Model-Centric, Context-Aware Software Adaptation *

Oscar Nierstrasz, Marcus Denker, Lukas Renggli

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch

Abstract. Software must be constantly adapted to changing requirements. The time scale, abstraction level and granularity of adaptations may vary from short-term, fine-grained adaptation to long-term, coarse-grained evolution. Fine-grained, dynamic and context-dependent adaptations can be particularly difficult to realize in long-lived, large-scale software systems. We argue that, in order to effectively and efficiently deploy such changes, adaptive applications must be built on an infrastructure that is not just model-driven, but is both *model-centric* and *context-aware*. Specifically, this means that high-level, causally-connected models of the application *and* the software infrastructure itself should be available at run-time, and that changes may need to be scoped to the run-time execution context.
We first review the dimensions of software adaptation and evolution, and then we show how model-centric design can address the adaptation needs of a variety of applications that span these dimensions. We demonstrate through concrete examples how model-centric and context-aware designs work at the level of application interface, programming language and runtime. We then propose a research agenda for a model-centric development environment that supports dynamic software adaptation and evolution.

1 Introduction

It is well-known that real software systems must change to maintain their value [26]. It is therefore curious to observe that the technology we use to develop software systems tends to hinder and inhibit change rather than to enable and support it [31]. Statically typed languages, for example, are based on the assumption that first-class values have fixed types that will not change, especially at run-time. Few mechanisms are available to developers to deal with the fact that interfaces do change over time, and real software systems may need to cope with different versions of the same libraries, possibly depending on the run-time context. Design patterns offer further evidence of ungainly workarounds that developers need to regain flexibility at run-time, for example to change the apparent behaviour of objects as a consequence of a change in state [17].

* B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 128–145, 2009.

Long-lived, software intensive systems [50] cannot always be modified in a static way. Furthermore, although certain kinds of anticipated adaptations can be built in by design as run-time configuration parameters, there are many kinds of dynamic adaptation that cannot be anticipated so easily. One canonical example of such an adaptation is run-time instrumentation: certain kinds of anomalies only manifest themselves with deployed software systems. As it is not possible to anticipate for all cases what and where to trace to observe the problematic behaviour, it may be necessary to dynamically adapt the running system. Other examples exist (such as adding new features to an always-running system), but the key characteristics remain the same — the software may need to be adapted dynamically, in a fine-grained way, while taking care not to disturb existing behaviour.

There are many important dimensions of software change. Let us just consider three of these that pose challenges for software development:

Timescale — Software is changed not only at the coarse scale of versions and releases, but also at a medium scale (*e.g.*, start-up configuration) and at a fine scale (run-time adaptation and instrumentation). Particularly at the dynamic end, little support is available to developers aside from certain design patterns and relatively low-level reflective mechanisms.

Granularity — Here too we see that software is changed not only at the coarse granularity of subsystems and packages, or the medium granularity of classes and methods, but also at a finer granularity within methods and procedures. Fined-grained, run-time adaptation of software must typically be anticipated by design, and necessitates the use of boilerplate code (*e.g.*, case-based reasoning over anticipated scenarios) or design patterns (*e.g.*, State or Strategy patterns). Unanticipated run-time adaptation will typically entail low-level techniques such as bytecode transformation.

Scope — Changes may be globally visible, they may be localized to individual users, or they may depend on an even finer context. The same software entities may need to behave differently as the run-time context changes. Mobile applications, for example, may need to switch to a fall-back behaviour as services become unavailable. Run-time instrumentation of software entities, as another example, may need to be dynamically adapted if the same entities are used by the instrumentation layer itself (*i.e.*, to avoid endless instrumentation loops) [14].

Although model-driven and round-trip engineering techniques have proved to be effective in maintaining the connection between high-level and low-level views of software systems, they do not especially address the problem of dynamic adaptation. We argue that it is necessary to go a step further from model-driven towards *model-centric* software, in which high-level, causally connected views of software and their application domain are available at run-time. In this paper we show several examples of run-time adaptation at the level of source code, so the "high-level models" appropriate to these applications take the form of ASTs that reflect the structure of software to be adapted.

Furthermore, such systems must be *context-aware* in order to control the scope of adaptations and changes. In our examples we show how context can play an important role in software adaptation to control the scope of change. We argue that current programming technology offers only very weak support for developing context-aware applications, and that new research is urgently needed into novel *context-oriented programming* mechanisms [21].

