3GERP Workshop

Beyond the crystal ball assumption: Towards upgradable ERP systems

Sebastien Vaucouleur,
vaucouleur@itu.dk
IT University of Copenhagen.

September 26, 2008

Abstract

Most software engineering techniques that deal with software products customization are based on *anticipation*: The software designer has to foresee, somehow, the future needs for customization so that other programmers can adapt the software product with as little modifications as possible (programmers hide implementation details behind previously defined interfaces, or alternatively, they refine some pre-defined properties). While practical, this approach is unfortunately not completely satisfactory for *Enterprise Resource Planning* systems (ERPs). These software products have to be customizable for numerous and various local contexts; they cover a very large domain, one that cannot be fully comprehended — hence accurate anticipation is difficult. To solve this problem, an extreme measure is to give the programmers the means to do modifications in place, directly in the source code. This approach trades control for flexibility. Unfortunately, it also makes the customized software product very sensitive to upgrades. We propose a more mitigated solution, that does not require accurate anticipation and yet offers some resilience to evolution of the base software product through the use of code quantification.

We introduce the Eggther framework for customization of evolvable software products in general and ERP systems in particular. Our approach is based on the concept of code query by example. The technology being developed is based on an initial empirical study on practices around ERP systems. We motivate our design choices based on those empirical results, and we show how the proposed solution helps with respect to the upgrade problem.

1 Introduction

We give a very short introduction to *Enterprise Resource Planning Systems systems* (ERPs), and we define informally some of the main concepts. Then we discuss *anticipation* and what we call the *upgrade problem*. Finally, we give the road-map for the rest of paper.

1.1 ERP systems

We assume that the reader has some basic knowledge about ERP systems. We will only recall the details that are directly relevant for the discussion. We refer the reader to Shanks et al. [20] for a detailed treatment of ERP systems. ERP systems are usually defined as being business support systems, that deal with the management of the various functions found in modern companies, such as manufacturing, financial, human resources and customer relationship management. ERP systems are data-oriented: The back-end database is typically seen as the central element of the infrastructure (while there is a trend to give better support for processes, ERP systems remains heavily data-oriented). When a company purchases a license, it can decide to use an ERP system as it is — the ERP system is fully functional and can be used off-the-shelf. Nonetheless, many companies prefer to customize their newly acquired ERP system to the local context in which it will be deployed. The local context can be for example related to the company's unique business model, or to some local regulations: State regulation, industry specific regulations, etc. An alternative is to adapt the company to the ERP system — which does happen in practice.

Customizations can be made directly by customers, but usually they are done by small software houses that specialize in this activity. The competencies of these software houses consist in their knowledge of the ERP system, but most importantly in their knowledge of the vertical domains (accountancy, transport industry, etc.). Their mastery of both the ERP system and of the vertical domains allow them to quickly develop customizations that fit the needs their customers. They typically charge high fees for their services, hence time-to-market is important to the customers.

Our work targets evolvable software products in general, and ERP systems in particular. We grounded our work in the study of two existing ERP systems, Microsoft Dynamics AX and Microsoft Dynamics NAV, that we will call collectively Microsoft Dynamics. Section 2 provides a summary of the empirical study. We refer the reader to [19, 21, 12] for a more complete treatment about Microsoft Dynamics.

1.2 Definitions

Customizations Customizations add some new and un-foreseen features to a software system. We contrast customizations with *configurations* (configurations enable or disable features already present in the program).

Software products A software product is software that can be customized for a specific context. Successful software products are evolving on a regular basis [19].

Base software product The base software product is the software product *before* customizations.

Software product maker The software product maker is the company that design and implement the base software product. In the case of Dynamics, Microsoft is the software product maker.

Partners Partners are software houses that specialize in making customizations for other companies (their customers could be partners themselves or regular customers). The term "partner" recalls the privileged business relationship that they have with the software product maker[1].

Customers Customers will simply refer to the companies which are purchasing an ERP system for their own needs. Typically, customers are also purchasing customizations services or customization code from partners.

Figure 1 on page 4 shows a simple instance of an ERP eco-system. Nodes denote actors in the business model. An edge from a node A to a node B denotes that A is providing some customization code, or a base software product to B. $User_3$ decides to use the software product as it is. $User_4$ and $User_5$ on the other hand use some customization services from $Partner_2$. $Partner_1$ does not deal with customers but only provides customization solutions to some other partners, $Partner_3$ and $Partner_4$. This last one also uses some further customizations made by $Partner_3$. This example is very simplified scenario of the real ecosystem around ERP system: A complex and evolving graph of business entities, where customization code flows from one entity to an other.

Microsoft Dynamics follows this business model. Using this unstructured scheme, Microsoft can scale the scope of their product to a very large market around the globe, and yet can keep its focus on its main competency: The core horizontal functionalities of the ERP system (for example the transactional sub-system, a convenient graphical interface, web services, etc.).

This artificial and simplified example illustrates how complex it would be to attempt to reason about the intent behind a specific part of the system given the current business model — especially since the code base can very large, more than 1 million lines of code, with no explicit specification (modulo some informal documentation).

[1]The terminology around Microsoft Dynamics sometimes makes a distinction between companies that implement customization solutions for particular vertical domain, and companies that make customizations for a single company. For the sake of simplicity, we will ignore this distinction in this paper, and use the generic name of *partner*.

$$\begin{tikzcd}
 & & Software\,product\,maker \arrow[dl] \arrow[dr] \arrow[dddl] \arrow[ddd] \arrow[dddr, bend left] \arrow[ddll, bend right] & & \\
 & Partner_1 \arrow[dl] \arrow[d] & & Partner_2 \arrow[dd] \arrow[ddr] & \\
Partner_3 \arrow[r] \arrow[d] & Partner_4 \arrow[d] & & & \\
User_1 & User_2 & User_3 & User_4 & User_5
\end{tikzcd}$$

Figure 1: A simple instance of an ERP systems ecosystem

1.3 The Crystal ball assumption

At the core of many approaches to evolution and customization within the field of software engineering is, in one form or an other, what we call the *crystal ball assumption*: Software designers are supposed to be equipped with a rather accurate *crystal ball*, by which they can anticipate the future needs for customizations and evolution of their software product. A concrete example in object-oriented programming is the use of virtual methods and factory patterns that are positioned in strategic positions of the code base to deal with *likely* variations. The same need for anticipation (the need for a "crystal ball") can be found in most approaches, whether object-oriented or not [19].

