Abstract
Model Driven Engineering has shown the feasibility to generate tools (editors, analyzers ...) from models and for a domain. Unfortunately these generated tools are much focused while a large application spans different domains and different activities; currently these tools do not support concurrent engineering, and incomplete life cycle support.

In a similar way we developed a technology capable of generating a complete Computer Aided Domain Specific Environment (CADSE) from a set of model and metamodels describing the specific domain and the environment behavior. Our technology solves the above two problems, allowing defining a number of CADSE addressing specific activities, as extensions and adaptation of a core CADSE; and relying on workspaces. CADSEs workspaces are model driven; they contain tools, models and usual artifacts (code, documents...); they support different activities by switching from an activity to another one, and concurrent engineering is supported through import/commit applied to models, metamodels and artifacts.

1. Introduction

Traditional Code Driven Engineering environments are supporting the collaborative work of different actors, working on different activities and phases of the software lifecycle, over a shared set of artifacts. The key concept for that support is the workspace: an isolated private area which contains only the artifacts needed to perform a given task, but that can be synchronized with a public shared repository.

The concept of workspace is the cornerstone of both Integrated Development Environments (IDE), and versioning systems. IDEs are performing activities in the workspace, oblivious of the concurrent engineering issues, while version and configuration systems, conversely, are addressing evolution and concurrent engineering control through import/commit between workspace and a shared repository, irrespective of the tools used and activities performed in the workspaces. Modern IDEs are dynamically extensible and customizable. It is therefore possible to build a very specialized environment that fits the needs of a particular actor performing a given task in the software lifecycle; without sacrificing the possibility of easily switching activities, working concurrently and sharing the manipulated artifacts. In Code Driven Engineering, it is common practice to switch from activity to activity in the same environment, and to benefit from evolution and concurrent engineering control. Unfortunately, this is not true (yet) in Model Driven Engineering (MDE) approaches.

Indeed, MDE, when applied to specialized domains, such embedded systems or business information systems, also aims at producing specialized environments that fit the needs of experts in the domain. Most Domain Specific Modeling tools rely on the generative approach in which the tools handling models (editors, validators, transformers, etc) are partially generated from a domain meta-model. This generative approach, clearly, reduces the cost and time needed to produce such specialized environments, but little attention has been paid so far, on concurrent engineering issue: how the generated tools will allow multiple actors to perform simultaneously different activities over the same artifacts; while we do believe it is a mandatory feature.

Our environment CADSEg (Computer Aided Domain Specific Environment generator) [1][2] is a Domain Specific Modeling tool, but our emphasis is not the generative technologies per se, but rather the concurrent engineering issue: how the generated tools support concurrent activities. Concurrent engineering imposes some obvious requirements on model driven tools regarding model repositories, model versioning, and the ability to merge and compare models. Model versioning is a tough issue that cannot be solved “simply” like in CDE, storing the different “versions” of a model. In CADSEg, a specific model describes how is to be handled model and code co-evolution, and which is the versioning strategy to use [3].

We realized soon that this approach solves the evolution and concurrent engineering issues in the case
where the different actors are working with the “same” model in different workspaces. But in a domain specific context, a model is specialized for a specific kind of activity, taking place in a specific life-cycle phase of a software project. Nevertheless, these different activities share software artifacts (codes and documents), and therefore concepts and model elements. Evolution control requires a wider view subsuming the different local views as used by the different activities.

This is the case with our generated CADSEs. A CADSE is specialized for a given domain, be it business or technology driven. Even within the realm of a narrow domain, there is a need to support different activities, performed by different experts, from design to deployment and run-time monitoring, for instance. A CADSE is thus not only dedicated to a specific domain but also to a specific activity. The support provided by each CADSE, and therefore the models and metamodels associated with these activities are significantly different. In practice we are not dealing with independent meta-models: when two experts are working concurrently in two different workspaces, they are not really producing two different models that need to be eventually synchronized; they are working on the same artifacts and the same model elements. We have observed that, very often, activities in a domain share a common set of core concepts that represents the common metamodel that allows collaborative work to proceed, but at the same time this core metamodel is extended to cope with the needs of each particular activity.