In this paper we make our case for model-centric, context-aware software adaptation by presenting two examples of platforms that adopt this approach. We show how the presence of sufficiently high-level models at run-time can enable very dynamic forms of context-dependent software adaptation.

In Section 2 we present *Reflectivity*, a relatively mature platform for dynamic, model-centric software adaptation. We have used Reflectivity extensively in various projects to support different forms of adaptation, such as run-time instrumentation, dynamic aspects, and software transactional memory. Next, in Section 3, we present ongoing work on *Diesel*, a lightweight language workbench which can be used to adapt the programming environment to support the expression of high-level application concepts by introducing numerous, lightweight domain-specific languages. We discuss further applications of these ideas and our vision for a research agenda in Section 4 and provide an overview of related work in Section 5. We conclude in Section 6 with some remarks on future work.

2 Reflectivity — a platform for model-centric software adaptation

In this section we present Reflectivity, a platform that supports dynamic adaptation of software by means of causally connected, high-level models of the source code [9]. The purpose of this section is (i) to motivate the need for dynamic software adaptation for various applications such as runtime instrumentation, dynamic aspects, and software transactional memory, (ii) to motivate the need for better mechanisms to support context-dependent adaptation, and (iii) to demonstrate that sufficiently high-level models available at run-time (in this case ASTs causally connected to bytecode) facilitate run-time adaptation.

Reflectivity is built on top of Smalltalk, since it already provides extensive support for run-time reflection, albeit at a relatively low-level of abstraction [9]. Furthermore, Smalltalk provides full access to the implementation of its infrastructure, making it ideal for extensive experimentation. Any other language that supports run-time structural and behavioural reflection and access to the infrastructure would also be suitable.

2.1 A model for dynamic software adaptation

The particular challenge we are focusing on is support for dynamic, fine-grained and possibly context-dependent software adaption. Let us consider the canonical example of run-time instrumentation:

- We may need to install the instrumentation code *dynamically* in the running system because the phenomena we wish to study only occur in the deployed system (say, a web service).
- The adaptation is *fine-grained* because we wish to monitor only part of a given method (say, conditional access to an authorization service).
- The adaptation is *context-dependent* because we are only interested in monitoring calls made from a specific application, not others.

Other plausible scenarios, such as adding features to a running system, would serve as well for establishing our requirements.

In order to dynamically adapt software, we need a *model* to reason about it. In our run-time instrumentation scenario the following properties are important:

Abstraction Level: This model should be high-level, reflecting the language concepts we wish to instrument, rather than, say, the generated bytecode.

Completeness: The model should represent the complete software, from coarse-grained structures like classes, methods down to sub-method structures such as variable accesses and method calls.

Although these properties may seem obvious, in most cases the representations used for software adaptation today do not satisfy them. The representation used is often plain (source) text. Modern development environments do better: here the code is represented with dedicated data-structures that better support code presentation (*e.g.* pretty printing) or code change (*e.g.* refactoring). But these data structures are those of the development environment, not of the language itself. They are not available to support run-time adaptation.

Runtime representations are often tailored solely towards execution, such as bytecode representations for Java or Smalltalk. Representations based on bytecode are low-level, and therefore suffer from a semantic mismatch with the core language concepts.

The reflective representation of the structure of software available in many modern object-oriented languages provides a high-level model for packages, classes and methods, but it lacks any representation of sub-method structure.

As we are especially interested in adaptation *at runtime* and *by the system itself*, we conclude that the model needs to have the following properties:

Self representation: The model of the software needs to be available from within the running system itself.

Causal connection: When we change this model (either from the outside or from within the system), the behavior of the program needs to change. Conversely, when the system changes, the representation needs to change, too. The program needs to stay in sync with the model at all times.

Meta-annotations: We need to be able to extend the representation to use it in many contexts and annotate it with meta-data. For example, different tools that deal with the structure of the system need slightly different information. Annotations allow the programmer to associate meta-data with any node, making the existing AST-based representation extensible.

Models that have the properties of *self representation* and *causal connection* are called *reflective*. There is a long history of reflection in programming languages in general and in object-oriented languages in particular [45]. *Sub-method reflection* [10,9] provides a model of the software that exhibits all the properties discussed above. Software is represented down to the statement level by a causally connected, annotated *Abstract Syntax Tree* (AST). Changes to the AST will be (lazily) propagated to the generated bytecode using runtime just-in-time compilation. Semantics are given to annotations by an open compiler infrastructure: annotations are interpreted by dedicated compiler plug-ins.

We will now illustrate this approach by a series of examples.