We would like to argue that the problem is particularly prominent in the field of ERP systems: The domain covered by ERP systems is very large and diverse. For example, tax rules are obviously not the same in Denmark as in France. But what about this particular region of France; what about the rules for a particular industry (say textile); what about the rule for this particular time of the year, and what about the combination of any of these special cases? Not only the domain is very large, it is also evolving very quickly: Tax rules typically come and go as new governments are put in place. In short, the domain is very large, and evolving — full anticipation is not an option.

ERP systems can be contrasted with software products that deal with more stable domains. Consider for example graph libraries: Graphs have been thoroughly studied and the concepts and design alternatives around graphs are well comprehended, extensively explored and have been well documented. Of course new important variations around graphs do surface once in a while — but it is relatively rare. Typically, that new variation would simply be incorporated in the next version of the library: What is required in this case is not code customization but *code evolution*.

1.4 Anticipation is not a panacea

Several lines of research have tried to address explicitly the anticipation problem. For example, the work on *Multi-Dimensional Separation of Concerns* by Tarr et al. at IBM Research lead to Hyper/J [18, 19], an ambitious framework for multiple decomposition of existing software products. Quoting Ossher and Tarr:

> "Anticipation causes ulcers : Deeply ingrained within software engineering is the notion of anticipating and designing for the *most likely* kinds of changes, towards the goal of limiting the impact of future evolution. [...] We believe in anticipating and planning for changes whenever possible. Anticipation is not, however, a panacea for evolution. It clearly is not possible to anticipate all major evolutionary directions. Further, even if it were possible, building in evolutionary flexibility always comes at a price: it increases development cost, increases software complexity, reduces performance, or often all of the above." [18].

The stand of Ossher and Tarr is very close to ours: Anticipation is definitively useful, but it is not a panacea. As noticed in the previous section, we believe that the problem becomes prominent in the field of ERP systems: The domain that they cover is very large and is constantly evolving, hence the software maker cannot *comprehend* it. Even if he could, finding a convenient and useful safe approximation (an abstraction) of all those local variations is likely to be difficult. We will show how we address the anticipation problem in Section 3.

1.5 Considering a simple approach

A simple approach to software customization is to let third-parties do customization in place, directly in the source code. Using this scheme, the software maker makes part of the source code available (typically the part of the source code that focus on domain functionalities), and let third-parties do modifications where they need it. This approach is indeed very simple, one may even say *naive*, but it is concrete and practical — therefore not to be dismissed immediately. To some extent, making customizations in-place is just an application of how software is built on a daily basis: Consider two people collaborating on a software project; programmer B makes changes in place to some previous code written by programmer A. Indeed, customization in this case is not directly distinguishable from software construction. We say that in this case the two activities of software evolution and software customization are symmetric. This approach is very close to the one offered by Microsoft Dynamics: Partners can modify part of the source code with a pre-defined granularity of change[2], and in the case of Microsoft AX, a system

[2]For example, the method is the granularity of change for class elements.

of layers defines a linear ordering of these changes, see [19] for details.

1.6 The upgrade problem

We now present the upgrade problem. We focus on code upgrade; data migration is another important problem that we do not discuss here. ERP systems are being customized by numerous independent partners. Eventually, the software product maker will release a new version of the product. Figure 1.6 on page 7 is an abstract and simplified representation of the upgrade problem. Nodes represent a variant of a software product. A directed edge from a node B towards a node A denotes that B is an evolution or a customization of A. Horizontal edges denote evolutions, and vertical edges denote customizations. A particular version of a software product P_y^x can be customized by a partner, leading to the software product P_{y+1}^x. This software product can be itself further customized by an other partner, leading to P_{y+2}^x, etc. On the other dimension, P_y^x will eventually evolve to a new version P_y^{x+1}, forcing the partner that produced the software product P_{y+1}^x to adapt its customizations in order to come up with P_{y+1}^{x+1}.

This idealized commuting diagram is convenient for illustration purposes but can also be misleading. Indeed, we would like to emphasize that the very role of customizations is to have an impact on the semantics of the program. Similarly most software evolution also intentionally changes the semantics of the program (modulo code-refactoring and pure performance improvements). Semantic changes are done simultaneously in both dimensions, so it is difficult to reason about the correctness of a particular variant of the software product. In particular, interactions between evolution changes and customization changes might actually be required to convey the intent of both the software maker and of the partner(s). Informally, to be able to reason about the correctness of particular variant of software product one would have to differentiate between “good interactions” and “bad interactions”. The absence of any explicit specification make this task particularly difficult.

Imagine that a partner wants to implement a feature called F as a customization. To do this, he identifies a particular code fragment C whose behavior needs to be modified. In the next version of the software product several problems can surface when the customization (made the previous version) need to be ported to the latest version of the base product. First, C might have been modified, for example a loop was rewritten to use recursion, or C was moved to an other location in the code base. One can also consider the extreme case: The code fragment C might have disappeared all together in the new version. Or maybe the feature F is now provided by default by the software product maker, in which case the customization should not be ported to the new version of the software products but should be simply discarded. Yet an other problem can surface: In the new version of the software

$$\begin{tikzcd}
P_1^1 \arrow[r] \arrow[d] & P_1^2 \arrow[r] \arrow[d] & P_1^3 \arrow[r, dotted] \arrow[d] & P_1^n \arrow[d] \\
P_2^1 \arrow[r] \arrow[d] & P_2^2 \arrow[r] \arrow[d] & P_2^3 \arrow[r, dotted] \arrow[d] & P_2^n \arrow[d] \\
P_3^1 \arrow[r] \arrow[d, dotted] & P_3^2 \arrow[r] \arrow[d, dotted] & P_3^3 \arrow[r, dotted] \arrow[d, dotted] & P_3^n \arrow[d, dotted] \\
P_m^1 \arrow[r] & P_m^2 \arrow[r] & P_m^3 \arrow[r, dotted] & P_m^n
\end{tikzcd}$$

Figure 2: The upgrade problem

product, C was un-touched, but in addition to C another code fragment now needs to be modified to support the partner's intent.

