of middleware systems is developed specifically for mobile ad-hoc networks. XMIDDLE [9] allows the sharing of XML documents between mobile nodes. Lime [10] and L2imbo [11] are based on the idea of tuple-spaces [12], which they share between neighbored nodes. But due to the coupling of nodes, these approaches are not well-suited for highly mobile multihop ad-hoc networks. MESH-Mdl [13] employs the idea of tuple-spaces as well, but avoids coupling of nodes by using mobile agents, which communicate with each other using the local tuple-space of the agent platform. Proem [14] provides a peer-to-peer computing platform for mobile ad-hoc networks. STEAM [15] limits the delivery of events to geographic regions around the sender which is similar to the geographically bound communication at marketplaces. STEAM provides no long distance

communication, it is only possible to receive events over a distance of a few hops.

Mobile agent frameworks exist in numerous variations, Aglets [16] or MARS [17] may serve as examples. These frameworks were designed for fixed networks and thus the above mentioned problems of traditional middleware approaches apply to them as well. The SWAT infrastructure [18] provides a secure platform for mobile agents in mobile ad-hoc networks. This infrastructure requires a permanent link-based routing connection between all hosts and thus limits the ad-hoc network to a few hops and it is therefore not applicable to en passant communication pattern.

6. CONCLUSIONS

The specific nature of multihop ad-hoc networks enforces a tight coupling between the middleware and any mobile application. The sole dependence on information local to the mobile device leads to new programming and execution models, that favor self-organization and adaption to a continuously changing environment. The specific architecture of the GecGo middleware as presented in this paper is trying to address these issues by supporting mobile application components and by providing flexible interaction mechanisms between entities on a single device as well as entities on mobile devices that are within communication range for short period of times.

One of the major goals of this project is to verify that the system software offered by Microsoft, which addresses wireless infrastructures and mobile applications, also suits application needs in multihop ad-hoc networks. At the time of writing, only preliminary results are available. The successful implementation of the GecGo middleware indicates, that in principle no arguments prohibit the usage of the .NET Compact Framework in such an environment. But in many situations, it was obvious that the current version of the Compact Framework addresses issues in singlehop networks only. In such an environment, the wireless communication facilities are primarily substitutes for a physical wire with the traditional protocol stack on top of it. This is no disadvantage, since it is meant to support exactly this environment. But questions to be answered in the future of this project will address issues on additional support for multihop ad-hoc networks and how to integrate these required functions into existing system software such as the .NET Compact Framework.

7. FUNDING

This work is funded in parts by the german science foundation DFG as part of the Schwerpunktprogramm SPP1140 „Basissoftware für selbstorganisierende Infrastrukturen für vernetzte mobile Systeme“. The Microsoft Windows CE and Microsoft .NET Compact Framework implementation of GecGo is funded by an Microsoft Research Embedded Systems IFP Grant (Contract 2003-210).

8. REFERENCES

[1] H. Frey, J.K. Lehnert, and P. Sturm. “*Ubibay: An auction system for mobile multihop ad-hoc networks*,” Workshop on Ad hoc Communications and Collaboration in Ubiquitous Computing Environments (AdHocCCUCE'02), New Orleans, Louisiana, USA, 2002

[2] D. Görgen, H. Frey, J. Lehnert, and P. Sturm, “*Marketplaces as communication patterns in mobile ad-hoc networks*," in Kommunikation in Verteilten Systemen (KiVS), Leipzig, Germany, 2003

[3] J. K. Lehnert, D. Görgen, H. Frey, and P. Sturm. “*A Scalable Workbench for Implementing and Evaluating Distributed Applications in Mobile Ad Hoc Networks*,” Western Simulation MultiConference WMC'04, San Diego, California, USA , 2004

[4] D. Görgen, J. K. Lehnert, H. Frey, and P. Sturm. “*SELMA: A Middleware Platform for Self-Organzing Distributed Applications in Mobile Multihop Ad-hoc Networks*;” Western Simulation MultiConference WMC'04, San Diego, California, USA, 2004

[5] H. Frey, D. Görgen, J. K. Lehnert, and P. Sturm. “*Auctions in mobile multihop ad-hoc networks following the marketplace communication pattern*,” submitted to 6th International Conference on Enterprise Information Systems ICEIS'04, Porto, Portugal, 2004

[6] H. Frey, D. Görgen, J. K. Lehnert, and P. Sturm. “*A Java-based uniform workbench for simulating and executing distributed mobile applications*,” FIDJI 2003 International Workshop on Scientific Engineering of Distributed Java Applications, Luxembourg, Luxembourg, 2003 (to appear in Springer LNCS)

[7] T. Ledoux. “*OpenCorba: A reactive open broker*," Springer LNCS, Volume 1616, pp. 197ff, 1999

[8] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. “*An architecture for next generation middleware*," in Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer-Verlag, London, UK, 1998

[9] S. Zachariadis, L. Capra, C. Mascolo, and W. Emmerich. “*XMIDDLE: Information sharing middleware for a mobile environment*," in ACM Proc. Int. Conf. Software Engineering (ICSE02). Demo Presentation, Orlando, Forida, USA, 2002

[10] G. P. Picco, A. L. Murphy, and G.-C. Roman. “*LIME: Linda meets mobility*," in International Conference on Software Engineering, pp. 368-377, 1999

[11] N. Davies, A. Friday, S. P. Wade, and G. S. Blair. “*L2imbo: A distributed systems platform for mobile computing*," ACM Mobile Networks and Applications (MONET) - Special Issue on Protocols and Software Paradigms of Mobile Networks, Volume 3, pp. 143-156, Aug. 1998

[12] S. Ahuja, N. Carriero, and D. Gelernter. “*Linda and friends*," IEEE Computer, Volume 19, pp. 26-34, Aug. 1986.

[13] K. Herrmann,. “*MESHMdl - A Middleware for Self-Organization in Ad hoc Networks*," in Proceedings of the 1st International Workshop on Mobile Distributed Computing (MDC'03), 2003

[14] G. Kortuem. “*Proem: a middleware platform for mobile peer-to-peer computing*," ACM SIGMOBILE Mobile Computing and Communications Review, Volume 6, Number 4, pp. 62-64, 2002

[15] R. Meier and V. Cahill. “*STEAM: Event-based middleware for wireless ad hoc networks*," in 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02), Vienna, Austria, 2002

[16] D. Lange and M. Oshima. “*Programming and Deploying Java Mobile Agents with Aglets*,” Addison-Wesley, 1998

[17] G. Cabri, L. Leonardi, and F. Zambonelli. “*MARS: A programmable coordination architecture for mobile agents*," IEEE Internet Computing, Volume 4, Numer 4, pp. 26-35, 2000

[18] E. Sultanik, D. Artz, G. Anderson, M. Kam, W. Regli, M. Peysakhov, J. Sevy, N. Belov, N. Morizio, and A. Mroczkowski. “*Secure mobile agents on ad hoc wireless networks*," in The 15th Innovative Applications of Articial Intelligence Conference, American Association for Articial Intelligence, 2003

Peer-to-Peer Applications on Mobile Devices: A Case Study with Compact .NET on Smartphone 2003

Fabio De Rosa
Università di Roma "La Sapienza"
Dipartimento di Informatica e Sistemistica
Via Salaria 113 (2nd floor, lab C4)
00198 Roma, Italy
derosa@dis.uniroma1.it

Massimo Mecella
Università di Roma "La Sapienza"
Dipartimento di Informatica e Sistemistica
Via Salaria 113 (2nd floor, room 231)
00198 Roma, Italy
mecella@dis.uniroma1.it

ABSTRACT

The traditional approach to information systems, accessed by users by means of powerful devices (such as desktops and laptops) with known features, will not be anymore significant in the future years. Indeed, the current trend suggests that it will be possible to offer continuous access to all information sources, from all locations and through various kinds of devices, mainly small and mobile (e.g., palmtops and PDAs, cellular phones). Therefore, the need emerges for the design of applications for smart devices, which are highly flexible, capable of exploiting in an optimal way the resources. This experience paper analyzes the opportunity to design, develop and deploy interactive applications running on smart cellular phones (commonly referred to as smartphones), based on a peer-to-peer communication model and GPRS technology. A case study is presented to verify whether current development tools and technologies for small devices require a radical different approach with respect to more traditional application development. As a development platform, Window Mobile for Smartphone 2003 with Compact .NET has been used, which is currently available, at least in Europe, only on prototype devices.

Keywords

Peer-to-Peer – Mobile Device – Compact Framework .NET – Smartphone 2003.

1. INTRODUCTION

The traditional approach to information systems, accessed by users by means of powerful devices (such as desktops and laptops) with known features, will not be anymore significant in the next years. The current trend suggests that it will be possible to offer continuous access to all information sources, from all locations and through various kinds of devices, either powerful but mainly static (e.g., PCs, laptops) or small but mobile (e.g., palmtops and PDAs, cellular phones). Moreover, users are interested in a wider and wider variety of applications, beyond traditional vocal interaction access to data of any kind and complex interactive applications, also with transactional properties.

Telecommunication networks, indeed, are continuously evolving and diversifying; each kind of network has its own features, in terms of capacity, reliability, quality of service security, availability, cost. Such features change significantly with the various applications that make use of the network services. However, the user is not interested in technical details: he/she wants to access the end services from the current location and with the best possible performances. Therefore, the need emerges for the design of applications which are highly flexible, capable of exploiting the resources in an optimal way. Finally, traditional client/server computing, based on the availability of centralized or clustered servers offering services and applications to clients, is leaving the place to more decentralized paradigms, such as peer-to-peer computing, in which loosely coupled devices (i.e., the peers) interact with each other without previously established mutual agreements and knowledge. The goal of the "MAIS"[1]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies'2004 *workshop proceedings*,
ISBN 80-903100-4-4
Copyright UNION Agency – Science Press, Plzen, Czech Republic

[1] Multi-channel Adaptive Information Systems – MAIS – is an Italian research projects, jointly carried

project is the development of models, methods and tools that allow the implementation of adaptive information systems able to provide services with respect to different types of networks and access devices.

The work presented in this experience paper, with respect to the MAIS project sphere, is centered on mobility and small device application development, specifically on design and implementation issues related to the development of distributed applications running on cellular phones.