2.2 Run-time instrumentation

With a causally connected model of the whole software, we can provide a framework to support *instrumentation at runtime*. The model chosen is that of *partial behavioral reflection* [47,12,40].

The central notion is the *Link*. The link is set as an annotation to one or more nodes of our AST. The link points to a meta-object and can be parameterized to indicate which information is passed to the meta-object. In addition, we can set a condition to specify when a link is active and to specify if the meta-object is called before, after or instead of the original instruction. Figure 1 shows the interaction of the AST, the link and the meta-object.

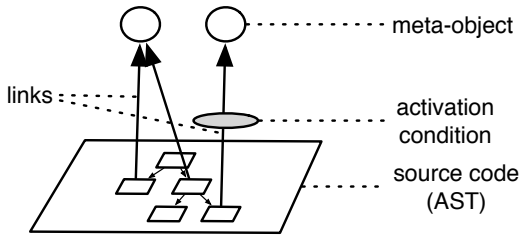

Fig. 1. The link-meta-object model

Links are specified as annotations on the AST. A compiler plugin transforms the AST before execution to take the links into account. A link thus results in code to be inserted in the program at the nodes where it is installed.

This model provides some interesting characteristics: it is completely dynamic due to just-in-time compilation at runtime. We can create links at runtime, configure them and install them in the system. Links can even be installed by other links, or they can remove or install themselves.

A side-effect of using the higher-level representation provided by the AST is improved performance. It is actually easier to generate more efficient code using the AST [10]. Furthermore, we only have to generate bytecode for those

parts of the system that take part in the execution. The actual set of classes used by an application is generally smaller than the overall code base by an order of magnitude. As a result, dynamic code generation provides an additional performance benefit [9].

2.3 Localization: annotating structure

We have seen both a structural model of our system, and a framework for dynamic behavioral reflection based on annotating the structural model with *links*. Now we will see how to manipulate this structure using behavioural reflection.

To make this possible, we need to be able to reference the structural model from the behavioral world. The simplest way to do this is to allow the nodes of the structural model to be meta-objects, as shown in Figure 2.

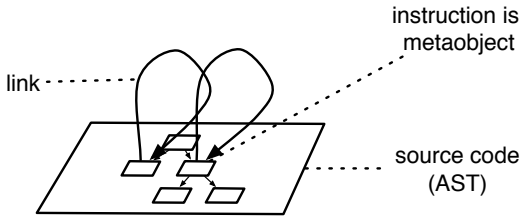

Fig. 2. Bridging structural and behavioral model

This allows a node to be annotated before it is executed. This way one can easily realize tracing or feature analysis [13]. For example, a simple code-coverage tool can be realized by installing a link on each AST node of interest. We just provide a method `markExecuted` to mark AST nodes as having been executed. The links are activated when the AST nodes are executed, and simply invoke this method to record the fact.

```
link := GPLink new metaObject: #node;
            selector: #markExecuted.
```

Listing 1.1. Code-coverage analyzer realized with Reflectivity.

When we install the link on the node representing methods, we obtain method level coverage. But with our sub-method model, we can go a level deeper and even install the link on all assignments.

To improve performance, it is even possible to remove the tagging-link at runtime just after tagging the node. In this way, the method would, at the next execution, be recompiled to only call `markExecuted` on those nodes that have not yet been executed before. We can also take advantage of activation conditions, for example, to only tag nodes that are executed in the context of a unit-test.

The possibility of both installing and removing links at runtime allows for just-in-time annotation: links are installed on-demand on all methods that are to be executed next. This way an annotation can spread itself through the system, driven by the flow of execution itself. Examples like these make Reflectivity especially suitable for building self-monitoring and self-evolving systems.

2.4 Scoping the effect of changes

An interesting problem arises when instrumenting basic system classes. The instrumented code itself is used at runtime by the meta-object, leading to endless loops. This makes any use of reflection on basic system classes like *Number* or *Array* impractical. This problem can be seen with all reflective systems — a well-known example is CLOS [6].

To solve this problem, we provide the possibility to scope the activation of links towards meta-level execution [14]. Links are parameterized with the level for which they are activated. This way we can restrict the introduced change towards, for example, base-level program execution. Note that the same mechanisms can be used to reason about execution of meta-level programs. For example, it may be the case that a profiler realized as a meta-object needs to be analyzed to improve performance. By restricting the link that activated the profiler to the meta-level, we can use the profiler on itself without the danger of endless loops.