Road-map

Section 2 will give a summary of the empirical study, focusing on the most relevant results for the discussion. Section 3 will introduce the Eggther framework for customization of software products, and will carefully motivate the design choices based on the results from the previous sections. Section 4 will propose to use two well-known measures to quantify exactness and completeness in the context of our framework. Important implementation details are addressed in Section 5. Section 6 describes some future work. Some of the most frequent questions around this work are discussed in Section 7; the same section will briefly point to related work. Finally Section 8 will conclude.

2 Empirical grounds

Our work on software products is grounded on an initial empirical and qualitative study that focused on customizations and upgrades of Microsoft Dynamics ERP Systems [6, 7]. We shortly summarize here the most relevant results for the discussion.

Cost of the upgrade problem It is difficult to precisely measure the cost of the upgrade problem described in Section 1.6. Our empirical survey points to a range between 10 and 15% of the price of the original customization for a single upgrade. These numbers are derived from informal discussions with ERP practitioners and were not collected using a rigorous statistical approach. Upgrades have to be operated on each deployed project. Customers typically consider that upgrades are almost mandatory: They fear that they will not benefit from the latest bug fixes if they do not upgrade. They also avoid to jump a version because upgrading will be harder later on. Upgrades happen approximately every two years.

Partners are domain-experts When recruiting new staff members, partners tend to favor domain experts to highly skilled software programmers. Ideally, they try to form groups of two persons, who work together on the same customization project: One is the domain expert (for example an accountant), and the other one is a more technically minded person. Typically, even the "technically minded" staff members have little formal computing science education: They either learned programming through internal company training, or are autodidacts. Staff members share their experience with their co-workers through informal discussions and pair programming.

Partners rely extensively on code examples Staff members working for partners acquire their knowledge about the ERP system using code examples. They look at the existing application code (given by the software product maker), or to some code previously written by a colleague, and imitate the practices. When it is available, documentation is of course used from time to time, but we would like to stress that knowledge is mainly acquired by looking at code examples.

Customizations are varied in kind and granularity Customizations are various in their kind, and can target any part of the base source code. Some part of the code base are referred to as "hot points" since they tend to be customized more often than others. But customization can target any part of the available source code depending on the needs of the customers. Similarly, customizations have various granularity: Some are extremely fine-grained and target a small expression inside a specific method, others require rewriting of numerous methods across various classes.

Mainly incremental Customizations are described by the partners as being "mainly incremental": Partners typically avoid to remove functionalities, and prefer to simply hide unused elements at the graphical user interface level. It is commonly perceived as "adding functionality" or "adding features". Note that the terminology used by partners is informal: Their customizations do have side-effects and impact the semantics of the existing code base (obviously a customization that has no impact is of little value).

Exit points Customizations are done by inserting exit points to some hook methods. That is, they avoid to the extent possible to insert a lot of code in the existing code base and simply insert external calls to their newly defined methods. This offers a form of textual modularization, albeit not a perfect one since these method-calls are intrusive.

Time to market is more important than correctness Little systematic testing is done by the partners. Partners are aware of the consequence of having weak quality control practices around their customizations. According to our study, it seems that the root of this choice comes from the customers: Customers are not willing to pay for the extra-time required for testing. They prefer to do have quick fixes done directly on production site whenever a problem comes up. Of course the back-end database is typically backed up on regular basis to deal with worst-case scenarios.

Partners are unsatisfied by the feedback channels In the Dynamics business model partners can report their needs to the software product maker. According to our survey, partners are not satisfied with the feedback channels currently in place.

3 Eggther framework

This section introduces the Eggther framework for customization of software products. We show that our approach (a) Allows for non-anticipated customizations, and (b) Provides some resilience to upgrades. The work is grounded in the empirical work described in section 2, and builds on the .NET framework [9].

3.1 Overview of the approach

We summarize the approach, and we will come back in details on each step in the rest of this section.

First, the software product maker designs his software product using the dominant decomposition mechanism in object-oriented programming: Data decomposition. This decomposition is based on his knowledge of the vertical domains, and on his capacity to anticipate future needs for customizations, see Section 1.3.

Second, the software product maker gives to the partners part of the source code. The first difference with the scheme described in Section 1 is that partners only have read-access to the source code: We say that this is a glass-box approach to code reuse. If the decomposition of the software product fits their needs, the partner just use the traditional object-oriented extensibility mechanisms (sub-classing, method redefinition, etc.); if the decomposition is not convenient, the partners write code queries to denote the variability points, see Section 3.2.

The partner then writes customization code. The binding between the variability points and the customization code is described in Section 3.3. Customizations can be further customized by another partner. Upon upgrade queries are re-applied, and eventually modified. Customizations code is eventually modified. Optionally, the software maker collects the code queries (the code queries only, not the customization code).

3.2 Code queries

We now enter the core of the problem. We mainly deal here with the limitations posed by the crystal ball assumption and how we can approach these limitations in a way that allows us at the same time to deal with the upgrade problem.

From white-box to glass-box As noticed previously, a white-box approach is not satisfactory since this will make upgrades difficult. A pure black-box approach is not satisfactory neither since it requires anticipation. Therefore, we move from a white-box to a glass-box approach: Partners are given part of the source code, but with a read-only access. The first step for them is to denote the variability points, the locations in the code where customizations should happen.

From extensional to intensional definitions The programmer needs to define the set of code fragments that have to be customized. We distinguish two ways to define sets: B*y extension* and by *intension*. An extensional definition would have the form

$$CodeFragments = \{c_1, c_2, c_3, ...\}$$

whereas an intensional definition would have the form

$$CodeFragments = \{c \,|\, P(c)\}$$

where P defines a property of the source code fragment — this is also known as *set comprehension*. Using our framework, the variation points will be described, to the extent possible, by *intension* rather than by *extension*. This will allow the denotations to be more resilient to evolution of the base code because the variation points are not mentioned explicitly.

One way to express variation points by intension is to use a reflective language with a set of user-defined and pre-defined relations, that could include for example an implementation of the predicates

$$IsMemberPublic : \; Member \; \rightarrow \mathbb{B}$$

$$IsMemberName : Member \times String \to \mathbb{B}$$

Using set comprehensions, the partners could then express that all public members called “F” should considered as variation points:

$$VariationPoints = \{x \in Method \,|\, IsMemberPublic(x) \wedge MemberName(x, "F")\}$$

This is a perfectly valid approach and it also maps almost directly to a rule-based engine, such as Prolog. Unfortunately, it requires partners to think at a higher level of abstraction, at a meta-level. This conflicts with the fact that partners are not computing science specialists: They are domain experts, see Section 2. We want to give to partners simple and yet convenient programming primitives: Primitives that are close to how they approach their daily programming tasks.