This work aims at experimenting the opportunity to design, develop and deploy interactive applications running on smart cellular phones (commonly referred to as smartphones), based on a peer-to-peer communication model. Current services offered by telecommunication operators are mainly based on a centralized paradigm, in which cellular phones download simple applications from services available on the Web, and, if communication with other users/devices is needed, it is obtained through centralized services. Apart from vocal interaction and exchange of SMS/MMS[2], no other end-user application is currently designed and deployed assuming a direct and peer-to-peer interaction between users/devices. Even if this can be due to commercial and exploitation considerations, we argue that peer-to-peer interactions between devices should be considered, as a possible alternative for the future, on which to base future commercial and exploitation strategies[3]. In depth, the work presents the development of a peer-to-peer application running on smartphones, in which all communication is based on GPRS technology: as development platform, Window Mobile for Smartphone 2003 with .NET Compact Framework has been used, which is currently available, at least in Europe, only on prototype devices. From a practical software engineering point of view, the aim was also to verify whether current development tools for small devices require a radically different approach with respect to more traditional application development.

The paper is organized as follows. In Section 2 relevant background is presented, focusing on the peer-to-peer computing model and on the technologies for application development on smartphones. In Sections 3 and 4, the application used as case study is presented, whereas in Section 5 a discussion on the gained experience and some insights are presented.

2. BACKGROUND

In this section we give a brief overview and state of art on peer-to-peer (P2P) systems and architectures, as well as on technologies and tools commonly used to realize smart device applications.

Peer-to-Peer Systems and Protocols

The interest for peer-to-peer (P2P) systems has been considerably growing during the last years. Although it is considered a revolution in network based environments, it is actually only an evolution of the original Internet model, that enables packet exchanges among nodes with interchangeable roles. The P2P acronym refers to each distributed system in which nodes can be both clients and servers. In other words, all nodes provide access to some of the resources they own; in the context of this paper, the resources are services provided/accessed from the peers (i.e., mobile devices), enabling a basic form of cooperation among them. An interesting classification of P2P systems can be found in [1], in which the following three models are introduced: **Decentralized Model, Centralized Model,** and **Hierarchical Model**. With respect to such a classification, the application presented in Section 3 has been developed according to the decentralized model. Example of P2P software architectures and systems are [2, 3, 4] and Gnutella [5], the first system implementing a fully distributed file search. All such systems and protocols have been thought for wired networks, that is, networks in which the connection between two peers remains established as long as peers dwell in the system (static connections). Works that take into account mobility scenarios (i.e., dynamic connections) can be found in [6] and [7], respectively. In the former, a mobile P2P architecture and platform is proposed; in the latter, instead, a special-purpose P2P file sharing tailored to Mobile Networks, denoted Optimized Routing Independent Overlay Network (ORION), is presented.

---

out by about 10 subjects, including Universities and enterprises. The interested reader can refer to http://black.elet.polimi.it/mais/index.php.

[2] In this work, when we refer to interaction, we consider it at the application level, not at the network one. Of course also vocal interaction and SMS/MMS exchange run through centralized servers (e.g., the SMS dispatch center), but users perceive such communication as direct with the others. Conversely, current applications, such as the recently appeared distributed games, require that all application-level communication is collected through a centralized service, and users/devices do not communicate directly.

[3] New computing paradigms for cellular phones could foster GPRS and UMTS technologies in a similar way that Napster/Gnutella-based systems made the Internet popular among teenagers.

Table 1. Differences among .NET CF and J2ME (CLDC/CDC) platforms

	.NET CF	J2ME
Supported Devices	Devices with Windows Mobile (in Europe cellular phones not yet available)	Java-enabled devices (for MIDP 2.0 only high-end phones)
Language Support	C#, Visual Basic .net, C++	Only Java
Virtual Machine	Unique CLR Virtual Machine	Different versions: CDC and CLDC Virtual Machines
Byte Code Compatibility	Standard .net CLR	No compatibility with J2SE, and between CDC and CLDC
API Compatibility	Between all platforms supporting .net CF (currently only Windows Mobile)	Partial compatibility between CDC, CLCD, and J2SE
Development Tools	Visual Studio .net 2003	Several tools (by SUN and different vendors, not completely integrated)
Testing	Emulators in Visual Studio .net 2003	Various emulators (provided by SUN and by device vendors)
Client Installation	ActiveSync or through Internet Explorer download	Device synchronization mechanisms and OTA (Over The Air) download

The convergence of Instant Messaging [8] with the *conference* peer-to-peer protocols (commonly referred to as call protocols between two or more peers) considers issues similar to the ones addressed by our simple application. In particular, both the ITU (International Telecommunications Union) standard H.323 [9,10] and SIP (Session Initiation Protocol, [11]) are protocols for multimedia conference calls over IP, that can be used to establish, maintain and terminate calls between two or more peers. A current effort to combine Instant Messaging and SIP is made by the SIMPLE (SIP for Instant Messaging and Presence Leveraging Extensions) working group [12]: its goals are of applying SIP to the instant messaging and presence (IMP) suite of services, thus enabling the development of distributed multimedia applications between different peers communicating through messages (and not requiring continuous connections as in SIP). To the best of our knowledge, currently there is not yet a complete specification and no implementation is available. The purpose of our application is very similar to the one of SIMPLE (even if much narrower in generality), and SIMPLE availability would have considerably reduced the development issues.

Smartphone Application Development

Nowadays, two of the most promising technologies for developing and running applications on smart cellular phones are Java 2 Micro Edition (J2ME, [14]) and the Compact .net Framework (.net CF) [13]. Both J2ME and .net CF are platforms that, differently from micro-browser technologies such as WAP (Wireless Application Protocol), support rich user interfaces, leverage device extensions (e.g., GPS – Global Positioning System - and barcode scanners), and security protocols. Furthermore, compared with smartphone native platforms (e.g., eMbedded Visual C++, C++ SDKs for the Symbian OS), both those technologies have managed environments enabling component-based application development, thus improving developer productivity and application reliability. Table 1 summarizes the differences among the .NET CF and J2ME (CLDC/CDC) platforms, with respect to the designer's/developer's point of view.

.NET Compact Framework

.NET Compact Framework (.net CF) [13] is a subset of the desktop .NET Framework. It has two main components, namely the Common Language Runtime (CLR) and the Base Class Library. The Common Language Runtime is responsible for managing code during execution: it provides core services such as memory and thread management, designed to enhance performance. Just-In-Time (JIT) compilers enable the generated code to run in the native machine language of the target platform.

The Base Class Library is a collection of reusable classes that are used to develop applications; they provide common and reusable programming tasks such as string management, data collection, database connectivity, user interface, etc. The classes included in the .NET CF provide an identical interface to their counterparts in the workstation/server .NET Framework; some functionalities are not supported due to size constraints, performance issues, or limitations in the target operating system (e.g., printing, Multiple Document Interface forms, Drag-and-Drop functionalities, etc). Class behaviors, properties, methods, and enumeration values are the same under both versions of the .net Framework.

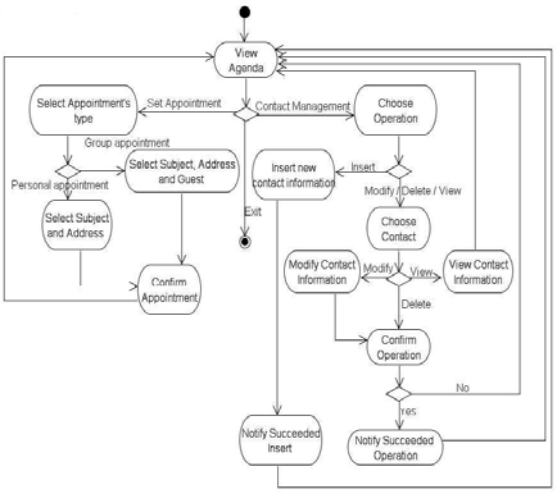


Figure 1. Activity Diagram for the Appointment and Contact Management.

Such libraries include classes for connecting to a remote data source, submitting queries, and processing results; frequently, they are used as a robust, hierarchical, disconnected data cache to off-line (i.e., during disconnections) work with remote data. XML APIs, instead, are the same classes provided by the .NET Framework, and used to develop applications manipulating XML structures.

3. THE INTERACT-AGENDA APPLICATION

In this section we describe the design of our SMS- and GPRS-based application (called Interact-Agenda), as well as the proposed peer-to-peer architecture and protocol on top of which the application is based. In next section implementation details are provided.

Application Requirements

The Interact-Agenda application is an interactive agenda for smartphone devices, which allows users to automatically organize appointments between several persons. The application offers to users the following functionalities:

- **visualization**, to view details of one or more appointments, also in the mode “all of the week";
- **appointment management**, for inserting, deleting and modifying both personal and group appointments;
- **contact management**, for inserting, deleting and modifying contacts in the personal book.

The novelty of the application with respect to already existing ones (e.g., Pocket Outlook) providing the same functionalities, is that the appointment management is carried out in a (semi-)automatic way, on the basis of the protocol described in Section 3.3. Currently, a user willing to organize an appointment with several persons: *(i)* decides a candidate time slot (on the basis of its agenda); *(ii)* manually contacts all involved persons (by calling them, by sending them an SMS, by writing an e-mail, etc.) and waits for their reply; *(iii)* if all invited persons agree upon the time slot, he sends them a confirmation, else *(iii ')* he/she chooses a new time slot and begins the process again. All such activities are carried out manually by the proposing user, and they are a serious burden for very busy persons.

The idea of our Interact-Agenda is to provide an application, running on user smartphones, that carries out all the negotiation automatically, and only at the end (i.e., after finding a suitable time slot) asks all involved users for confirmations[4].

When a user creates a new **personal** appointment in his/her agenda, all the details are stored (as it normally happens when the user takes an appointment in Outlook): conversely, when the user creates a new group appointment, a negotiation procedure with all involved persons is required.

In Figure 1 we report the complete Activity Diagram for the application. When a user selects the Interact-Agenda menu, he/she can choose between the following options: to enter in the appointment management section; to enter in the contact management section; or to exit from the application. On the basis of the choice, the user can do several tasks; for example, if he/she has chosen the appointment management section, the user can view, modify, delete or create an appointment.

In Figure 2 we report an example of the Sequence Diagram for the negotiation phase between three users (and their smartphone devices), carried out in order to establish a group appointment proposed by **user_1**. After the proposer (i.e., **user_1**) has entered all appointment information (through an appropriate sequence of windows), the peer instance of Interact-Agenda running on its smartphone communicates with the peer instances of Interact-Agenda running on the smartphones of **user_2** and **user_3**, in order to verify the availability near the proposed date, and if not possible, asking the user a new time slot and conducting a new negotiation round[5].