2.5 Implementing higher-level dynamic language features

With partial behavioral reflection, we can easily adapt the programming language to support new language features. Very deep changes can be realized that normally require changes at the level of language implementation (*i.e.* the virtual machine).

Dynamic aspects. Partial behavioral reflection can serve as an efficient technique for implementing *Aspect-Oriented Programming* [39]. We have used the dynamic features of Reflectivity to implement *dynamic aspects*, which are not woven into the code at compile time, but instead can be introduced and retracted at runtime. Compared with traditional runtime AOP implementation techniques, we can see some improvements. We can generate better code than typical bytecode transformation based approaches as we can leverage the higher-level AST representation [9]. We can leverage the link-conditions to efficiently control aspect activation at runtime.

Transactional memory. We have realized software transactional memory for a dynamic language [37]. It is notable that this realization was done without any changes to the underlying virtual machine. With the help of Reflectivity we introduced a transactional execution context and were able to reify all state access to be handled by the transactional model implemented in the host language.

3 Diesel — an engine for bringing models closer to code

We have seen that it is important to have higher level models of software available at run-time in order to enable dynamic software adaptations. But what about the application logic itself? Models of the software structures can be far removed from the application domain. What we need to enable dynamic adaptation of application logic is to make application models more explicit in the code. One way of doing this is to raise the level of programming by means of specialized domain specific languages (DSLs).

This raises new, extrinsic problems, as a new DSL will not be able to automatically benefit from existing development tools available for the general-purpose host language. We need to be able to adapt the host language and environment to support new DSLs. As before, the adaptions must be *fine grained* because DSL code may be interspersed with regular source code, *dynamic* because we want to be able to change host compilers and tools on the fly, and *context-dependent* because DSL code may be restricted to certain parts of an application.

Diesel is a lightweight language workbench that closely integrates with the host language [16]. This enables developers to incrementally bend the syntax and semantics of the host language to suit their exact needs for a particular problem domain. As such, Diesel is an environment for developing domain specific languages (DSLs), with the aim of giving application developers more suitable abstractions than the host language provides. Contrary to other approaches, Diesel reuses the traditional compiler toolchain and closely integrates with the existing tools of the programming environment, such as editors, debuggers, inspectors, *etc.* A close integration is crucial to keep the abstraction gained through new language features. While language developers might want to toggle between a view on the original source and the transformed result, domain developers would like to stay at the abstraction level of their code at all times.

The host language of our implementation is Smalltalk, which provides us with a uniform development environment. The abstract code representation of the compiler is reused in all parts of the system and can thus take advantage of our extensions. For example, if we change the syntax in a specified part of the system, syntax highlighting and debugger continue to work. Application developers do not have to learn new tools, but continue to use the existing ones even if they mix multiple languages.

3.1 Example: modelling relationships

A common challenge in transforming UML models to code is how to implement relationships between objects. The problem has long been solved in relational databases [2,32], but none of today's mainstream languages provide first-class relationships [29]. If done by hand, it is easy to introduce subtle bugs that might be very hard to detect. With code generation, huge chunks of boilerplate code may appear that are hard to understand and impossible to change. In either case, debugging and maintaining the code is cumbersome, because developers

not only have to think in terms of the high-level DSL code that they specify, but also in terms of the code that is generated.

We present an approach to solve this problem with Diesel. The example only handles 1:1 relationships, however it could easily be extended to support arbitrary n:m relationships.

To implement the write accessor `next:` of a double linked list, a developer or a code generator would write something like this:

```
1 Link>>next: aLink
2   next isNil ifFalse: [ next instVarNamed: 'prev' put: nil ].
3   next := aLink.
4   next isNil ifFalse: [ next instVarNamed: 'prev' put: self ]
```

At run-time lines 2 and 4 are need to ensure that the inverse relations are properly updated. The actual assignment only occurs on line 3. Most parts of the code are not interesting to developers. It is an unnecessarily complex code fragment specifying an implementation detail of 1:1 relationships that is probably used at several places throughout the application.