A simple programming primitive: code-query by example As observed in our empirical study, programmers like to work with code examples. This is how they work when they face their daily programming tasks: This approach seems natural and appealing to them. Hence, we propose to use the concept of query-by-example to denote variability points. Query-by-example is a well-known concept in the field of database systems, we adapt it here to the domain of code query: What partners are effectively querying is the code base of part of the software product.

Query methods .NET attributes can be used to annotate class members [9, 8]. Class members marked with the `[Query]` attribute will denote code queries. In the case of methods, we will simply call them *query methods*. Several query methods can participate in the same query as we will demonstrate. For a given query, the framework will look for matching code fragments in the existing software product. The matching code fragments will be the variability points. The framework will insert method invocations to the customization code at the variability points (we will describe customizations in Section 3.3).

Formal arguments of query methods Variables available at the customization points will be bound to formal arguments of the query methods. Query methods formal arguments of type delegate [9] are used to denote an arbitrarily large well-formed code fragment whose static type is the return type of the delegate: For example, `Func<int>` denotes an arbitrarily large expression of static type int. A delegate with a `void` return type denotes arbitrarly large well-formed sequence of instructions.

An example In the following example, a partner wants to perform a customization in all the code locations where a transaction is performed. Using a code query example, the partner describes his concept of a transaction as a

BankAccount being debited from a given amount, some further action taking place, and a BankAccount being credited, within the same procedure. In the following listing, the formal argument `action` is of static type `Action` (a delegate with a `void` return type), hence any arbitrary large sequence of instructions would match the delegate call, line 5. (Note that all example compiles with a standard C# compiler.) The query is given a name, here "Transaction":

```
1 [Query("Transaction")]
2 void SimpleTransaction(double amount, BankAccount b1, BankAccount b2, Action action)
3 {
4   b1.Debit(amount);
5   action();
6   b2.Credit(amount);
7 }
```

For example, the query "Transaction" would match the following code fragment from line 2 to line 4:

```
1 static void F(double x, BankAccount a, BankAccount b, double tax) {
2    a.Debit(x);
3    x -= tax;
4    b.Credit(x);
5 }
```

Note that here the identifier `x` is not required to be bound to the same value for the debit and credit operations: In this case, the action applies a tax, and rebinds `x` to a new value. Note also that nothing prevents the identifiers `a` and `b` to be bound to the same object (aliasing). The action can refer to an empty action, hence the following code fragment would match our first code query:

```
  a.Debit(x);
  b.Credit(x);
```

and the following code would return four matches:

```
  a.Debit(x);
  a.Debit(x);
  b.Credit(x);
  b.Credit(x);
```

Disjunction of query methods Following on the same example, if the partner wants to cover the case where a Credit operation is done first, then a new query method must be added. Note that both query methods below are given the same query name, hence both of them participate in the definition of the query "Transaction". Informally, the code query can be interpreted as the disjunction of two cases: Code fragments of the form `DebitFirst` *or* code fragments of the form `CreditFirst`.

```
[Query("Transaction")]
void DebitFirst(double amount, BankAccount b1, BankAccount b2, Action action)
{
  b1.Debit(amount);
  action();
  b2.Credit(amount);
}

[Query("Transaction")]
```

```
void CreditFirst(double amount, BankAccount b1, BankAccount b2, Action action)
{
  b1.Crebit(amount);
  action();
  b2.Debit(amount);
}
```

Full methods matching versus code fragments matching Sometimes it is convenient to query for full methods, and not just code fragments. To this extent it seems natural to rely on the concept of a delegate. For example, the following code query will match any method that has formal arguments compatible with types double, BankAccount, and BankAccount, exactly in that order.

```
[Query("Transaction")]
Action<double, BankAccount, BanckAccount> transaction;
```

One can also instantiate this delegate to match any method that has the corresponding formal arguments (as described above) and additionally have the full method body matching the delegate body:

```
[Query("Transaction")]
Action<double, BankAccount, BanckAccount> transaction = { ... /* delegate body */ ... }
```

Summary of code-queries To summarize, we moved from a white-box to a glass-box approach ("see but don't touch"), by adding a level of indirection: Code queries. The query language is based on the concept of query-by-example. This simple programming primitive makes the approach accessible to partners (non-programmers experts), as they do not have to think at a meta-level. The code queries are a form of code quantification. This code quantification allows us to break procedural abstraction and hence to deal with fine-grain unanticipated customizations. Quantification allows us to textually localize the definition of the variability points outside of the base code of the software product.

3.3 Customization code

Once the variability points are defined using code queries, the actual customization code can be expressed using a regular .NET language, such as C#. Methods annotated with the attribute `[Customization]` are called *customization methods*. Classes that contain customization methods are called customization classes. The customization attribute takes as an argument the name of the code query that denotes a set of variability points that should be customized.

Example of customization code Continuing on the transaction example, now that the variability points are defined, the partner wants to log all transactions before they take place.

```
[Customization("Transaction")]
static public void LogTransaction(double amount, BankAccount b1, BankAccount b2) {
  Log("Transaction from account {0} to account {1}, amount {3}", b1.Number, b2.Number,amount);
}
```

Binding Formal arguments of customization methods are bound to available identifiers at the scope of the variability point. The framework builds a sequence of available identifiers within the scope of the variability point $\langle i_1, i_2, i_3, ...\rangle$ respectively of static types $\langle I_1, I_2, I_3, ...\rangle$. The signature of a customization method M defines a sequence of formal arguments $\langle f_1, f_2, f_3, ...\rangle$ respectively of types $\langle F_1, F_2, F_3, ...\rangle$. For each formal argument f_n the framework looks sequentially in the sequence of available identifiers for an identifier i_m, such that i_m was not already bound to a formal parameter of M, and such that $I_m <: F_n$, where $<:$ denotes the usual subtyping relation[3]. If the framework cannot find such an identifier the customization method will not be called. Note that only a prefix of the available identifiers at the scope of the variability point is necessary in the signature of the customization method. For example, if the customization would simply need to log the amount of the transaction, the following method signature would be sufficient:

```
[Customization("Transaction")]
static public void LogTransaction(double amount) {
  Log("Transaction amount " + amount);
}
```