[4] As users, the authors would not like a smartphone taking appointment without at least letting them know !!

[5] The Interact-Agenda instance of the proposer asks the others for the availability of time slots near the initially

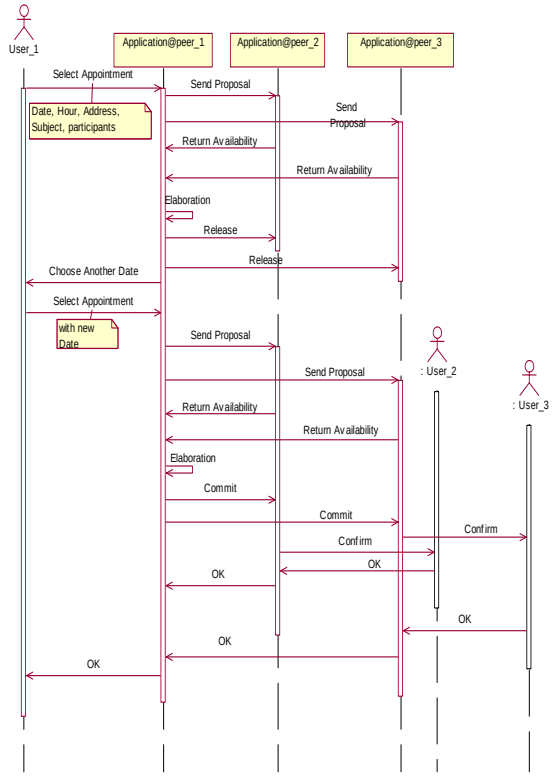


Figure 2. Sequence Diagram of the Appointment Negotiation.

All this process happens silently and without (or minimal) intervention by the users. At the end, when a time slot has been "agreed upon" by the peer instances of the Interact-Agenda, a message is displayed to the respective users in order to gain confirmation. That negotiation process runs on top of SMS messages and GPRS network connections, as detailed in Section 3.3.

Application Design

The Interact-Agenda is developed and deployed as a peer-to-peer smartphone application: each device hosts an instance of the application. In Figure 3, the structure of (an instance/copy /peer of) the Interact-Agenda application, is shown. The application consists of:

- the `User Interface` package, in which all the user-interface functionalities are managed;
- the `Application Logic` package, in which all the negotiation logic is contained;
- the `Logical Data` and `Database` are the packages managing the local database (storing appointment records and contact records). Logical Data is an abstraction of simple DBMS functionalities (table creation, tuple insertion, deletion, modification and selection, etc.), while Database is a concrete implementation of it (in our application it implements that operations on .txt files).
- the `Peer-to-Peer` package, which implements the peer-to-peer protocol (see Section 3.3), and the Networking package that manages SMS-based communications and GPRS connections.

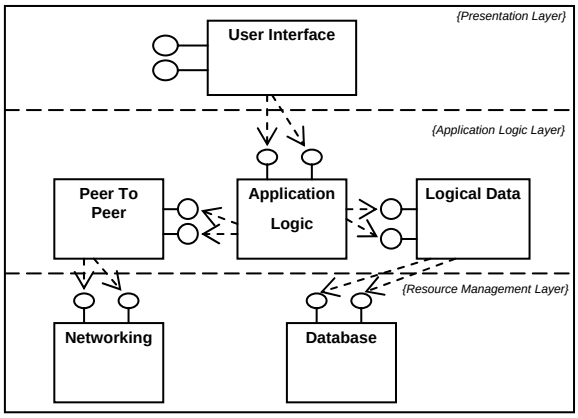


Figure 3. Structure of the Application.

Communication Protocol

The communications between the smartphone devices are conducted over GPRS and SMS. Indeed, each device (peer) may be both client and server at the same time, sending and receiving messages (by SMS sockets) to/from another peer and exchanging information by GPRS sockets. A device receives and replies to an appointment request incoming from another peer, which is the *negotiation initiator*; in turn, the device can initiate an appointment negotiation phase (its user wants to establish an appointment with other persons).

In Figure 4 we illustrate the whole protocol. It may be split into two phases: in the first phase the initiator communicates to all members its IP address by a SMS with a particular header; each active destination, after receiving the message, replies with an "I-am-alive" message; if the initiator receives all the responses from all the clients, then it can proceed with the second phase, that is, the exchange of appointment information (month, day, hour, etc.)

proposed one, in order to find a match; near means a given number N of time slot before or after the originally proposed one, and it can be configured. If a match cannot be found, the details of a new appointment are requested from the proposer; the number of repetition of the negotiation process is therefore directly driven by the proposing user.

with all participants, through sockets on GPRS. Otherwise, i.e. if at least one device has not replied to the initial message, the initiator terminates the negotiation phase and closes all opened connections with the others clients. Then the initiator (which acts as a server of the different sockets) exchanges information with the other devices (acting as clients) until the agreement has been reached or the proposing user decides to abort the appointment. Therefore the second phase of the protocol is straightforwardly derived from Figure 2. During all the protocol, the initiator sets timeouts: if not all replies are collected before their expiry, the initiator aborts everything. Different timeouts are set for the first and second phase (longer for SMS-based communications, lasting hours, and shorter for socket-based communications, as in usual network

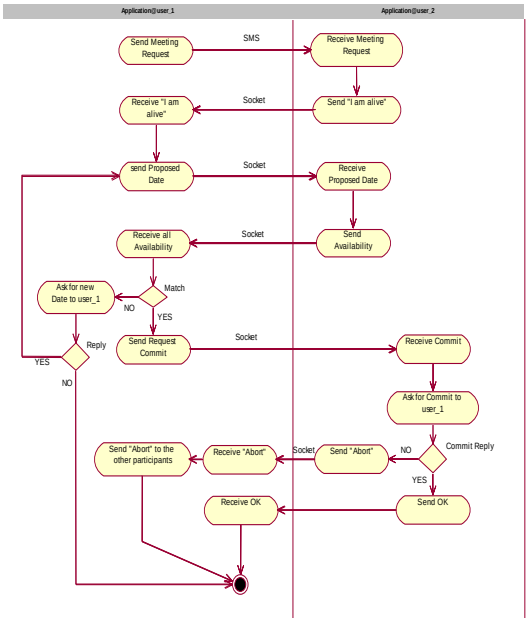


Figure 4. Communication Protocol.

- Currently (at least in Italy), an SMS is more expensive than the GPRS transmission cost. The former is about 20 cent of Euro, the latter is about 0.6 cent of Euro per transmitted kilobyte. As the dimension of data exchanged between peers is low (few bytes), it is more convenient to use a GPRS transmission rather than a SMS communication. Indeed, the proposed protocol has been thought to reduce the number of exchanged SMS messages; to establish an appointment among *N* peers, it needs no more than *N* SMS: the peer that initiates the negotiation sends its IP to all participants (*N* - 1).

4. IMPLEMENTATION FEATURES

In the current section we provide some implementation insights of our Interact-Agenda application. We will concentrate only on those packages in which .NET CF has been heavily used, providing specific functionalities to be considered during the development of smartphone application, i.e., networking, data storage and data access, user interface logic (windows, forms, listbox, etc.) and device interactions (keypad, joypad, home button, record button, soft keys, etc.). The development of the other packages, being pure business logic, does not present peculiarities due to smartphone device and .net CF (it is “normal” C# code). We will stress the possibility and the simplicity for an programmer to implement powerful and graceful application running on smartphone devices, both in terms of used libraries and generated source code lines.

programming practice). The choice of two different technologies (SMS during the first phase and GPRS connections in the following) is needed because of:

- GPRS connections assign an IP address dynamically (i.e., each time the device connects), therefore it is not possible to statically store information such as < user, telephone number, IP address of its device >. Conversely, the initiator can discover its IP address (as described in Section 4.2), then it can send such an IP address to all other devices (through SMS messages, as the couple < user, telephone number > is static), listening to a specific GPRS sockets, and finally all contacted devices can connect to such an initiator's socket through GPRS.

User Interface Development

The Base Class Library provides a sufficient subset of the components/widgets provided in the workstation/server .NET Framework; therefore no further training time is needed by a programmer in order to develop the user interface. In order to build graceful forms, we use the `System.Windows.Forms` and the `System.EventHandler` packages. The produced code is similar to that would have been produced for workstation applications.

Networking

The networking logic has been realized by combining the .NET CF library `System.Net` and the .NET CF mechanism for external procedure call, i.e., the method provided by the .NET CF technology (more in general by the .net Framework) to invoke procedures contained in external libraries (`.dll`). Indeed, in order to use both SMS and GPRS communication, we had to consider external native libraries, and embed calls to these `.dll` in the source code managing our peer-to-peer protocol. Then, in order to establish a connection between two peers

over the GPRS protocol, we used the system library `System.Net`, that provides all functions needed to manage socket-oriented connections over TCP/IP networks.

The code shown in the following is the one executed by the initiator for sending an SMS with its IP address and creating a specific socket; contacted device has to reply with an "I-am-alive" message on that socket (see class `Receiver`). Again, it is not very different from the one to be used in workstation/server scenarios for managing socket-oriented communications.

```
public class Initiator{
 ..............
 /* GPRS Connection Management */
   private System.Net.Sockets.Socket
      getBindSocket() {
      GPRS.DataCall();
      /* Getting local device IP */
      IPHostEntry ipHostInfo =
      Dns.Resolve(Dns.GetHostName());
      IPAddress ipAddress =
ipHostInfo.AddressList[0];
     ...........
      /* Opening TCP/IP socket */
      System.Net.Sockets.Socket sock = new
System.Net.Sockets.Socket(System.Net.Sockets.Addres
sFamily.InterNetwork,System.Net.Sockets.SocketType.
Stream,System.Net.Sockets.ProtocolType.Tcp);
      try {
         sock.Bind(localEndPoint);
         sock.Listen(10);
      }
      catch(Exception e) {
         MessageBox.Show(e.ToString());
         throw(e);
      }
      return sock;
   }
   /* Message sending */
   private void contact() {
      for(int i = 0; i < guest.Count; i++) {
      try {
SMS.SendMessage(guest[i].phoneNumber,this.myIP);
      }
      catch(Exception sendSms){}
      }
}
   ..........
}


public class Peer
{    ..............
   private void sendMessage(Message msg) {
      netStream = new
         System.Net.Sockets.NetworkStream(getConnec
         tSocket());
      writerClient = new StreamWriter(netStream);
      ...........
   }
}
```

Database

In the Smartphone 2003 SDK, up to now, there are no libraries and tools to manage local relational database. Therefore the Database package has been realized on the basis of the `FileSystem`. Specifically, all the μ-databases used in the Interact-Agenda consist of collections of text files stored on an external SD-CARD memory. Through the file management, the package provides a very simple relational-like interface, allowing upper layer to create tables, columns simple constraints and primary keys. Simple querying capabilities (specifically *select* but not *join*) have been provided. Specifically, the .NET CF libraries `System.IO` and `System.Data` have been used for writing and reading operations on files and for table and column manipulation, respectively. In the following code sample we report database management, in particular how tables are stored into text files.

```
public class DataBase{
   ...........
   /* Writing table to file.txt */
   public static bool WriteTable(DataTable tab) {
      System.Data.DataTable table = tab;
      System.String fileTable = table.TableName;
      ForeignKeyConstraint key = null;
      UniqueConstraint pKey = null;
      /* openig table file.txt */
      System.IO.StreamWriter fileWriter =
         System.IO.File.CreateText("\\"+fileTable+"
.txt");
/*Writing Table information: Name, column number */
      fileWriter.WriteLine(
      table.TableName.ToString());
      fileWriter.WriteLine(
      table.Columns.Count.ToString());

      /* Writing column's name and type */
      for(int i = 0; i < table.Columns.Count; i++)
         fileWriter.WriteLine(table.Columns[i].Colu
mnName);
         fileWriter.WriteLine(table.Columns[i].Data
Type.ToString());
      }
```

. . . .