We now replace the complex write accessor from above with a plain write accessor that does not update the opposite relationships:

```
Link>>next: aLink
   next := aLink
```

However we put high-level annotations next to the instance variable declaration of the `Link` class, to tell Diesel that all write access to these variables requires their respective inverse relationships to be updated:

```
Link instanceVariables: #(
   next <opposite: prev>
   prev <opposite: next>
)
```

When compiling the method, Diesel will automatically generate the necessary boilerplate code around it. This generated code is never visible, not even in the debugger, to ensure that the developer can concentrate on the high-level model. The magic behind the transformation comes from a rule that has been added to the Diesel engine. Whenever source-code is parsed, translated and annotated these rules are processed to enable interaction with the compiler:

```
1  TreePattern
2    match: '`variable := `expression'
3    do: [ :context |
4      variable := context at: '`variable'.
5      opposite := variable annotationNamed: 'opposite'.
6      opposite notNil ifTrue: [
7        context addNodeBefore: ``(`,variable isNil
8          ifFalse: [ `,variable instVarNamed: `,opposite put: nil ]).
9        context addNodeAfter: ``(`,variable isNil
10         ifFalse: [ `,variable instVarNamed: `,opposite put: self ]) ]
```

The rule given above matches all parse tree nodes that assign an expression to an instance variable (line 2). The remaining code checks if the instance variable is annotated with an opposite annotation (lines 5–6) and then defines the transformation programmatically. This is done using a quasi-quoting mechanism [3] to build and inject the AST nodes that update the opposite relationship into the tree. Subsequently the tree is transformed to bytecodes, by the standard Smalltalk compiler. The handwritten and the transformed code result in identical bytecodes, therefore both approaches perform equally at runtime. In practice the minimal increase in compilation time due to the additional transformations can be neglected.

3.2 Scoping the effect of changes

In the above example the scope of the transformation is given at the level of parse tree nodes. Often such transformation rules only apply to a carefully chosen part of the system however. For example, the above transformation should be used in model code only, but not in the UI implementation. Diesel supports a wide variety of additional constraints that can be composed and added to transformation rules: packages, namespaces, classes, class hierarchies or even specific methods.

Furthermore arbitrary conditions can be added to the transformed parse-tree nodes, so that the transformation is only in effect if a certain runtime condition is met. The generated code can resort to the reflective capabilities of the system [38] and select the appropriate behaviour depending on the runtime context [7].

4 Towards a research agenda

Full reflection (*i.e.*, with run-time intercession) has been widely available in dynamic programming languages for many years, notably in Smalltalk [38] and CLOS [22]. Nevertheless, reflection has been commonly considered to be either too dangerous or too difficult to use for common programming tasks. Static languages such as Java and C++ offer a weaker form of reflection that only supports introspection at run-time (*i.e.*, the possibility to examine but not to affect the model elements).

There is increasing pressure to adapt software systems at run-time. If the host programming language does not offer reflective features, programmers can be forced to adopt workarounds, such as reifying and interpreting model elements within their programs. This obviously places a heavy burden on developers who must build up this infrastructure themselves, possibly in *ad hoc* ways.

The Reflectivity framework simplifies the development of reflective applications by offering ASTs as relatively high-level, causally connected, run-time models of program elements. Reflection at the sub-method level is enabled since the deep structure of programs is captured, unlike in other approaches which stop at the method level. We have used the Reflectivity framework extensively for various purposes, including the analysis of software features [13], dynamic

monitoring of software entities from the IDE [41], and even for the implementation of pluggable types, where type expressions are encoded as source code annotations and interpreted by compiler plug-ins [19].

Despite these successes, Reflectivity is focused on reifying model elements of the host programming language, not those of the application domain. (One could also say that the application domain of Reflectivity *is* the host programming language.) What we are missing is Reflectivity for domain models. We envision a system where end users can change their domain models on the fly, without having to touch the host programming language [36].

Traditionally domain concepts are translated and encoded in the source code in such a way that makes it difficult to reason about these concepts once the software is deployed. It is often difficult, for example, to find the software components responsible for a given end-user feature in the source code. In a model-driven approach, one would express domain concepts at the level of meta-models and models, and then generate code from these descriptions. In a straightforward approach, this still has the consequence that the domain models are no longer directly expressed in the code. If features must be dynamically adapted, there is no easy way to manipulate these model elements from the running system. Instead of generating code from models, we feel that it is necessary to *bring models closer to code*. This means that domain concepts should be expressed directly in the source code, rather than being encoded using concepts of the solution domain. In essence, rather than applications being model-driven, with models merely being used to generate code, we believe that they should be *model-centric*, with models being first-class entities that can be manipulated at run-time.

One way of bringing models closer to code is to provide higher-level, domain specific languages for model concepts more directly. One downside of this approach is the potential for the proliferation of DSLs, each with their own obscure syntax. DSLs should therefore be simple and lightweight. Another important downside is that the existing development environment will need to be adapted to work with each new DSL. Diesel addresses these problems by offering a lightweight framework for specifying simple DSLs that are transformed into the ASTs used by Reflectivity. The transformations are used to keep the development tools in sync with each DSL, so that editors and debuggers, for example, can present developers with the original DSL code rather than the generated host language code. In a sense, the model *is* the code.