On the other hand the following customization would not be called (otherwise it would be ambiguous which BankAccount we refer to).

```
[Customization("Transaction")]
static public void LogTransaction(BankAccount b) {
  Log("Transaction account " + b.Number);
}
```

The current object Sometimes, it is useful to have access to some members of the current object (the object where the customization is called). To support this, the partner can annotate a formal argument f_x of a customization method with the attribute `[Current]`, in which case the framework will bind the current object to f_x. If the variability point is in a static scope, f_x will be bound null. If the current object is not a subtype of Fx, the customization method will not be called. Note that using this attribute is optional feature, and can be safely omitted if it is not required. For example suppose that we want to log the number of all the TransactionManagers when a transaction takes place:

```
[Customization("Transaction")]
static public void LogTransaction(double amount, [Current] TransactionManager tm) {
  Log("Transaction amount {0} executed by transaction manager {1}", amount, tm.Number);
}
```

[3]Type compatibility is defined by the ECMA standard [9, 8]

Note also that we do not break encapsulation here, since the property Number of the class TransactionManager should have public access, or at least the member should be accessible from the customization class.

Before versus after customizations By default customizations are triggered just before the matched code fragments execute. If partners want a customization to be called just after the variation points, he can simply set to true the `After` property on the `[Customization]` attribute. For example the following will log the balance of the debited account after the transaction has taken place:

```
[Customization("Transaction", After = true)]
static public void LogDebitedAccount(double amount, BankAccount b) {
   Log("After transaction of {0}, balance of the debited account is {1}", amount, b.Balance);
}
```

Side effects in code customizations

So far our customizations mainly added behavior. This corresponds to a great part of customizations tasks performed by partners, see Section 2. Given the techniques that we already covered, a partner can already introduce side effects in customization methods, for example by writing `b.Debit(1)` in a customization method.

Nonetheless, it is useful to be able to modify the binding of identifiers within the scope of the variability points. To do this, partners can annotate the formal arguments of customization methods with the standard `ref` modifier. The value of a reference parameter is the same as the argument in the method member invocation [9, 8] (they represent the same storage location). The framework will take care to bind the variables of the matched code fragments by reference. In the following customization method, a partner wants to convert the transaction amount from Euros to Danish Kroners:

```
[Customization("Transaction")]
static public void ConvertAmountToDKK(ref double amount) {
    amount *= EuroToDKK ;
}
```

The effect of this customization is that the `amount` identifier will be rebound to an new value. As a further example, the following (hazardous) customization swaps the debit and the credit accounts before the transaction takes place!

```
[Customization("Transaction")]
static public void SwapAccounts(ref double amount, ref BankAccount a, ref BankAccount b) {
   var tmp = a;
   b = a;
   a = tmp;
}
```

Short notation for customizations So far code queries and their corresponding code customizations were textually separated. Sometimes it is useful to express both concerns using a more compact notation. To this extent, partners can annotate class members using both attributes `[Query]` and `[Customization]`. This notation is considered by the framework as syntactic sugar for two class members: The formal arguments being the code query and the body being the code customization. For example, the following class member `M` is an example of the short notation:

```
[Query]
[Customization]
static void M(T t) {
    /* customization code */
}
```

Note that in this case, the query name does not need to be mentioned. The code above is considered as syntactic sugar for the following two members:

```
[Query(X)]
Action<T> MQuery;

[Customization(X)]
static MCustomization(T t) {
    /* customization code */
}
```

The name X is an automatically generated and unique string identifier. As a concrete example, the following code will log all method calls to procedures that have a double as their first (and possibly unique) formal argument:

```
[Query]
[Customization]
static public Action(double d) {
     Log(d);
}
```

3.4 Interrupting the flow of control

Note that the approach that we propose is mainly incremental: Functionality is added to some specific places in the code base. This fits closely with the result of the empirical study on customization practices, see Section 2. Nonetheless, it is possible to interrupt the flow of execution at the variability points by throwing an exception in the customization code. We take advantage of the fact that exceptions are unchecked in .NET [9]. Throwing an exception to interrupt the flow of control is, arguably, a reasonable thing to do: We want to warn callers (at run-time), up in the call-stack, that the expected execution of part of the base code did not take place. Continuing on the transaction example, a partner wants to forbid transactions of more than 1.000.000 Euros:

```
[Customization("Transaction")]
static void NoLargeTransaction(double amount)
{
  if(amount > 1000000) throw new LargeTransactionException();
}
```

Another customization, now using two formal arguments, checks whether the debit account is large enough for the transaction:

```
[Customization("Transaction")]
static void CheckAccountBalance(double amount, BankAccount account)
{
  if(amount > account.Balance) throw new InsufficientFundsException();
}
```

Note once again that the partner only has to declare a prefix of the available identifiers in the scope of the variability points.

3.5 Unit testing

Customization methods are directly amenable to the usual unit testing procedures. For example, a unit test for the `CheckAccountBalance` customization would look like:

```
[Test]
[ExpectedException(typeof(InsufficientFundsException))]
public void TransferWithInsufficientFunds()
{
  BankAccount b1 = new Account(1000);
  BankAccount b2 = new Account(1000);
  TransactionManager.Instance.Transaction(b1,b2,2000);
}
```

3.6 Stateful customizations

As mentioned in Section 1, data and state are an important part of modern ERP systems, therefore it is important to support stateful customizations: The framework must allow the partners to preserve some state across several invocations of the same customization method. In our framework, customization classes can simply declare some class or instance variables that will preserve state across invocations of customization methods. The following example counts the daily number of debit operations (made as part of a transaction) for all bank accounts, and throws an exception if a threshold is crossed:

```
// Daily initialize NumberOfDebitOperations to 0 for all existing accounts
Dictionary<BankAccount, int> NumberOfDebitOperations { get { ... } }

[Customization("Transaction")]
void CheckNumberDebitOperations(double amount, BankAccount account)
{
  var num = NumberOfDebitOperations[account];
  if(num > 100) throw new TransactionThresholdException();
  NumberOfDebitOperations[account] = num + 1;
}
```

3.7 Extension of customizations

Customizations done by partners can be extended or modified by another partner using the usual redefinition mechanisms provided by inheritance:

As shown in the previous example customization methods do not have to be static. Alternatively, code changes can also be done by modifying the code queries or, by quantifying on the customization code written by the other partner. Note that since customizations are more textually localized it is easier for the partner to modify some existing customizations.