}

Finally, Figure 5 shows the sequence of windows presented to the initiator user when establishing a group appointment.

Figure 5. Task to Establish a Group Appointment.

5. CONCLUSION AND DISCUSSION

The principal goal of the present work has been to investigate how to develop and design distributed applications for small devices (PDAs, smartphones, etc.), using current technologies such as .net CF platform. Our main conclusion, supported by the analysis of the produced code, is that concepts, patterns and development methods that are used in traditional software construction can be seamlessly applied to smart device application development. Clearly this is due to the availability of a framework that is similar to desktops/workstations/servers. In particular, the network programming interfaces are powerful enough for developing peer-to-peer applications on cellular phones, that was the second main objective of this work.

Some observations and recommendations can be made; specifically, when we design applications for smart devices, we must consider some factors such as transmission cost, security, privacy, performances and development platforms. With respect to them, smartphone applications must be able to use several transmission channels (our application uses two distinct channels: SMS and GPRS), and their protocols must be able to support different communications. In the future work we would like to apply the gained knowledge about design, development, and deployment of mobile applications in particular scenarios, such as the one of MANET (Mobile or Multi-hop ad hoc NETwork), in which mobile devices communicate on the basis of a non-fixed network [15].

6. ACKNOWLEDGMENTS

This work is supported by MIUR through the FIRB 2001 Project MAIS. Thanks to Microsoft Research - University Relationship Department and Marco Combetto for providing the prototype devices.

7. ADDITIONAL AUTHORS

Angelo Ritucci and Giuseppe Santoro, bachelor students of the Faculty of Computer Engineering, Università di Roma “La Sapienza”.

8. REFERENCES

[1] K. Aberer and Z. Despotovic. Managing Trust in a Peer-2-Peer Information System. In Proceedings of the 10th International Conference on Information and Knowledge Management, Atlanta, GA, USA, 2001.

[2] KaZaA. http://www.kazaa.com.

[3] Napster. http://www.napster.com.

[4] M.C. Fauvet, M. Dumas, B. Benatallah, and H.Y. Paik. Peer-to-Peer Traced Execution of Composite Services. In Proceedings of the 2nd VLDB International Workshop on Technologies for e-Services (VLDB-TES 2001), Rome, Italy, 2001.

[5] Gnutella. The Gnutella Protocol Specification (version 0.4). http://www9.limewire.com/developer/gnutella_protocol_l 0.4.pdf, June 2001.

[6] T. Kato, N. Ishikawa, H. Sumino, J. Hjelm, Y. Yu, and S. Murakami. A Platform and Applications for Mobile Peer-to-Peer Communications. NTT DoCoMo & Ericsson Research Document, 2003, http://www.research.att.com/~rjana/Takeshi Kato.pdf.

[7] A. Klemm, C. Lindemann, and O.P. Waldhorst. A Special-Purpose Peer-to-Peer File Sharing System for Mobile Ad Hoc Networks. In Proceedings IEEE Semiannual Vehicular Technology Conference (VTC2003-Fall), Orlando, FL, October 2003.

[8] RFC 2779 - Instant Messaging / Presence Protocol Requirements. http://www.faqs.org/rfcs/rfc2779.html.

[9] International Telecommunications Union Standard H.323. http://http://www.itu.int/.

[10] The OpenH323 Project. http://www.openh323.org/

[11] SIP - Session Initiation Protocol. http://rfc.sunsite.dk/rfc/rfc3261.html.

[12] SIMPLE - SIP for Instant Messaging and Presence Leveraging Extensions. http://www.ietf.org/html.charters/simple-charter.html.

[13] .NET Compact Framework. http://msdn.microsoft.com/mobility/prodtechinfo/devtools/netcf/.

[14] Java 2 Micro Edition. http://java.sun.com/j2me/.

[15] F. De Rosa, V. Di Martino, L. Paglione, and M. Mecella. Mobile Adaptive Information Systems on MANET: What We Need as Basic Layer? In Proceedings of the 1st Workshop on Multichannel and Mobile Information Systems (MMIS'03), Rome, Italy, December 2003, IEEE.

The correctness of the definite assignment analysis in C#

Nicu G. Fruja
Computer Science Department, ETH Zürich
8092 Zürich, Switzerland
fruja@inf.ethz.ch

ABSTRACT

In C# the compiler guarantees that each local variable is initialized before an access to its value occurs at runtime. This prevents access to uninitialized memory and is a crucial ingredient for the type safety of C#. We formalize the definite assignment analysis of the C# compiler with data flow equations and we prove the correctness of the analysis.

Keywords

definite assignment, C#, type safety, static analysis

1 INTRODUCTION

Let us suppose that an attacker wants to fool the C# type system. His idea is expressed by the next block:

```
{
  int[] a;
  try {a = (int[])(new object());}
  catch(InvalidCastException)
    {Console.WriteLine(a[7]);}
}
```

A pure `object` is type casted into an array of integers. The attacker thinks the following will work: after the `InvalidCastException` which is thrown at runtime is caught, the `object` can be used in the handler of the `catch` clause, as an array to generate unpredictable behavior. Some people might think that a `NullReferenceException` is thrown at runtime when `a[7]` is accessed and thus the attacker will not succeed. Actually, his idea does not work since the block is rejected already at compile time due to the definite assignment analysis. Through this analysis, the C# compiler infers that `a` might not be assigned in one execution path to the access of `a[7]`. The analysis states that `a` is not definitely assigned at the beginning of the `catch` block since it is not definitely assigned at the beginning of the `try` statement.

A necessary condition for C# to be a type safe language is the following: whenever an expression is evaluated, the resulting value is of the type of the expression. If we suppose that a local variable is uninitialized when its value is required, the execution proceeds with the arbitrary value which was at the memory position of the uninitialized local variable. Since this value could be of any type, we would obviously violate the type safety of C# and we could easily produce unpredictable behavior.

Since local variables are not initialized with default values like static variables or instance variables of class instances, a C# compiler must carry out a specific conservative flow analysis to ensure that every local variable is *definitely assigned* when any access to its value occurs. This definite assignment analysis which is a static analysis (see [Nie99, Gru00] for other static analyses) has to guarantee that there is an initialization to a local variable on every possible execution path before the variable is read. Since the problem is undecidable in general, the C# Language Specification [Wil03, §5.3] contains a definition of a decidable subclass. So far, the definite assignment analysis of the Java compiler has been formalized with data flow equations in the work of Stärk et al. [Sta01] and related to the problem of generating verifiable bytecode from legal Java source code programs. A formalization of the analysis for Java which uses type systems

Permission to make digital or hard copies of all part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or fee.
.NET Technologies'2004 *workshop proceedings,*
ISBN 80-903100-4-4
Copyright UNION Agency - Science Press, Plzen, Czech Republic

is presented in [Sch03]. Since in our case, the analysis involves a fixed point iteration, the presentation as type systems does not appear to be a feasible solution.

The formalization of the C# definite assignment analysis we provide, sheds some light in particular on the complications generated by the `goto` and `break` statements (incompletely specified in [Wil03]) and by the method calls with `ref`/`out` parameters - these are crucial differences with respect to Java. We also use the idea of data flow equations (see [Sta01]) but due to the `goto` statement, the formalization cannot be done like in Java. For a method body without `goto`, however, the equations that characterize the sets of definitely assigned variables can be solved in a single pass. If `goto` statements are present, then the equations defined in our formalization do not specify in a unique way the sets of variables that have to be considered definitely assigned. For this reason, a fixed point computation is performed and the greatest sets of variables that satisfy the equations of the formalization are computed. Another difference with respect to Java is the presence of structs. Regarding the correctness of the analysis, we prove that, these sets of variables represent exactly the sets of variables assigned on all possible execution paths and in particular they are a *safe approximation*.

A series of bugs in the Mono C# compiler were detected during the attempt to build the formalization of the definite assignment analysis (see [Fru03b] for details). This is the reason we refer here only to .NET and Rotor C# compilers. A bug in the assignment analysis of the Rotor C# compiler is mentioned also here.

The rest of the paper is organized as follows. Section 2 introduces the data flow equations which formalize the C# definite assignment analysis while Section 3 shows that there always exists a maximal fixed point solution for the equations. In order to define the execution paths in a method body, the control flow graph is introduced in Section 4. The paper concludes in Section 5 with the proof of the correctness of the analysis, Theorem 1. Due to space limitations we do not make here the proofs in detail. We focus on illustrating how we deal with the jump statements and their complications in the presence of `finally` blocks. The full details, as well as further examples can be found in the full technical report [Fru03b].