Returning to the theme of software adaptation, we note that any adaptation manifests itself as a kind of software change, which is possibly intrusive and possibly context-dependent. As an example, consider software that should adapt its policies for ensuring changing requirements (*e.g.*, related to concurrency control, or security, or transactional behaviour) dynamically according to (i) the run-time context (*e.g.*, the presence of competing applications), and (ii) availability of related services (*e.g.*, for optimistic or pessimistic transaction support). Such an application has clearly defined requirements but can only partially anticipate its run-time context and the nature of services available to meet those requirements.

Depending on the context, the same software will need to behave differently, and may need to be adapted in unanticipated ways.

We summarize the research directions that we see as essential to support dynamic software adaptation as follows:

Bring models closer to code. In order to enable dynamic software adaptation, models should be first-class, high-level artifacts available at run-time for both introspection and intercession. Structured source code and run-time annotations offer one light-weight technique to embed domain knowledge in source code [28]. Lightweight DSLs are another promising technique.

Model-centric development. Rather than seeking ways to embed models in source code, perhaps we should replace the source code as an artifact and directly program with models. After all, third generation languages were once seen as a way to "generate (machine) code" from high-level specifications. Nowadays we consider programs in these languages to be the source code. We need to take the next step and jettison our third generation object-oriented languages in favour of models as source code. Environments to support model-centric development would concentrate on directly manipulating first class models and their meta-models. Models would be available at run-time to support the same kinds of adaptations available to the developer at development time.

Context-oriented programming. COP refers to programming language mechanisms and techniques to support dynamic adaptation to context [21]. Although many present-day applications need to be context-aware, context-dependent behaviour is generally programmed in *ad hoc* ways, due to the lack of support in modern programming languages. Some COP languages have been proposed [7,18], and both Reflectivity and Diesel are examples of a frameworks that provide some degree of COP support, but research is very young, and there is no consensus how best to support COP in programming languages.

5 Related work

There is a long history of reflective programming languages, ranging from dynamic languages such as Lisp, Smalltalk, Scheme and CLOS, to static languages like C++ and Java, which provide a more limited form of run-time introspection rather than full intercession. All of these approaches are limited to models of the code base, and do not take models of requirements, design decisions or architecture into consideration.

Over the years various approaches have attempted to keep high-level knowledge about software in sync with the software itself. The earliest examples of these is probably Literate Programming [24], in which documentation and source code are freely interspersed and maintained together.

In some cases Architectural Description Language (ADL) specifications are considered to be part of the running software system, rather than simply a higher-level description of it, as it is the case with Darwin [27]. Although ADLs

provide a high-level interface for specifying and configuring components at an architectural level, there is not really any explicit representation of a model that is developed in tandem with the rest of the software.

Generative programming approaches [8] produce software from higher-level descriptions using such mechanisms as generic classes, templates, aspects and components. A general shortcoming of these approaches is that the transformation is uni-directional — there is no way to go from the code back to higher-level descriptions. Round-trip engineering refers to approaches in which transformations are bi-directional [1]. Models and code are still considered to be separate artifacts, so models are not available at run-time for making adaptive decisions.

Case tools and 4GLs represent an attempt to simplify the generation and adaption of an application. However the main focus of these approaches was to generate code and not to consider models as executable artifacts of software development.

Model-driven engineering (MDE) [4,42] refers to a more recent trend in which application development is driven by the development of models at various levels of abstraction. Platform-independent models are transformed to platform-specific models, and eventually to code which runs on a specific platform. Generally these transformations are performed off-line, so models are not necessarily available to the run-time system, though some approaches support this [20].

The Eclipse Modeling Framework (EMF) [5] provides facilities for manipulating models and generating Java source code from these models. Here too the focus is on models of source code, rather than on other views of a software system. The model and the code are still separate entities.

Naked objects [35] is an approach to software development in which domain objects and software entities are unified. Business logic is encapsulated in the domain objects and the user interface is completely generated from these domain objects. In this approach the domain model and the executing runtime are tightly coupled. Although naked objects address the earlier complaint against approaches which separate domain models from the source code or the running system, they do not offer any help in integrating other views of the software as it is being developed (*i.e.* requirements models, architectural views, and so on).