From this observation, we have enhanced CADSEg to natively support a mechanism that allows modularizing a domain meta-model into a core meta-model and a number of extensions specialized to particular activities. Notice that this is similar to Aspect Oriented Modeling [4], applied at the meta-model level, where the concerns are the different software life-cycle phases and activities. This is the main idea that will be further developed in the rest of the paper.

Section 2 presents what a CADSE is and how it is produced. Section 3 discusses the need for model driven workspaces. Section 4 describes our motivation for CADSE extensibility from a domain expert point of view. Section 5 explains how it is realized according to CADSE user needs. Section 6 discusses experiences of CADSE extensibility. Finally, section 7 presents related works and section 8 concludes and proposes future works.

2. CADSE approach

Most MDE platform such as GME [5] or MetaEdit+ [6], are capable of generating tools from its description (a model of that tool), or tools supporting predefined activities (like editing) on all models conform to a given metamodel.

![Figure 1. Model driven generative approach.](image1)

In the same vision, our environment CADSEg takes in input the models that define a CADSE and generates the corresponding CADSE. Today, we generate Eclipse plug-ins which extend Eclipse with our editors, generators, builders and so on.

We take advantage of Eclipse extensibility to provide the user with the specific environment he/she needs. It allows the engineer to switch at any time from the generic Eclipse to the CADSE domain and activity specific and conversely.

![Figure 2. Model driven generative approach bis.](image2)

A CADSE is defined by a metamodel and a set of models. The metamodel, called the data model, is a kind of class diagram where classes represent the different concepts with their attributes and their relationships. This language is very similar to eCore [7]; it describes the structure of the target application, but not its semantic and behavior. The other CADSE models describe the CADSE behavior: how the environment should supports users in their everyday tasks.
- **Interaction.** In a CADSE, a large part of the activity is performed on the application model through generated editors. To each model entity type is associated an interaction model, from which is generated a wizard. During the user activity performed on the model, the wizards assist on creating, modifying and deleting model elements and their links. Each editor is called a view which defines what the elements currently visible are and the actions allowed on them.

- **Mapping.** To each model entity type is associated a mapping model which expresses what are the artifact(s) to be associated to each instance of that type, in term of Eclipse resources like project, directory or files. It is a model in which the expert describes the synchronization needed between the physical artifacts and their corresponding model elements in both ways. This synchronization mechanism uses generators, mergers and so on that the expert must specify.

- **Build.** As the first obvious requirement of a CADSE is to construct an application, expert must also define how applications are built.

- **Evolution and concurrent engineering.** While evolution control is well supported today for files, model evolution control and versioning is a current research area. A CADSE model describes with annotations on the data model the way the application model can be evolve and how it is versioned in coordination with the related files and directories it represents [3].

### 3. Workspace: a model and artifact view

In our approach, the repository contains all the information produced during the software project; i.e. all the model elements, their associated artifacts and their versions. It also means that the repository “knows” all the metamodels in order to manage consistently the model elements, their mappings, in accordance with the evolution policy (associated to the metamodel).

This is too much and confusing information for the user in charge of performing a very specific activity on a tiny sub-set of the repository. For this reason CADSE relies heavily on an extended concept of workspace, seen as a limited repository view both on the data (instances) and metamodel (types, relationships and characteristics). The developer imports only the model elements (and consequently their associated artifacts) in which he is interested in, limited to the information (metamodel view) required by his/her current activity (sub-set of attribute and relationships, specific mapping etc.). This view is extracted from the global model at import time.

To illustrate this point with a familiar example, consider the CADSE support needed for a team of engineers in charge of developing, testing and deploying Service Oriented Architecture based applications. In our approach, the first step is to design a meta-model for this domain of applications representing the main artifacts managed by the engineers; this may looks like the simplified data model presented in Figure 3.

![Figure 3. Service Oriented application example.](image)

From this data model, and the other related domain models presented in section 2, CADSEg generates a number of tools: an editor for creating models of a service-based application, different code generators, and the schema of a global repository that will contain the versioned elements (model and artifacts) developed by the team.

In this example it is clear that all team members do not work simultaneously on the whole model, and that the generated tools must allow managing the model (and artifacts) in the workspace at a finer level of granularity.