4 Exactness and completeness of code queries

Upon upgrade, existing code queries might not denote *exactly* code fragments that fit the partner's intentions since the customizations were developed for the previous version of software product. Similarly, code queries might not refer *completely* to the code fragments that should be customized to fit the partner's intention. A measure of exactness and completeness of code queries would be helpful to characterize these issues. To this extent, we introduce precision and recall.

4.1 Precision and recall

Precision and *recall* are two measures widely throughout science used, and especially in the field of information retrieval, to evaluate the quality of results, focusing respectively on exactness and completeness [2]. We introduce precision and recall as they provide a convenient and well-defined terminology for the rest of the discussion. The two measures are traditionally defined in terms of a set of retrieved documents, and a set of relevant documents. Precision is the percent of retrieved documents that are relevant to the search (exactness):

$$Precision = \frac{|\{Relevant\, documents\} \cap \{Retrieved\, documents\}|}{|\{Retrieved\, documents\}|}$$

In turn, recall is the fraction of documents relevant to the query that are successfully retrieved (completeness):

$$Recall = \frac{|\{Relevant\, documents\} \cap \{Retrieved\, documents\}|}{|\{Relevant\, documents\}|}$$

One can immediately observe that a recall of 1 can be easily obtained by returning all documents in response to any query. Dually, one can very easily obtain a high precision by returning no documents to any query. Hence, it is useful to use those two measures together.

4.2 Precision, recall and code queries

When a partner writes a code query, he wants to denote the variability points for customization purposes. We will say that precision is high when the code query returns mostly variability points that are necessary for the customization. Dually, recall will be high when most required variability points to implement the customization are returned by the code query.

When precision is lower than 1 some customizations will happen at places where they should not happen. Similarly when recall is lower that 1, then some locations in the code base where customization should happen will not take place.

4.3 Exactness and completeness problems upon upgrade

Upon upgrade, a partner takes the new version of the base software product and reapplies the code queries. We simplify the discussion by considering that the customization consists of only one code query. We can immediately identify two problematic cases:

(1) The query result now points to some code fragments that *should not* be customized in the new version,

(2) The query fails to match some code fragments that *should* be customized in the new version of the base code.

From a practical point of view, the first problem is less problematic: The partner is presented with the result of the code query, and can inspect whether the new customizations points fit his intentions. If he identifies a problem, he can decide to rewrite the customization query to fit his needs. The second problem seems more difficult to tackle. We identify two sub-cases to problem (2):

(i) Some new code that *should* be customized is not part of the query result,

(ii) Some old code that *should not* be customized in the previous version *should* now be customized in the new version.

Helping with (i) can be done by simply providing a diff of the new version versus the old version to partners, and requiring them to study very carefully the changes. This is highly impractical, of course. Alas, (ii) seems even more difficult to deal with; it is not clear at this point how to approach this last problem in a scalable way in the absence of any explicit specification.

4.4 Giving control back to the partners

Ideally, one would like to have both high precision and high recall. Nonetheless, one can sense that there is a tension between the two: Design decisions that favor precision will hamper recall and vice versa. Since these design decisions directly impact the upgrade process, it would be good if partners could regain control of them. In other words, let them decide whether exactness or completeness is more important to them. Our framework allows for this:

- Partners can abstract their code queries by making use of delegate calls to denote typed expressions. Making extensive use of delegate calls in code queries will increase recall but decrease precision.
- Dually, partners can add more context code in their queries (make their code query more lengthy than strictly necessary) and will thereby increase precision but decrease recall.
- Similarly, partners can increase or decrease the number of formal arguments in their code query, which will respectively increase precision (but decrease recall), and increase recall (but decrease precision).

Section 6, which describe some further work, details yet an other way by which partners could favor precision or recall.

5 Implementation aspects

We quickly describe the current implementation of the Eggther framework since the design choices have some important implication on the application of the framework. A more precise account of the implementation will be presented in an other paper.

5.1 General design choices

An obvious design choice for our framework would have been to reuse an existing parser for .NET language (or to generate one), and to do the code queries on the abstract syntax tree instantiated by this parser. Unfortunately this has major drawbacks:

- For each language that we would like to support we would have to implement a new variant of the framework.
- Every time that one of the high-level language evolve we would have to evolve our framework.
- Conceiving a high-performance parser for a complex high-level language is difficult.
- We would loose integration with current development environments.

Therefore, we decided against this approach, and instead settled to work at the intermediate code level.

$$\begin{tikzcd} Software\,product_{original} \arrow[r, "illusion"] \arrow[d, "\text{standard compiler}"'] & Software\,product_{instrumented} \arrow[d] \\ IL_{original} \arrow[r, "instrumentation"'] & IL_{instrumented} \end{tikzcd}$$

Figure 3: Customization scheme

Implementation details We give an outline of our current implementation:

- First, the base code is compiled to .NET intermediate code (IL) [8, 9] using one of the standard compilers for the corresponding high-level language (the standard C# compiler, the standard VB.NET compiler etc.), and the same is done with code query and the customization code.
- At design time, the framework will work at the IL level, and for each code query will look for matching IL code in the compiled base code. When a match is found the framework will instrument the bytecode by injecting a method call to a proxy method, binding the formal arguments of the proxy method to the available identifiers in the scope of the method call.
- At run-time, the framework will construct a list of the customization methods and instantiate a delegate for each them. A singleton [11] is instantiated of each customization classes. When a proxy method is called, it will look for the customization methods that subscribed to this particular customization points (the query name), and will invoke the corresponding delegates. Binding of formal arguments is done as described in Section 3.

Note that even if matching and instrumentation is done at the IL level, partners get the illusion that is performed at the same level at which they write high-level code, see Figure 5.1 on page 21.

5.2 Advantages

We summarize some of the advantages of this approach:

- Our framework re-uses the standard high-level compilers, and we can rely on Microsoft and others to maintain them and evolve them as new version of the languages surface.

- The standard compilers have been well-tuned for many years and are highly performant.
- We get support out of the box not just for one high-level language for a large number of them (C#, VB.NET, Eiffel, etc.): The only requirement is that the base software product and the code queries are written in the same language and are compiled using the same compiler.
- Partners enjoy the full development environment that is already available to them and that they know well (the interactive development environment of Visual Studio that includes a type checker, syntax highlighting, refactoring tools, etc.). We would like to stress that our use of the standards type checkers allows us to make sure at design time that the query and customization methods are well-formed.