Aspect-Oriented Programming (AOP) [23] provides a general model for modularising cross cutting concerns. *Join points* define points in the execution of a program that trigger the execution of additional cross-cutting code called *advice*. Join points can be defined on the runtime model (*i.e.*, dependent on control flow). Although AOP works at a sub-method level, it does not provide a structural model of the system or any other reflective capabilities. The goal of AOP is to modularize crosscutting concerns, not to provide a model for dynamic software adaptation.

Reflex [47] pioneered partial behavioral reflection in the context of Java. Here links are associated with so called *hooksets*, abstractions of operations at the bytecode level. Therefore, the structural model of Reflex is that of bytecode, not the higher-level AST. In addition, Reflex does not support meta-annotations on bytecode.

Context-Oriented Programming (COP) [21] refers to programming language support for developing applications whose behaviour depends on the run-time context. Present prototypes of COP languages focus on mechanisms for adapting behaviour to context, but provide little support for reasoning about context at the model level.

Changeboxes [11] provide a mechanism to control the scope of change in a running system. Deployment and development versions of a running software system can co-exist without interfering. Mechanisms for merging differences and resolving conflicts, however, must be handled in an *ad hoc* fashion, as no fully general approach exists for all usage scenarios. Changeboxes currently operate at the level of source code changes. There is no notion of changes to higher-level models.

Unlike general-purpose programming languages, domain specific languages (DSLs) tend to be compact languages that provide appropriate notations and abstractions for a particular problem domain. It was shown that DSLs increase productivity and maintainability for specialized tasks [15]. DSLs are often categorized as being either homogenous (internal), where the host-language and the DSL are one and the same, or heterogeneous (external), where the two languages are distinct [44]. Techniques have been proposed to define language and semantics for new DSLs [25]. The idea of designing languages that embrace the addition of new DSLs has been a focus of research in the past [33,49,48]. However integrating new languages into existing tools has been largely neglected.

In the 1990s there was considerable interest in the development of architectural description languages (ADLs) [43] to capture and express architectural knowledge of a software system. ADLs can be viewed as DSLs for describing the architecture of complex software systems. Many DSLs formalize architecture in terms of components, connectors, and the rules governing their composition [43]. This idea is also implicitly contained in the notion of *scripting languages*, which can be seen as DSLs for composing applications from components written in another, usually lower-level programming language [34]. Despite this, the interplay between conventional object-oriented languages, ADLs, scripting languages and DSLs has not yet been thoroughly studied nor has it been exploited in practice.

6 Concluding remarks

Software systems are under increasing pressure to support run-time adaptation for localization, mobile platforms, dynamic service availability, and countless other context-dependent applications. Unfortunately mainstream programming languages and development environments focus on limiting and restricting change rather than enabling it. Specifically, dynamic, fine-grained and context-dependent software adaptation is not well-supported by modern development technology.

In this paper we have presented two ongoing research projects that illustrate the principle of model-centric, context-aware software adaptation, and we have outlined a number of promising research directions for further exploration.

Reflectivity is a mature, model-centric framework for dynamic software adaptation. Software structures are represented at run-time by means of abstract syntax trees, which enables both dynamic and fine-grained adaptation. Furthermore, by means of partial behavioural reflection, adaptations can be scoped to the dynamic context. As a typical example, run-time instrumentation can be scoped to the context of a given feature, so it is possible to identify which software components support a given feature.

Diesel is a new framework for bringing models closer to code by supporting the definition of lightweight DSLs that are transformed to the host programming language using the same high-level source code models as provided by Reflectivity. The development tools, such as editors and debuggers, are aware of the transformations, so developers can continue to work with the high-level models rather than the transformed code, even while debugging. Here too, run-time models are used to support fine-grained adaptations that can be scoped to specific parts of the application requiring these DSLs.

Reflectivity and Diesel both build on top of the same rich programming environment and therefore could profit from each other in the future. On one hand, the causal connection of model and code would help Diesel to automatically propagate language changes to all its users. Furthermore Diesel could apply language transformations that normally happen at compile-time from the dynamic world. On the other hand, Reflectivity could profit from a richer language infrastructure and the compiler plugins could reuse the Diesel transformations.

These are only two examples of promising research directions to support dynamic change and software adaptation. We argue that more research is needed to close the gap between models and code. In general, software systems need to be *change-enabled* — instead of limiting and restricting change, they should actively enable change by treating change as a first-class entity [30,31]. Ultimately we want model-centric and context-oriented programming environments where we can directly manipulate models both during development time and run-time, and software can be dynamically adapted by context.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

We thank Tudor Gîrba and Jorge Ressia for their careful reviews of various drafts. We also thank the anonymous reviewers for their numerous suggestions on how to improve the presentation of this paper.