2 THE DATA FLOW EQUATIONS

In this section, we formalize the rules of definite assignment analysis from the C# Specification [Wil03, §5.3] by data flow equations. Since this analysis is an intraprocedural analysis, we restrict our formalization only to a given method *meth*. We use labels in order to identify the expressions and the statements. Labels are denoted by small Greek letters and are displayed as superscripts, for example, as in ${}^{\alpha}exp$ or in

${}^{\alpha}exp$	the data flow equations
`true`	$true(\alpha) = before(\alpha)$ $false(\alpha) = vars(\alpha)$
`false`	$false(\alpha) = before(\alpha)$ $true(\alpha) = vars(\alpha)$
$(\texttt{!}\ {}^{\beta}e)$	$before(\beta) = before(\alpha)$ $true(\alpha) = false(\beta)$ $false(\alpha) = true(\beta)$
$({}^{\beta}e_0\ \texttt{?}\ {}^{\gamma}e_1\ \texttt{:}\ {}^{\delta}e_2)$	$before(\beta) = before(\alpha)$ $before(\gamma) = true(\beta)$ $before(\delta) = false(\beta)$ $true(\alpha) = true(\gamma) \cap true(\delta)$ $false(\alpha) = false(\gamma) \cap false(\delta)$
$({}^{\beta}e_1\ \texttt{\&\&}\ {}^{\gamma}e_2)$	$before(\beta) = before(\alpha)$ $before(\gamma) = true(\beta)$ $true(\alpha) = true(\gamma)$ $false(\alpha) = false(\beta) \cap false(\gamma)$
$({}^{\beta}e_1\ \texttt{||}\ {}^{\gamma}e_2)$	$before(\beta) = before(\alpha)$ $before(\gamma) = false(\beta)$ $false(\alpha) = false(\gamma)$ $true(\alpha) = true(\beta) \cap true(\gamma)$

Table 1: Definite assignment for boolean expressions

${}^{\alpha}stm$. We will often refer to expressions and statements using their labels. In order to precisely specify all the cases of definite assignment, static functions *before*, *after*, *true*, *false* and *vars* are computed at compile time. Note that *true* and *false* are only for boolean expressions. These functions assign sets of variables to each expression or statement α and have the following meanings. $before(\alpha)$ contains the local variables definitely assigned before the evaluation of α and $after(\alpha)$ the variables definitely assigned after the evaluation of α when α completes normally. $true(\alpha)$ and $false(\alpha)$ consist of the variables definitely assigned after the evaluation of α when α evaluates to `true` and `false`, respectively. $vars(\alpha)$ contains the local variables in the scope of which α is.

We skip those language constructs (e.g. `foreach`, `for`, `do`, `switch`, `++`, `--`) whose analysis is similar to the one of the constructs dealt with explicitly in our framework. Note that the definite assignment analysis for variables of a struct type is a little bit different: such a local variable is considered definitely assigned iff all its instance fields are definitely assigned. The structs are not considered here but their detailed analysis is included in the full technical report [Fru03b].

A equation is given by initial conditions: for the method body *mb* of *meth* we have $before(mb) = \emptyset$. Actually we should consider the set of value and reference parameters of *meth* but there is no worry that an access to any of them could cause troubles since when *meth* is invoked they are supposed to be definitely assigned [Wil03, §5.1].

For the other expressions and statements in *mb*, instead of explaining how the functions are computed,

${}^{\alpha}$*exp*	the data flow equations
loc	$\mathit{after}(\alpha) = \mathit{before}(\alpha)$
lit	$\mathit{after}(\alpha) = \mathit{before}(\alpha)$
$(loc = {}^{\beta}e)$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta) \cup \{loc\}$
$(loc\ op = {}^{\beta}e)$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta)$
$({}^{\beta}e_0\ \texttt{?}\ {}^{\gamma}e_1\ \texttt{:}\ {}^{\delta}e_2)$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma) = \mathit{true}(\beta)$ $\mathit{before}(\delta) = \mathit{false}(\beta)$ $\mathit{after}(\alpha) = \mathit{after}(\gamma) \cap \mathit{after}(\delta)$
$c.f$	$\mathit{after}(\alpha) = \mathit{before}(\alpha)$
`ref` ${}^{\beta}$*exp*	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta)$
`out` ${}^{\beta}$*exp*	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta)$
$c.m({}^{\beta_1}arg_1, \ldots, {}^{\beta_k}arg_k)$	$\mathit{before}(\beta_1) = \mathit{before}(\alpha)$ $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i)$, $i = \overline{1, k-1}$ $\mathit{after}(\alpha) = \mathit{after}(\beta_k) \cup$ $\cup\, \mathit{OutParams}(arg_1, \ldots, arg_k)$

Table 2: Definite assignment for arbitrary expressions

we simply state the equations they have to satisfy. Table 1 contains the equations for boolean expressions (see [Fru03b] for details). In addition, we have for all expressions in Table 1 the equation $\mathit{after}(\alpha) = \mathit{true}(\alpha) \cap \mathit{false}(\alpha)$. For a boolean expression α which is not an instance of one of the expressions in Table 1, we have $\mathit{true}(\alpha) = \mathit{after}(\alpha)$ and $\mathit{false}(\alpha) = \mathit{after}(\alpha)$.

Table 2 lists the equations specific to arbitrary expressions where *loc* stands for a local variable and *lit* for a literal. Note that following a method invocation, the `out` parameters $\mathit{OutParams}(arg_1, \ldots, arg_k)$ are definitely assigned. In cases not stated in Tables 1,2, if ${}^{\alpha}$*exp* is an expression with *direct subexpressions* ${}^{\beta_1}e_1, \ldots, {}^{\beta_n}e_n$, then the left-to-right evaluation scheme yields the *general data flow equations*: $\mathit{before}(\beta_1) = \mathit{before}(\alpha)$, $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i)$, $i = \overline{1, n-1}$ and $\mathit{after}(\alpha) = \mathit{after}(\beta_n)$.

The equations specific to every statement can be found in Table 3. We assume that `try` statements are either `try-catch` or `try-finally` statements (see [Bor03] for a justification of this assumption). Special attention is paid to the labeled statement. The set of variables definitely assigned before executing a labeled statement consists of the variables definitely assigned both after the previous statement and before each corresponding `goto` statement or after any of the `finally` blocks of `try-finally` statements in which the `goto` is embedded (if any). This can be formalized as follows. For two statements α and β, we consider $\mathit{Fin}(\alpha, \beta)$ to be the list $[\gamma_1, \ldots, \gamma_n]$

${}^{\alpha}$*stm*	the data flow equations
;	$\mathit{after}(\alpha) = \mathit{before}(\alpha)$
$({}^{\beta}$*exp*`;`$)$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta)$
$\{{}^{\beta_1}stm_1 \ldots {}^{\beta_n}stm_n\}$	$\mathit{before}(\beta_1) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta_n) \cap \mathit{vars}(\alpha)$ $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i) \cap$ $\cap\, \mathit{goto}(\beta_{i+1}), i = \overline{1, n-1}$
`if` $({}^{\beta}$*exp*$)$ ${}^{\gamma}stm_1$ `else` ${}^{\delta}stm_2$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma) = \mathit{true}(\beta)$ $\mathit{before}(\delta) = \mathit{false}(\beta)$ $\mathit{after}(\alpha) = \mathit{after}(\gamma) \cap \mathit{after}(\delta)$
`while` $({}^{\beta}$*exp*$)$ ${}^{\gamma}$*stm*	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma) = \mathit{true}(\beta)$ $\mathit{after}(\alpha) = \mathit{false}(\beta) \cap \mathit{break}(\alpha)$
`goto L;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`break;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`continue;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`return;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`return` ${}^{\beta}$*exp*`;`	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`throw;`	$\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`throw` ${}^{\beta}$*exp*`;`	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{vars}(\alpha)$
`try` ${}^{\beta}$*block* `catch`$(E_1\ x_1)$ ${}^{\gamma_1}$*block*$_1$. : `catch`$(E_n\ x_n)$ ${}^{\gamma_n}$*block*$_n$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma_i) = \mathit{before}(\alpha) \cup \{x_i\}$ $i = \overline{1, n}$ $\mathit{after}(\alpha) = \mathit{after}(\beta) \cap$ $\cap \bigcap_{i=1}^{n} \mathit{after}(\gamma_i)$
`try` ${}^{\beta}$*block*$_1$ `finally` ${}^{\gamma}$*block*$_2$	$\mathit{before}(\beta) = \mathit{before}(\alpha)$ $\mathit{before}(\gamma) = \mathit{before}(\alpha)$ $\mathit{after}(\alpha) = \mathit{after}(\beta) \cup$ $\cup\, \mathit{after}(\gamma)$

Table 3: Definite assignment for statements

of `finally` blocks of all `try-finally` statements in the innermost to outermost order from α to β. Then we define the set $\mathit{JoinFin}(\alpha, \beta)$ of definitely assigned variables after the execution of all these `finally` blocks: $\bigcup_{\gamma \in \mathit{Fin}(\alpha,\beta)} \mathit{after}(\gamma)$. Further, we define the set *goto* for a statement β. For a labeled statement ${}^{\beta}L$: *stm*, the set $\mathit{goto}(\beta)$ is given by $\bigcap_{\alpha \texttt{goto L;}} (\mathit{before}(\alpha) \cup \mathit{JoinFin}(\alpha, \beta))$ where we take only the `goto` statements in the scope of β. For all the other statements, as well for a labeled statement with no `goto` statements, $\mathit{goto}(\beta)$ is the universal set $\mathit{vars}(\beta)$. Now we are able to state the equation $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i) \cap \mathit{goto}(\beta_{i+1})$ from Table 3. In case of a labeled statement, the equation formalizes the above stated idea while for a non-labeled statement becomes $\mathit{before}(\beta_{i+1}) = \mathit{after}(\beta_i)$.

The following example is a simplification of an example from the C# Specification [Wil03, §5.3.3.15]:

```
int i;
δtry {α goto L;}
finally γ{i = 3;}
βL:Console.WriteLine(i);
```

The C# Specification states that `i` is definitely assigned before β , i.e. `i` $\in$ *before*(β). Our equation *before*(β) $=$ *after*$(\delta) \cap$ *goto*(β) led us to the same conclusion since *goto*$(\beta) =$ *before*$(\alpha) \cup$ *after*(γ) and `i` $\in$ *after*$(\gamma) \subseteq$ *after*(δ) (see the equations for a `try-finally` in Table 3). Surprisingly, the example is rejected by the C# compilers of .NET Framework 1.0 and Rotor: we get the error that `i` is unassigned. This problem was fixed in .NET Framework 1.1 but still exists in Rotor.

The following explanation holds for the equation *after*$(\alpha) =$ *after*$(\beta_n) \cap$ *vars*(α) corresponding to a block of statements: the local variables which are definitely assigned after the normal execution of the block are the variables which are definitely assigned after the execution of the last statement of the block. However, the variables must still be in the scope of a declaration. Thus, let us consider the example:

```
{α{int i;i = 1;} {int i;i = 2 * βi;}}
```

The variable `i` is not in *after*(α) since at the end of α, `i` is not in the scope of a declaration. Thus `i` $\notin$ *before*(β) and the block is rejected.