5.3 User interface

The concrete user interface that will be offered to partner is an add-in for Visual-Studio that will allow them at design time to execute the code queries and visualize the variability points in the base software product.

6 Further work

We shortly introduce some further work that are potential extensions to the framework we introduced in Section 3.

6.1 Non-Boolean matching

So far, code matching was defined as a simple predicate, taking as arguments two code fragments, and returning true or false depending on whether the two code fragments match each other. We can generalize this approach by defining a measuring function M that will give the distance between any two code fragments:

$$M : CodeFragment \times CodeFragment \to \mathbb{R}_0^+$$

We will call this distance the *code distance*. We are considering to use two well-studied algorithms as the foundation for code distance: The first is the Levenshtein distance (also called edit distance), and second, the tree-edit distance [4].

6.2 Partial ordering of customizations

It would be useful to be able to specify an order in which customization methods should be executed for a given variability point. A partner should be able to define a simple partial order between customizations (that includes his own customization as well as customizations from other partners), and the framework should simply compute a linearization compatible with this partial order. This seems, a priori, quite straight-forward to implement. We believe that the challenge is more on the language design side: How to allow partners to express this order relation in a convenient way, and make it at the same time seamlessly integrated with the programming primitives that we introduced. Furthermore, it is not clear what the framework should do if it is not possible to compute a linearization given the constraints given by the partners: Should it throw an exception? And if so, where and when?

6.3 Toward behavioral customizations

The customizations that we described were based on matching of the code structure. While it is a practical and convenient solution to many customizations scenarios, sometimes the customizations are not driven by specific code patterns but by more behavioral aspects.

Example Continuing on the previous example, consider the following customization scenario: Suppose that when the balance of a bank account exceeds a threshold, the local bank branch should get an alert (for example to send to the client some commercial offers for the latest financial products). This particular scenario is more abstract that the previous ones, since here the customization is not geared toward specializing the software product at some specific places in the code text, but rather to take some action under certain conditions, irrespective of the location in the code that made the bank account cross the threshold. Moving from this informal requirement towards a more precise specification, we define predicate $BalanceTreshold : BankAccount \times \mathbb{R} \rightarrow \mathbb{B}$, which evaluate to true if given a bank account and a threshold, the balance of the bank account is greater than the threshold. Whenever the predicate first evaluates to true, the customization should be triggered, namely sending an alert to the local bank.

Possible approach Consistent with our goal of reusing existing .NET technology as much as possible we envisage to use to Spec# to our benefit. Spec# is an extension of C# that adds to the language, among other things, the concept of invariants and pre and post-conditions. Spec# is based on the foundations laid down by the work on axiomatic semantics, pioneered by Floyd, Hoare and others. The definition given by Hoare [14] is based on the concept of a triple $\{A\}B\{C\}$ that defines partial correctness: Whenever A is true and B executes and terminates, then C will be true (where A and B are predicates on the state and B is a command). Similarly, but from a more concrete point of view, a method can be equipped with pre and post-conditions, where the precondition defines what has to be true at the beginning of method execution (to be satisfied by the client, i.e. a benefit to the supplier), and the post-condition what has to be satisfied by the supplier when the method terminates (to be satisfied by the supplier, i.e. a benefit to the client).

In a previous work [22], we redefined the notion of pre-condition in the concurrent case to move closer to the concept of conditional critical regions (first proposed by Hoare then championed by Brinch Hansen [13]). Similarly, we envisage to redefine the semantics of the pre-condition in the case of a customization: Whenever a customization method is equipped with a pre-condition, this precondition will define a condition for triggering the customization. More concretely, and using the concrete example introduced above, the condition is the predicate, expressed as pre-condition to the customization:

```
[Customization]
void AlertThresholdCrossed([Current] BankAccount b)
requires b.Balance > 10000;
{
  SendAlert(b.LocalBank, b);
}
```

The semantics of this customization is to be interpreted as follows[4]: For all bank accounts b that are instantiated in the runtime such that $AlertThresholdCrossed$ was not already executed with b as an actual argument, whenever the balance of b is more than 10.000 Euros, then execute $AlertThresholdCrossed$.

7 Discussion and Related work

This section discusses informally some of the questions related to the approach that we presented, and points to some related work.

What if one is completely satisfied with the crystal ball assumption? If anticipation can be done accurately in a way that satisfies the partners (that is, if customization points are well defined and the granularity of future customizations is well-understood), then usual customization techniques can be used: For example dynamic binding, together with a flexible software product design that makes use of a subset of the various extensibility-related design patterns : Factory methods, visitor pattern, adapter etc. Interface specifications can then be made more precise using for example design-by-contract (see Spec# [3], or JML [15], etc.). On the language side, the issue of co-variance versus contra-variance will then surface, and one will favor one option or the other, making a choice between openness and type safety [1, 5]. All this is now folklore in the software engineering community: It has been well-studied and well-documented. Of course more research in this field is certainly useful (see for example the work on ownership type systems [17]),

[4]Notice that, in this case, the semantics of the pre-condition is close to one of a guard.

but we believe that this does not address a problem which is *characteristic* of ERP systems. Once again, we believe that the main characteristic of ERP systems is the difficulty to anticipate the future needs for customization accurately.

What about Versioning Systems? Versioning systems bear some resemblance with our work but have a more textual approach to code evolution. Versioning systems rely on a "diff" program to show the difference between two files [16]. One can consider diff as a more general approach to what we have described: Using diff any text files can be compared, whether it is source-code or not. By targeting only .NET languages, we can implement some useful specific functionalities: For example, our use of delegates to denote typed expressions in our code queries, or simply ignoring irrelevant differences between code queries and the base code (such as lines indentation). An alternative approach to our work could have been to conceive a special version of an existing versioning system and specialize it to deal only with C# code. We decided for different approach which makes extensive use of the existing .NET infrastructure (re-use of the existing compilers, etc.).

Is there some connection with Software Product Lines? A part of the software engineering community now focus on a line of research called *software product lines* (SPL). The goals behind SPLs bears some similarities with our work: Both allow for the customization of software systems by third-parties. Nonetheless, there is one fundamental difference: SPLs have a close-world assumptions. That is, the SPL community usually assumes that the set of possible customizations is well-known in advance, by a central agent such as chief architect [19]. ERP systems cannot rely on this assumption, see Section 1.3. From this perspective ERP systems are not SPLs.