A scenario can be the following: an architect, using the generated editor, creates a *ServiceX* description in his workspace (in the simplified data model of Figure 3 this is a single object, but we can imagine that this can be a complex sub-model on itself). When the architect considers that the description is correct he/she commits the workspace to the repository. At this point, the quality engineer can import the *ServiceX* description to his/her own workspace and start developing the corresponding *Tests*. Simultaneously, the component engineer can import *ServiceX* description and start working in a workspace on the *Component* implementation, and defining its required *Services*.

This example shows that each team member works on a part of the model in an isolated workspace, like in the usual code based engineering, and that the private versions produced will be merged into the global repository when a commit is performed. How the merge should be performed, and what are the effect of the modification on the versions existing in the repository, is specified in the domain evolution model, so that the
generated CADSE editors are aware of the versioning policies and strategies. We also notice that there is a core set of concepts (in our example the Service class) in the meta-model that are shared by all actors and constitute the backbone around which is articulated the collaborative work.

This example also shows that this basic CADSE functionality, although powerful, is not really enough to support the specialized needs of each kind of actor. For instance, up to now we have made the hypothesis that all actors are using the same CADSE (generated by CADSEg from the meta-model of Figure 3) and that everybody can perform any activity in his workspace. In a more realistic scenario, we expect that the quality engineer can only import Service descriptions in read-only mode and that testing requires importing Component implementations in their binary form. Conversely, the architect should not be bothered by test suites; and developers should get the source code of components, not its binary code. Clearly, each kind of actor requires a different CADSE tailored to the activity to be performed.

We have then some conflicting requirements. On the one hand, we would like to have a global metamodel (like the one in Figure 3) needed, at least, by the repository, in order to keep track of all the relationships among all versioned parts of the model at a fine level of granularity. We also need to work easily on shared models, and to support evolution using the traditional import/commit functions between a shared repository and many private and isolated workspaces. On the other hand, we would like to have independent models and metamodels (like the shaded regions in Figure 3) specialized by activity, so that we can generate different environments for different needs.

Our proposition (detailed in sections 4 and 5) is to use an Aspect Oriented approach for building the domain meta-model: each kind of activity is considered to be an independent aspect, and each aspect is described by a meta-model fragment; the global meta-model is obtained by weaving those metamodel fragments with a core metamodel containing the concepts shared by all activities in the domain.

The real challenge is not to merge the syntactic part of the data model (as illustrated in our example, and as performed by most MDE platforms) but to weave, merge and consistently manage the associated semantics as described in the associated models which are used to generate a CADSE and the corresponding generated tools.

To illustrate the issue consider the model used to generate the editor. In our example, the definition of the core domain metamodel includes not only the data-model (the Service class in our case) but also the definition of a Service editor. When we define the aspect Test, we will add some classes and associations to the data model and we will define the behavior of the editor for this particular concern. However, we do not want to fully redefine the existing editor, but only to describe what has to be changed/extended with respect to the base editor defined in the core. In our example, it is enough to say that the class Service is read-only.

4. Extensibility for domain experts

Engineers are often responsible of multiple activities. In our example the same engineer could be in charge of developing and testing the application. In the multiple CADSE scenario presented above, he/she should start developing the components and commit the work when done. But to test, he/she should delete his current workspace, create a test workspace and import the components. This is very inconvenient, it wastes time, energy, and resources since each workspace, in practice may be very large.

In comparison, consider the extensible IDEs such as Eclipse or NetBeans. They allow users to perform different activities in the same workspace; which avoids data duplication and synchronization issues. We want to apply the same extensibility pattern on our CADSEs. After multiple different designs, we find out that CADSE specialization, CADSE composition and multi level abstractions can be realized by a single mechanism: CADSE extensibility.

A CADSE can extend one or many other CADSEs. This relationship is acyclic. We do not define what the aim of an extension is, only what are the specific aspects that this CADSE extends.

When launching for the first time a CADSE, the user is asked to tell what CADSE extensions are to be activated. To switch to another task, the user only has to activate or deactivate related CADSE extensions.
4.1. Data model extension

A data model defines concepts, attributes and relationships between concepts. A CADSE extension can define new concepts and relationships or extend existing concepts by adding new attributes and relationships. Therefore, extensions can both extend and specialize concepts. Removing and overriding attributes and relationships have been intentionally disallowed to avoid complex conflicts between extensions.