The idea for the equation which computes *after*(α) of a `while` statement α, is similar with that for a labeled statement. Similarly with the set *goto*, we define the set *break*(α) to be the set of variables definitely assigned before all corresponding `break` statements (and possibly after appropriate `finally` blocks). This means that the set *break*(α) is given by $\bigcap_{\beta\texttt{break;}}$(*before*$(\beta) \cup$ *JoinFin*(β, α)) where we take only the `break` statements for which α is the nearest enclosing `while`. If the `while` statement does not have any `break` statements, then we define *break*$(\alpha) =$ *vars*(α). With this definition of *break*(α), we have the equation for *after*(α) as stated in Table 3.

There is one more technical detail to be decided. Suppose we want to state the equation for *after* of a jump statements. Let α be the following statement:

```
if(b) γ{i = 1;} else δreturn;
```

It is clear that, the variables definitely assigned after α are the variables definitely assigned after the `then` branch and since our equation takes the intersection of *after*(γ) and *after*(δ), it is obvious that one has to require the set-intersection identity for *after*(δ). That is why we adopt the convention that *after*(α) is the universal set *vars*(α) for any jump statement α.

3 THE MAXIMAL FIXED POINT

The computation of the sets of definitely assigned variables from the data flow equations described in Section 2 is relatively straightforward. The key difference with respect to Java is the `goto` statement which brings more complexity to the analysis. Since the `goto` statement makes loops possible, the system of data flow equations does not have always a unique solution. Here is an example: if we consider a method which takes no parameters and has the following body

```
{αint i = 1; βL: γgoto L;}
```

then we have the following equations *after*$(\alpha) = \{$`i`$\}$, *before*$(\beta) =$ *after*$(\alpha) \cap$ *before*(γ) and *before*$(\gamma) =$ *before*(β). After some simplification we find that *before*$(\beta) = \{$`i`$\} \cap$ *before*(β) and therefore we get two solutions for *before*(β) (and also for *before*(γ)): $\emptyset$ and $\{$`i`$\}$. This is the reason we perform a fixed point iteration - which is not the case in Java. The set of variables definitely assigned after α is $\{$`i`$\}$ and since β does not 'unassign' `i`, `i` is obviously assigned when we enter β. Consideration of the example and the definition of *definitely assigned* show that the most informative solution is $\{$`i`$\}$ and therefore the solution we require is the maximal fixed point *MFP*.

In the rest of this section we show that there always exists a maximal fixed point for our data flow equations. In order to prove the existence, one needs first to define the function F which encapsulates the equations. For the domain and codomain of this function, we need the set *Vars*(*meth*) of all local variables from the method body *mb*. We define the function $F : D \to D$ with $D = \mathcal{P}($*Vars*$(\textit{meth}))^r$ such that $F(X_1, \ldots, X_r) = (Y_1, \ldots, Y_r)$, where *r* is the number of equations and the sets Y_i are defined by the data flow equations. For example in the case of an `if-then-else` statement, if the equation for the *after* set of this statement is the i-th data flow equation, then the set of variables Y_i is defined by $Y_i = X_j \cap X_k$ where j and k are the indices of the equations for the *after* sets of the `then` and the `else` branch, respectively. Note that the sets *vars* are interpreted as constants.

We define now the relation $\sqsubseteq$ on D to be the pointwise set inclusion relation: if $(X_1, \ldots, X_r) \in D$ and $(X_1', \ldots, X_r') \in D$, then we have $(X_1, \ldots, X_r) \sqsubseteq (X_1', \ldots, X_r')$ if $X_i \subseteq X_i'$ for all $i = \overline{1, r}$. We are now able to prove the following result:

Lemma 1 $(D, \sqsubseteq)$ *is a finite lattice.*

Proof. D is finite since for a given method body we have a finite number of equations and local variables and on the other hand, D is a lattice since it is a product of lattices: $(\mathcal{P}($*Vars*$(\textit{meth})), \subseteq)$ is a *poset* since the set inclusion is a partial order and for every two sets $X, Y \in \mathcal{P}($*Vars*$(\textit{meth}))$ there exists a lower bound $(X \cap Y)$ and an upper bound $(X \cup Y)$. □

The following result will help us conclude the existence of the maximal fixed point.

Lemma 2 *The function F is monotonic on $(D, \sqsubseteq)$.*

Proof. In order to prove the monotonicity of $F = (F_1, \ldots, F_r)$, it suffices to remark that the components F_i are monotonic functions. This holds since they consist only of set intersections and unions which are monotonic (see the form of the equations). □

The next result guarantees the existence of the maximal fixed point solution for our data flow equations:

Lemma 3 *The function F has a unique maximal fixed point MFP $\in D$.*

Proof. $(D, \sqsubseteq)$ is a finite lattice (Lemma 1) and therefore a complete lattice. But in a complete lattice, every monotonic function has a unique maximal fixed point (known also as *the greatest fixed point*). In our case, F is monotonic (Lemma 2) and the maximal fixed point *MFP* is given by $\bigcap_k F^{(k)}(1_D)$. Here 1_D is the r-tuple $(\mathit{Vars}(\mathit{meth}), \ldots, \mathit{Vars}(\mathit{meth}))$, i.e. the top element of the lattice D. □

From now on, for an expression or statement α we denote by $\text{MFP}_b(\alpha)$, $\text{MFP}_a(\alpha)$, $\text{MFP}_t(\alpha)$ and $\text{MFP}_f(\alpha)$ the components of *MFP* corresponding to *before*(α), *after*(α), *true*(α) and *false*(α), respectively.

4 THE CONTROL FLOW GRAPH

The main result we want to prove is that, for an arbitrary expression or statement, the sets of local variables MFP_b, MFP_a (and MFP_t, MFP_f for boolean expressions) correspond indeed to sets of *definitely assigned* variables, i.e. variables which are assigned on every possible execution path to the appropriate point. The considered paths are based on the control flow graph. The nodes of the graph are actually points associated with every expression and statement. We suppose that every expression or statement α is characterized by an *entry* point $\mathcal{B}(\alpha)$ and an *end* point $\mathcal{A}(\alpha)$. Beside these two points, a boolean expression α has two more points: a *true* point $\mathcal{T}(\alpha)$ (used when α evaluates to `true`) and a *false* point $\mathcal{F}(\alpha)$ (used when α evaluates to `false`). The edges of the graph are given by the *control transfer* defined in the C# Specification [Wil03, §8]. We show in Tables 4 and 5 the edges specific to each boolean and arbitrary expression, respectively. If the expression α is not an instance of one expression in these tables (e.g. $\mathit{exp}_1 \,|\, \mathit{exp}_2$) and has the *direct subexpressions* $\beta_1, \ldots, \beta_n$, then the left-to-right evaluation scheme adds to the flow graph also the following edges: $(\mathcal{B}(\alpha), \mathcal{B}(\beta_1))$,$(\mathcal{A}(\beta_n), \mathcal{A}(\alpha))$ and $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1}))$, $i = \overline{1, n-1}$.

For each boolean expression α in Table 4, we have supplementary edges: $(\mathcal{T}(\alpha), \mathcal{A}(\alpha))$, $(\mathcal{F}(\alpha), \mathcal{A}(\alpha))$ which connect the boolean points of α to the end point of α. These edges are necessary for the control transfer in cases when it does not matter whether α evaluates to `true` or `false`. For example, if β is the method invocation `c.m(true)` and α is the argument `true`, then the control is transferred from the end point of the last argument - that is $\mathcal{A}(\alpha)$ - to the end point of the method invocation - that is $\mathcal{A}(\beta)$. But since in Table 4 we have no edge leading to $\mathcal{A}(\alpha)$, we need to define also the supplementary edge $(\mathcal{T}(\alpha), \mathcal{A}(\alpha))$.

For a boolean expression α which is not an instance of any expression from Table 4, we add to the graph the edges $(\mathcal{A}(\alpha), \mathcal{T}(\alpha))$, $(\mathcal{A}(\alpha), \mathcal{F}(\alpha))$. They are needed if control is transferred from a boolean expression α to different points depending on whether α evaluates to `true` or `false`. For example, if α is of the form exp_1 `|` exp_2 and occurs in $^{\beta}$`(!(`exp_1 `|` exp_2`))`,

$^{\alpha}$*exp*	edges
`true`	$(\mathcal{B}(\alpha), \mathcal{T}(\alpha))$
`false`	$(\mathcal{B}(\alpha), \mathcal{F}(\alpha))$
$(\,!\,{}^{\beta}e)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{F}(\beta), \mathcal{T}(\alpha))$ $(\mathcal{T}(\beta), \mathcal{F}(\alpha))$
$({}^{\beta}e_0 \,?\, {}^{\gamma}e_1 \,:\, {}^{\delta}e_2)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma))$, $(\mathcal{F}(\beta), \mathcal{B}(\delta)), (\mathcal{T}(\gamma), \mathcal{T}(\alpha))$, $(\mathcal{T}(\delta), \mathcal{T}(\alpha)), (\mathcal{F}(\gamma), \mathcal{F}(\alpha))$, $(\mathcal{F}(\delta), \mathcal{F}(\alpha))$
$({}^{\beta}e_1$ `&&` $^{\gamma}e_2)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma))$, $(\mathcal{F}(\beta), \mathcal{F}(\alpha)), (\mathcal{T}(\gamma), \mathcal{T}(\alpha))$, $(\mathcal{F}(\gamma), \mathcal{F}(\alpha))$
$({}^{\beta}e_1$ `||` $^{\gamma}e_2)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{T}(\alpha))$, $(\mathcal{F}(\beta), \mathcal{B}(\gamma)), (\mathcal{T}(\gamma), \mathcal{T}(\alpha))$, $(\mathcal{F}(\gamma), \mathcal{F}(\alpha))$

Table 4: Control flow for boolean expressions

$^{\alpha}$*exp*	edges
loc	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
lit	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
$(\mathit{loc} = {}^{\beta}e)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$(\mathit{loc}\ \mathit{op} = {}^{\beta}e)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$({}^{\beta}e_0 \,?\, {}^{\gamma}e_1 \,:\, {}^{\delta}e_2)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma))$ $(\mathcal{F}(\beta), \mathcal{B}(\delta)), (\mathcal{A}(\gamma), \mathcal{A}(\alpha))$, $(\mathcal{A}(\delta), \mathcal{A}(\alpha))$
$c.f$	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
`ref` $^{\beta}$*exp*	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
`out` $^{\beta}$*exp*	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
$c.m(^{\beta_1}\mathit{arg}_1, \ldots, ^{\beta_k}\mathit{arg}_k)$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta_1)), (\mathcal{A}(\beta_k), \mathcal{A}(\alpha))$, $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1})), i = \overline{1, k-1}$

Table 5: Control flow for arbitrary expressions

then the control is transferred from $\mathcal{F}(\alpha)$ to $\mathcal{T}(\beta)$ (if α evaluates to `false`) or from $\mathcal{T}(\alpha)$ to $\mathcal{F}(\beta)$ (if α evaluates to `true`). The necessity of the edges $(\mathcal{A}(\alpha), \mathcal{T}(\alpha))$, $(\mathcal{A}(\alpha), \mathcal{F}(\alpha))$ arises since, so far we have defined for exp_1 | exp_2 only edges to $\mathcal{A}(\alpha)$.