Is this aspect-oriented programming? According to Filman and Friedman aspect-oriented programming (AOP) is quantification and obliviousness [10]. The code-query by example of our framework provides quantification: The result of the code queries are the joint-points in AOP parlance. Obliviousness is achieved when the programmers should not be required to insert join-points markers into they source code. Our approach provides obliviousness since no special markers are introduced. According to this definition, our framework is AOP, and code query by example is a point-cut language.

What seems to be, at this point, the pros and cons? It is still early to give a precise evaluation of our approach, since we are designing and developing the framework and experimenting with the expressiveness of the code query primitives. Nonetheless, we would like to emphasize the following pros and cons that seem, *a priori*, to characterize the framework. First, ease-of-use is an important aspect of this work: The programming primitives are simple; they do not force domain experts to think at a meta-level and do not force them to learn a complete new language; partners can think at the level of abstraction they are used to, using for a great part the same high-level language that they already know – They do not *say* what they want to customize but they *show* it. Second, adoption of our approach is completely incremental (no change is required to the existing code base). This is important since many software products such as Dynamics have a very large existing code base. Third, partners have control over the discussed precision-versus-recall trade-off by adding more or less context code to their query, and by abstracting more or less the code queries using delegates. An other positive aspect of our approach is that we are re-using extensively existing .NET technologies: For example, we rely on the standard .NET compiler for C# or VB.NET, which means that support for the next version of these languages is much simplified. One issue is that we have to support the evolution of the intermediate language (IL), but the recent history has shown that IL evolves at a slower pace than high-level languages. The approach *seems* to allow for fast code queries, but this has to be confirmed by experimentation. Also, it *seems* that the framework supports almost out of the box many high-level languages that were written for the .NET platform, since most of the work is done at IL level. Customizations can be further customized but it is not clear at this point how convenient this is. The main con seems to be that query is mainly done by matching code patterns, and it is not clear how convenient this is to denote complex variability points. Nonetheless, we hope that this potential issue could be addressed by what we called behavioral customizations, see Section 6.

8 Conclusions

We described the upgrade problem and we emphasized that anticipation, one of the pillars of modern software engineering technology, is not completely adequate for modern ERP systems. We presented the Eggther customization framework that mitigates the upgrade problem while at the same time allowing for un-anticipated and fine-grained customizations. We introduced precision and recall as two useful measures for exactness and completeness. We gave an outline of the implementation and presented the pros ans cons of our main design decisions. After a discussion we briefly introduced some future work, and finally we mentioned some related research.

Acknowledgments

We would like to thanks Antonio Cisternino with whom we developed many of the ideas during a stay at University of Pisa. This stay was made possible thanks to the kind invitation of Egon Börger. Thanks to Peter Sestoft, Yvonne Dittrich, and Morten Rhiger for their help and support. This work takes place under the umbrella of the Evolvable Software Project, and is sponsored by NABIIT under the Danish Strategic Research Council, Microsoft Development Center Copenhagen, DHI Water and Environment, and the IT University of Copenhagen.

References

[1] ABADI, M., AND CARDELLI, L. *A Theory of Objects*. Springer-Verlag, 1996. 24

[2] BAEZA-YATES, R., AND RIBEIRO-NETO, B. *Modern Information Retrieval*. Addison-Wesley, 1999. 18

[3] BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. The Spec# programming system: an overview, 2005. 24

[4] BILLE, P. A survey on tree edit distance and related problems. *Theoretical Computer Science 337*, 1-3 (2005), 217–239. 22

[5] CASTAGNA, G. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems 17*, 3 (May 1995), 431–447. 24

[6] DITTRICH, Y., AND VAUCOULEUR, S. Customizing and upgrading ERP systems: a reality check. Tech. Rep. TR2008-105, IT University of Copenhagen, 2008. 7

[7] DITTRICH, Y., AND VAUCOULEUR, S. Practices around customization of standard systems. 7

[8] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*, second ed. European Association for Standardizing Information and Communication Systems, 2002. 11, 14, 15, 21

[9] ECMA. *ECMA-334: C# Language Specification*. European Association for Standardizing Information and Communication Systems, 2005. 9, 11, 14, 15, 16, 21

[10] FILMAN, R. E., AND FRIEDMAN, D. P. *Aspect-Oriented Programming Is Quantification and Obliviousness*. Addison-Wesley, Boston, 2005, pp. 21–35. 25

[11] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 21

[12] GREEF, A., ET AL. *Inside Microsoft Dynamics AX 4.0.* Microsoft Press, 2006. 2

[13] HANSEN, P. B. Structured multiprogramming. *Commun. ACM 15*, 7 (1972), 574–578. 24

[14] HOARE. An axiomatic basis for computer programming. *CACM: Communications of the ACM 26* (1983). 23

[15] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D. R., MÜLLER, P., KINIRY, J., CHALIN, P., AND ZIMMERMAN, D. M. *JML Reference Manual.* May 2008. 24

[16] MACKENZIE, D., EGGERT, P., AND STALLMAN, R. *Comparing and Merging Files With Gnu Diff and Patch.* 2002. 25

[17] MÜLLER, P., POETZSCH-HEFFTER, A., AND LEAVENS, G. T. Modular invariants for layered object structures, 2006. 24

[18] OSSHER, H., AND TARR, P. Multi-dimensional separation of concerns and the hyperspace approach. In *Symposium on Software Architectures and Component Technology: The State of the Art in Software Development* (2000). 5

[19] SESTOFT, P., AND VAUCOULEUR, S. Evolvable software products. In *Springer LNCS volume Advances in Software Technology* (2008). 2, 3, 4, 5, 6, 25

[20] SHANKS, G., SEDDON, P. B., AND WILLCOCKS, L. P. *Second-wave Enterprise Resource Planning Systems.* Cambrige Press, 2003. 2

[21] STUDEBAKER, D. *Programming Microsoft Dynamics NAV.* Packt Publishing, 2007. 2

[22] VAUCOULEUR, S., AND EUGSTER, P. Atomic features. In *OOPSLA Workshop on Synchronization and concurrency in object-oriented languages (SCOOL)* (2005). 24