References

1. Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, volume 4199 of *LNCS*, pages 692–706, Berlin, Germany, 2006. Springer-Verlag.
2. Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–170, 1987.

3. Alan Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.
4. Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings of Automated Software Engineering (ASE'01)*, pages 273–282, Los Alamitos CA, 2001. IEEE Computer Society.
5. Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.
6. Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.
7. Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, pages 1–10, New York, NY, USA, October 2005. ACM.
8. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
9. Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.
10. Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.
11. Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.
12. Marcus Denker, Orla Greevy, and Michele Lanza. Higher abstractions for dynamic analysis. In *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.
13. Marcus Denker, Orla Greevy, and Oscar Nierstrasz. Supporting feature analysis with runtime annotations. In *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007)*, pages 29–33. Technische Universiteit Delft, 2007.
14. Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237, 2008.
15. Arie van Deursen and Paul Klint. Little languages: Little maintenance? In S. Kamin, editor, *First ACM-SIGPLAN Workshop on Domain-Specific Languages; DSL'97*, pages 109–127, January 1997.
16. Martin Fowler. Language workbenches: The killer-app for domain-specific languages, June 2005. `http://www.martinfowler.com/articles/languageWorkbench.html`.
17. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.
18. Sebastián González, Kim Mens, and Patrick Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 77–88, New York, NY, USA, 2007. ACM.

19. Niklaus Haldimann, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types for a dynamic language. *Journal of Computer Languages, Systems and Structures*, 35(1):48–64, April 2009.
20. Stefan Haustein and Jörg Pleumann. A model-driven runtime environment for web applications. *Software and System Modeling*, 4(4):443–458, 2005.
21. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.
22. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
23. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.
24. Donald E. Knuth. *Literate Programming*. Stanford, California: Center for the Study of Language and Information, 1992.
25. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated definition of abstract and concrete syntax for textual languages. In *Proceedings of MoDELS 2007*, volume 4735 of *LNCS*, pages 286–300. Springer Verlag, 2007.
26. Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.
27. Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeffrey Kramer. Specifying distributed software architectures. In *Proceedings ESEC '95*, volume 989 of *LNCS*, pages 137–153. Springer-Verlag, September 1995.
28. Philippe Marschall. Persephone: Taking Smalltalk reflection to the sub-method level. Master's thesis, University of Bern, December 2006.
29. Stephen Nelson, David J. Pearce, and James Noble. First class relationships for OO languages. In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2008)*, 2008.
30. Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, and Adrian Lienhard. Analyzing, capturing and taming software change. In *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.
31. Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, Adrian Lienhard, and David Röthlisberger. Change-enabled software systems. In Martin Wirsing, Jean-Pierre Banâtre, and Matthias Hölzl, editors, *Challenges for Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, pages 64–79. Springer-Verlag, 2008.
32. Brian Nixon, Lawrence Chung, John Mylopoulos, David Lauzon, Alex Borgida, and M. Stanley. Implementation of a compiler for a semantic data model: Experiences with taxis. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 118–131, New York, NY, USA, 1987. ACM.
33. Martin Odersky. Scala language secification v. 2.4. Technical report, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, March 2007.
34. John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.
35. Richard Pawson. *Naked Objects*. Ph.D. thesis, Trinity College, Dublin, 2004.
36. Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. Magritte — a meta-driven approach to empower developers and end users. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 106–120. Springer, September 2007.

37. Lukas Renggli and Oscar Nierstrasz. Transactional memory in a dynamic language. *Journal of Computer Languages, Systems and Structures*, 35(1):21–30, April 2009.
38. Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996.
39. Leonardo Rodríguez, Éric Tanter, and Jacques Noyé. Supporting dynamic cross-cutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, Arica, Chile, November 2004. IEEE.
40. David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.
41. David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Exploiting runtime information in the IDE. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008)*, volume 0, pages 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
42. Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.
43. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
44. Tim Sheard. Accomplishments and research challenges in meta-programming. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.
45. Eric Tanter. Reflection and open implementations. Technical report, University of Chile, 2004.
46. Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *LNCS*, pages 227–242, Vienna, Austria, March 2006.
47. Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.
48. Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM TOPLAS*, 30(6):1–40, 2008.
49. Alessandro Warth and Ian Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.
50. Martin Wirsing and Matthias Hölzl (editors). Report of the Beyond the Horizon thematic group 6 on Software Intensive Systems, 2006.