Table 6 introduces the edges of the control flow graph for each statement. Note that we assume that the boolean constant expressions are replaced by `true` or `false` in the abstract syntax tree. For example, we consider that `true||b` is replaced by `true` in the following `if` statement:

```
α if β(true||b) δi = 1;
  else γ{int j = i;}
```

Although the new considered test (i.e. `true`) cannot evaluate to `false`, we still add to the graph the edge $(\mathcal{F}(\beta), \mathcal{B}(\gamma))$ since anyway the false point of `true` is not reachable (see Table 4). In the presence of `finally` blocks, the jump statements `goto`, `break` and `continue` bring more complexity to the graph. Whenever such a jump statement exits one or more `try` blocks with associated `finally` blocks, the control is transferred first to the `finally` block (if any) of the innermost `try` statement. Further, if the control reaches the end point of the `finally`, then it is transferred to the next (with respect to the innermost to outermost order of the `try` statements) `finally` block and so on. If the control reaches the end point of the last `finally` block, then it is transferred to the target of the jump statement. For these control transfers we have special edges in our graph. But one needs to take care to some detail: these special edges cannot be used for paths other than those which connect the jump statement with its target. In other words, if a path uses such an edge, then necessarily the path contains the entry point of the jump statement. For this reason, we say that an edge e is *conditioned* by a point *i* with the meaning that e can be used only in paths that contain *i*. If we do not make this restriction, then $[\mathcal{B}(mb)\mathcal{B}(\alpha_1)\mathcal{B}(\alpha_2)\mathcal{B}(\alpha_3)\mathcal{B}(\alpha_4)\mathcal{B}(\alpha_5)\mathcal{A}(\alpha_5)\mathcal{B}(\alpha_6)]$ would be a possible execution path to the labeled statement in the following method body

```
α1 try α2 {
   α3(α4(i = 1);)
   goto L;
} finally α5{}
α6 L:Console.WriteLine(i);
```

in the theoretical case when the evaluation of α_4 would throw an exception. But this does not match the control transfer described in the C# Specification.

The following sets introduce the above described edges. If α and β are two statements and $Fin(\alpha, \beta)$ is the list $[\gamma_1, \ldots, \gamma_n]$, then the set $ThroughFin_b(\alpha, \beta)$ consists of the edges $(\mathcal{B}(\alpha), \mathcal{B}(\gamma_1))$, $(\mathcal{A}(\gamma_n), \mathcal{B}(\beta))$, $(\mathcal{A}(\gamma_i), \mathcal{B}(\gamma_{i+1}))$, $i = \overline{1, n-1}$ all conditioned by $\mathcal{B}(\alpha)$ and the set $ThroughFin_a(\alpha, \beta)$ has the edges $(\mathcal{B}(\alpha), \mathcal{B}(\gamma_1))$, $(\mathcal{A}(\gamma_n), \mathcal{A}(\beta))$, $(\mathcal{A}(\gamma_i), \mathcal{B}(\gamma_{i+1}))$, $i = \overline{1, n-1}$ all conditioned by $\mathcal{B}(\alpha)$. If $Fin(\alpha, \beta)$ is empty, then the set $ThroughFin_b(\alpha, \beta)$ has only the edge $(\mathcal{B}(\alpha), \mathcal{B}(\beta))$ while $ThroughFin_a(\alpha, \beta)$ refers to the edge $(\mathcal{B}(\alpha), \mathcal{A}(\beta))$.

$^{\alpha}$*stm*	edges
;	$(\mathcal{B}(\alpha), \mathcal{A}(\alpha))$
($^{\beta}$*exp*;)	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
{$^{\beta_1}$ *stm*$_1$ ... $^{\beta_n}$ *stm*$_n$}	$(\mathcal{B}(\alpha), \mathcal{B}(\beta_1)), (\mathcal{A}(\beta_n), \mathcal{A}(\alpha)),$ $(\mathcal{A}(\beta_i), \mathcal{B}(\beta_{i+1})), i = \overline{1, n-1}$
`if` ($^{\beta}$*exp*) $^{\gamma}$*stm*$_1$ `else` $^{\delta}$*stm*$_2$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)),$ $(\mathcal{F}(\beta), \mathcal{B}(\delta)), (\mathcal{A}(\gamma), \mathcal{A}(\alpha)),$ $(\mathcal{A}(\delta), \mathcal{A}(\alpha))$
`while` ($^{\beta}$*exp*) $^{\gamma}$*stm*	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{T}(\beta), \mathcal{B}(\gamma)),$ $(\mathcal{F}(\beta), \mathcal{A}(\alpha)), (\mathcal{A}(\gamma), \mathcal{A}(\alpha))$
`L:` $^{\beta}$*stm*	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$
`goto L;`	$ThroughFin_b(\alpha, \beta)$, where $^{\beta}$`L:`*stm* is the statement to which α points
`break;`	$ThroughFin_a(\alpha, \beta)$, where β is the nearest enclosing `while` wrt α
`continue;`	$ThroughFin_b(\alpha, \beta)$, where β is the nearest enclosing `while` wrt α
`return;`	no edges
`return` $^{\beta}$*exp*;	$(\mathcal{B}(\alpha), \mathcal{B}(\beta))$
`throw;`	no edges
`throw` $^{\beta}$*exp*;	$(\mathcal{B}(\alpha), \mathcal{B}(\beta))$
`try` $^{\beta}$*block* `catch`(E_1 x_1) $^{\gamma_1}$*block*$_1$ ⋮ `catch`(E_n x_n) $^{\gamma_n}$*block*$_n$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{A}(\beta), \mathcal{A}(\alpha))$ $(\mathcal{B}(\alpha), \mathcal{B}(\gamma_i)), (\mathcal{A}(\gamma_i), \mathcal{A}(\alpha)),$ $i = \overline{1, n}$
`try` $^{\beta}$*block*$_1$ `finally` $^{\gamma}$*block*$_2$	$(\mathcal{B}(\alpha), \mathcal{B}(\beta)), (\mathcal{B}(\alpha), \mathcal{B}(\gamma)),$ $(\mathcal{A}(\beta), \mathcal{B}(\gamma))$ and $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$ conditioned by $\mathcal{A}(\beta)$

Table 6: Control flow for statements

Note that in Table 6, for `goto` and `continue`, the set of edges $ThroughFin_b$ is added to the graph, since after executing the `finally` blocks the control is transferred to the entry point of the labeled statement and `while` statement, respectively, while in case of `break` the set $ThroughFin_a$ is considered, since at the end, the control is transferred to the end point of the `while` statement.

There are two more remarks concerning the `try` statement. Since in a `try` block can anytime occur a reason for abruption (e.g. an exception), we should

have edges from every point in a `try` block to: every associate `catch` block, every `catch` of enclosing `try` statements (if the `catch` clause matches the type of the exception) and to every associate `finally` block (if no `catch` clause matches the type of the exception). We do not consider all these edges, since from the point of view of the definite assignment analysis which is in particular an 'over all paths' analysis, it is equivalent to consider only one edge to the entry points of the `catch` and `finally` blocks - from the entry point of the `try` block (see Table 6).

The next remark is concerning the end point $\mathcal{A}(\alpha)$ of a `try-finally` statement α. The C# Specification states in [§8.10] that $\mathcal{A}(\alpha)$ is reachable only if both end points of the `try` block β and `finally` block γ are reachable. The only edge to $\mathcal{A}(\alpha)$ is $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$ and we know that the `finally` block can be reached either through a jump or through a normal completion of the `try` block. In case of a jump, if control reaches the end point $\mathcal{A}(\gamma)$ of the `finally`, then it is transferred further to the target of statement which generated the jump and not to $\mathcal{A}(\alpha)$. This means that all paths to $\mathcal{A}(\alpha)$ contain also the end point $\mathcal{A}(\beta)$ of the `try` block. That is why we require that the edge $(\mathcal{A}(\gamma), \mathcal{A}(\alpha))$ is *conditioned* by $\mathcal{A}(\beta)$ (see Table 6) - otherwise in the following example, $\mathcal{A}(\alpha)$ would be reachable in our graph (under the assumption that $\mathcal{B}(\alpha)$ is reachable):

$^{\alpha}$`try` $^{\beta}${`goto L;`} `finally` $^{\gamma}${}

We define now the sets of *valid* paths to all points in the method body. We will not consider all the paths in the graph but only the *valid* paths - that is the paths p for which the following is true: if p uses a *conditioned edge* then it contains also the point which conditions the edge. If α is an expression or a statement, then $path_b(\alpha)$ and $path_a(\alpha)$ are the sets of all valid paths from the entry point of the method body $\mathcal{B}(mb)$ to the entry point $\mathcal{B}(\alpha)$ and to the end point $\mathcal{A}(\alpha)$ of α, respectively. Moreover, if α is a boolean expression, then $path_t(\alpha)$ and $path_f(\alpha)$ are the sets of all valid paths from $\mathcal{B}(mb)$ to the true point $\mathcal{T}(\alpha)$ and to the false point $\mathcal{F}(\alpha)$ of α, respectively.