An inheritance relationship between concepts may exist which is very useful to specialize existing concepts for a specific activity. However, we encourage extending existing concepts rather than sub-typing in order to reuse the same data without complex type migration between activities.

4.2. Interaction extension

A CADSE identifies three interaction types:

- **Views.** A CADSE may contain multiple editors called a view. A view defines what the model elements to show are and what the actions allowed on them are in this specific editor. A CADSE extension can define new views.

- **Model edition** i.e. creation and modification of model elements. It is the most important part of the interaction definition. Creating a model element of a given type involves the initialization of its name, its ID, its attribute values, its links with the other elements. To cope with this issue, the user is assisted by the element type creation wizard that can fill some values or asks the user to provide values for some attribute and link. A wizard is modeled as a sequence of pages containing a list of widgets to edit values called fields. A CADSE extension can add, replace or specialize these pages. Page specialization is limited to changing a field editor by another one, but the underlying attribute and its type remains the same. However new pages supporting the edition of new attribute can be inserted. Modification on model elements can be performed through property pages. Property page definition is similar to creation wizard. An extension can also add, replace or specialize property pages.

- **Actions.** An action is an interaction on a model element which can be performed through an editor. An action specification defines its name and its semantic (the operation to execute). A CADSE extension can add actions on existing concepts. To avoid conflicts, it is not possible to remove and override existing actions.

Altogether, these interaction extensions deeply change the visibility, interaction and operations that can be performed; they can really be adapted to the activity at hand.

4.3. Versioning extension

The evolution metamodel defines a number of annotations that can be associated with any attribute or link defined in the data model. The evolution metamodel also defines the concept of version, repository and operations commit, import, update and so on defined on all model elements.

Attributes annotations (immutable, mutable, final, and shared) specify what should be the consequences with respect to evolution control, of an attempt to change an attribute value. *Final* means that the change is prohibited; *mutable* means that the attribute can be changed without side effect on other elements; *immutable* means that a new revision of the current element must be created. Relationships also can be annotated. Since relationships link two objects, they can define how changes performed on the destination object propagate to its origin object. To that end we defined the following annotations: *mutable* mean that there is no propagation; *branch* means that the links is valid for all revision of the destination; *effective* means that the link is valid for a range of destination versions; *immutable* means that we must create a revision of the origin object each time the destination object changes of version.

Based on this information, the system can perform extensive consistency control computation when importing a revision of an item in a workspace. For example, suppose you want to import an item in a workspace. The items already present in the workspace may have relationships to and from that imported item. In reality, the workspace contains specific versions of these items, and the relationships indicate which version(s) are compatible. Sophisticated evolution strategies, including those currently used in Software Engineering can be defined, and automatically supported, using these evolution annotations [3].

Since an extension may add new attributes and relationships definitions, it must also define the associated evolution annotations. Overriding an existing annotation is prohibited. It ensures that evolution control policy is shared by all engineers.

4.4. Mapping extension

A mapping is the definition of a two ways transformation between actions performed on a model element, and actions performed on the corresponding
computer artifacts. Defining mapping extensions requires the capability to modularize partial mapping, while maintaining consistency. One way transformation modularity is a current research topic recognized as a hard issue [8] and we must cope with modularity of bidirectional synchronization [9].

Of course, an extension must define the mapping of the new concepts, but we also allow to add a mapping to existing concepts, and to replace a mapping by another one. If the core-Service CADSE maps a Service to a Java interface, a documentation CADSE extension can attach in addition to the Java interface a document which describes guidelines of using this service. If the core-Service CADSE maps a Service component to an Eclipse Java project, the deployment CADSE extension can map a component to an executable file.

Figure 5 is a screen shot of our tool when defining the Test extension. In the right side panel, at the first level is found the service.core CADSE and its service.test and service.development extensions. In service.test, we can see that the data model defines a new concept: Test, with a relationship test leading to the service concept, defined in the service.core data model. Note that the user is currently defining the evolution-model for the test relationship. In the bottom left part of the figure, the user is defining that the test relationship is mutable (as an attribute) and immutableDestination.