5 THE CORRECTNESS OF THE ANALYSIS

We prove that, when a C# compiler relies on the sets MFP_b, MFP_a, MFP_t and MFP_f derived from the maximal fixed point of the equations in Section 2, the risk of accessing the value of an unassigned variable does not exist. The correctness means that, if the analysis infers a variable as definitely assigned at a certain program point, then this variable will actually be assigned at that point during every execution of the program, i.e. on every path. A variable *loc* is assigned on a path if the path contains an *initialization* of *loc*: a simple assignment to *loc*, a method invocation for which *loc* is an `out` parameter or a `catch` clause whose exception variable is *loc*. We prove actually more than the correctness. We show that the components of the maximal fixed point are exactly (not only a *safe approximation* of) the sets of variables for which there is an *initialization* on every path to the appropriate point. To formalize this, we define the following sets. If α is an arbitrary expression or statement, then $\text{AP}_b(\alpha)$ and $\text{AP}_a(\alpha)$ denote the sets of variables in $vars(\alpha)$ (the variables in the scope of which α is) for which there exists an initialization on every path in $path_b(\alpha)$ and in $path_a(\alpha)$, respectively. For a boolean expression α, we have two more sets: $\text{AP}_t(\alpha)$ and $\text{AP}_f(\alpha)$ are defined similarly as above, but with respect to paths in $path_t(\alpha)$ and $path_f(\alpha)$, respectively.

The following lemma is proved by induction over the abstract syntax tree, starting from the root of the method body. It claims that, the MFP sets of an expression or statement α, consist of variables in the scope of which α is (see [Fru03b] for details).

Lemma 4 *For every expression or statement α we have $\text{MFP}_b(\alpha) \subseteq vars(\alpha)$ and $\text{MFP}_a(\alpha) \subseteq vars(\alpha)$. Moreover, if α is a boolean expression, then we have also $\text{MFP}_t(\alpha) \subseteq vars(\alpha)$ and $\text{MFP}_f(\alpha) \subseteq vars(\alpha)$.*

The correctness of the definite assignment analysis in C# is proved in the next theorem, which claims that the analysis is a *safe approximation*.

Theorem 1 ***(safe approximation)*** *For every expression or statement α, the following relations are true: $\text{MFP}_b(\alpha) \subseteq \text{AP}_b(\alpha)$ and $\text{MFP}_a(\alpha) \subseteq \text{AP}_a(\alpha)$. Moreover, if α is a boolean expression, then we have $\text{MFP}_t(\alpha) \subseteq \text{AP}_t(\alpha)$ and $\text{MFP}_f(\alpha) \subseteq \text{AP}_f(\alpha)$.*

Proof. We consider the following definitions. The set $\text{AP}_b^n(\alpha)$ is defined in the same way as $\text{AP}_b(\alpha)$, except that we consider only the paths of length less or equal than n. Similarly, we define also the sets $\text{AP}_a^n(\alpha)$, $\text{AP}_t^n(\alpha)$, $\text{AP}_f^n(\alpha)$ (analogously, we have definitions for the sets of paths $path^n$). According to these definitions, the following set equalities hold for an arbitrary α: $\text{AP}_b(\alpha) = \bigcap_n \text{AP}_b^n(\alpha)$, $\text{AP}_a(\alpha) = \bigcap_n \text{AP}_a^n(\alpha)$ and if α is a boolean expression, then $\text{AP}_t(\alpha) = \bigcap_n \text{AP}_t^n(\alpha)$ and $\text{AP}_f(\alpha) = \bigcap_n \text{AP}_f^n(\alpha)$. Therefore to complete the proof, it suffices to show for every n: if α is an expression or statement, then $\text{MFP}_b(\alpha) \subseteq \text{AP}_b^n(\alpha)$ and $\text{MFP}_a(\alpha) \subseteq \text{AP}_a^n(\alpha)$ and in addition, if α is a boolean expression, $\text{MFP}_t(\alpha) \subseteq \text{AP}_t^n(\alpha)$ and $\text{MFP}_f(\alpha) \subseteq \text{AP}_f^n(\alpha)$. This is done by induction on n.

Basis of induction: $[\mathcal{B}(mb)]$ is the only path of length 1 (the entry point of the method body). There is no initialization of any local variable on this path and therefore we have $\text{AP}_b^1(mb) = \emptyset$ which satisfies $\text{MFP}_b(mb) \subseteq \text{AP}_b^1(mb)$ since from the equations $\text{MFP}_b(mb) = \emptyset$. From the definition of AP_a^1, we get $\text{AP}_a^1(mb) = vars(mb) = \emptyset$ and from the equations of a block, we derive also $\text{MFP}_a(mb) \subseteq vars(mb)$

and implicitly $\text{MFP}_a(mb) \subseteq \text{AP}_a^1(mb)$. If $\alpha \neq mb$, then $\text{AP}_b^1(\alpha) = \text{AP}_a^1(\alpha) = vars(\alpha)$ and $\text{AP}_t^1(\alpha) = \text{AP}_f^1(\alpha) = vars(\alpha)$ (if α is a boolean expression) and the basis of induction is complete (see Lemma 4).

Induction step: we prove here only $\text{MFP}_b(\beta_{i+1}) \subseteq \text{AP}_b^{n+1}(\beta_{i+1})$ for a labeled statement β_{i+1} in a block (see Table 3). Let *loc* be a local variable in $\text{MFP}_b(\beta_{i+1})$. If there are no `goto` statements pointing to β_{i+1}, then the proof is the same as for a `while` statement with no associated `continue` statements (see [Fru03b]). If there are `goto` statements which point to β_{i+1}, we prove that there exists an initialization of *loc* on every path to $\mathcal{B}(\beta_{i+1})$ of length at most $n+1$ that passes through a `goto` statement and possibly through `finally` blocks of enclosing `try` statements. Let p be such a path containing a `goto` statement α. The equations in Table 3 imply $loc \in goto(\beta_{i+1})$ and further $loc \in \text{MFP}_b(\alpha) \cup JoinFin(\alpha, \beta_{i+1})$. If there are no `finally` blocks in $Fin(\alpha, \beta_{i+1})$, then $JoinFin(\alpha, \beta_{i+1}) = \emptyset$ and implicitly $loc \in \text{MFP}_b(\alpha)$. Using the induction hypothesis, we obtain $loc \in \text{AP}_b^n(\alpha)$ and therefore p should contain at least one initialization of *loc*. If $Fin(\alpha, \beta_{i+1})$ is non-empty, i.e. $Fin = [\gamma_1, \ldots, \gamma_k]$, then from the definition of the set $JoinFin(\alpha, \beta_{i+1})$, we get $loc \in \text{MFP}_b(\alpha) \cup \bigcup_{j=1}^{k} \text{MFP}_a(\gamma_j)$. The case $loc \in \text{MFP}_b(\alpha)$ has been previously analyzed. If there is a `finally` block γ_j such that $loc \in \text{MFP}_a(\gamma_j)$, then we get $loc \in \text{AP}_a^n(\gamma_j)$ from the induction hypothesis. And since necessarily p contains $\mathcal{A}(\gamma_j)$, we are sure that p has one initialization of *loc*. Thus, we showed that each path to $\mathcal{B}(\beta_{i+1})$ of length at most $n+1$, contains an initialization of *loc*, i.e $loc \in \text{AP}_b^{n+1}(\beta_{i+1})$. □

We can prove actually more: the *MFP* solution is not only an approximation of *AP* but it is perfect (Theorem 3). For this, we need also the following theorem which states that the *MFP* solution contains the local variables which are initialized over *all possible paths*.

Theorem 2 *For every expression or statement α, the following relations are true:* $\text{AP}_b(\alpha) \subseteq \text{MFP}_b(\alpha)$ *and* $\text{AP}_a(\alpha) \subseteq \text{MFP}_a(\alpha)$. *Moreover, if α is a boolean expression, then we have also* $\text{AP}_t(\alpha) \subseteq \text{MFP}_t(\alpha)$ *and* $\text{AP}_f(\alpha) \subseteq \text{MFP}_f(\alpha)$.

Proof. Tarski's fixed point theorem states that *MFP* is the lowest upper bound (with respect to $\sqsubseteq$) of the set $Ext(F) = \{X \in D \mid X \sqsubseteq F(X)\}$. It suffices to show that the *r*-tuple consisting of the *AP* sets is an element of $Ext(F)$ since *MFP* is in particular an upper bound of this set. Since $\sqsubseteq$ is the pointwise subset relation, the idea is to prove, for the data flow equations in Tables 1, 2, 3, the left-to-right subset relations where instead of the sets *before*, *after*, *true* and *false* we have the sets AP_b, AP_a AP_t and AP_f, respectively. For the complete proof we refer the reader to [Fru03b]. □

The following result is then an obvious consequence of Theorem 1 and Theorem 2:

Theorem 3 *The maximal fixed point solution of the data flow equations in Tables 1,2,3 represents the sets of local variables which are assigned over all possible execution paths.*

6 CONCLUSION

In this paper, we have formalized the definite assignment analysis of C# by data flow equations. Since the equations do not always have a unique solution, we defined the outcome of the analysis as the solution of a fixed point iteration. We proved that there exists always a maximal fixed point solution MFP. We showed the correctness of the analysis, i.e. MFP is a *safe approximation* of the sets of variables assigned over all possible paths. This is a key property for the type safety of C#. This paper is part of a research project focusing on formalizing and verifying important aspects of C#. So far, we have an ASM model for the operational semantics of C# in [Bor03]. During the attempts to build this model, there were discovered in [Fru03a] a few discrepancies between the C# Specification and different implementations of C#.

References

[Bor03] E. Börger, N. G. Fruja, V. Gervasi, R. F. Stärk. A High–Level Modular Definition of the Semantics of C#. Accepted for publication in journal Theoretical Computer Science, 2003

[Fru03a] N. G. Fruja. Specification and Implementation Problems for C#. In B. Thalheim and W. Zimmermann, editors, Abstract State Machines 2004, LNCS. Springer, 2004.

[Fru03b] N. G. Fruja. The correctness of the definite assignment analysis in C#. Technical Report, ETH Zürich. `http://www.inf.ethz.ch/~fruja`

[Gou02] J. Gough. Compiling for the .NET. Common Language Runtime (CLR). Prentice Hall, 2002.

[Gru00] D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Langendoen. Modern Compiler Design. Wiley, 2000

[Nie99] F. Nielson, H.R. Nielson, C. Hankin. Principles of Program Analysis. Springer–Verlag, 1999.

[Sta01] R. F. Stärk, J. Schmid, E. Börger. Java and the Java Virtual Machine–Definition, Verification, Validation. Springer–Verlag, 2001.

[Sch03] N. Schirmer. Java Definite Assignment in Isabelle/HOL. ECOOP Workshop on Formal Techniques for Java–like Programs, 2003.

[Wil03] S. Wiltamuth and A. Hejlsberg. C# Language Specification. MSDN, 2003