5. Extensibility for users

The practice shows a large variability in the way developers allocate their time to activities. Sometimes the developer waits for a task completion before to start a new one (for example for some critical activities), but most often the developer switches many times between related activities.

5.1 Multiple activities.

From the user point of view, the need is to switch easily from an activity to another one; which means switching from the CADSE supporting the first activity to the CADSE supporting the new activity; of course without any need to commit / import the needed items if they are already present. In other word, the same data must change its appearance and behavior without being changed in any way; it is the way the data is interpreted that must change, not the data itself.

It may be sufficient to statically define the relevant combinations of CADSEs and the legal switches, in order to enforce some predefined process (RUP for example) and the company good practices. In practice, however, as in Eclipse (where plug-ins can be installed freely), it is often the user who decides what to do next. Experience shows that this feeling of being in charge is a major acceptability factor. In our context, it means that the user must be allowed to dynamically install the CADSE extension of its choice (indeed, a CADSE extension is an Eclipse plug-in) and use it immediately, without any need to restart the system or re import the data. Once installed the user is able to switch from any of the installed extensions.

We forecast that a complete software project will define a number of CADSEs, and that the number of meaningful combinations will be very high. In our approach, there is not any need to predefine these combinations (as well as in Eclipse, there is not need to define all the plug-in combinations).
Figure 6. Model and tool extension approach.

Note that defining an extension is often more difficult than designing the complete CADSE but it allows composing in many different ways CADSE extensions made by different experts of different company.

Figure 7. CADSE extension selection

Figure 7 is a screen shot of our tool when the user is prompted to make its CADSE selection. In the figure he/she selected the service core, development and test CADSEs. Note that it was not needed to explicitly select core, since selecting any one of its extensions will also import all the extensions it requires, including the core.

5.2 Hierarchical composition

Even if the user can freely compose the CADSEs of choice, some heavily used and consistent combinations can be provided in standard. This is easy since a CADSE composition is itself a CADSE. Importing a composite CADSE transparently imports all the inner ones; and it is possible to define an extension of a composite CADSE, not only the core one. This can be compared with the Eclipse strategy in which predefined compatible plug-in combinations are provided in standard under a single name (Europa, Ganime, etc.).

5.3 Evolution

CADSEg is itself a CADSE, whose domain of expertise is the definition of CADSEs and their extensions. It has two major consequences: (1) evolution, versioning and concurrent engineering at metamodel level is fully supported; (2) creating and updating a CADSE definition is for us a “standard” Software Engineering activity (under the CADSEg environment). Therefore, it is very easy to make changes to existing CADSEs definitions (adding attributes and links, making more helpful a wizard, improving a mapping or a builder and so on). Building a CADSE definition produces the corresponding plug-in. Then it is easy for users to install the new CADSE or their updates to get an up-to-date environment, with fixes and improvements that still work on the actual data; and without having to restart the tool.

This flexibility is to be compared with the actual generative tools, like GMF and others, in which any change in a metamodel generates a different set of tools incompatible with the previous tools and with the data they produced. Here, improvement and extensions can be smoothly integrated without disrupting the work under way, and at the moment more convenient for the developer.

Altogether, we provide Model Driven Development with the same level of flexibility and level of service which is currently available for Code Driven Development, like using Eclipse with Subversion.

6. Experience

Domain Specific environments are capable of providing a high level of support because they contain a large body of knowledge, know-how and tools which are only relevant in that domain. The product line community proved that this hypothesis is true, but at the cost of large, monolithic, inflexible and very expensive environments. We are convinced that this domain knowledge, know-how and tools can be, to a large extent, captured into explicit metamodels and models.

On this basis we have designed our first environments for Melusine in the late 90s [10]. We have learned many things: (1) it is true that domain specific development in practice requires a specific Software Engineering environment; (2) a Software Engineering environment is very demanding and involves very many aspects, and (3) building by hand a Software Engineering environment, even limited, is an enormous work. Mélusine and its dedicated environments have been used for years.

Following the MDE trend, in the early 2000, we made the hypothesis that from models and metamodels it should be possible to generate tools and environments supporting the Software Engineering tasks to be performed in the domain. Therefore, using the previous experience we developed CADSEg, a generator which provided the target environment model and metamodels,
generates a CADSE, which is the target environment and its associated tools. From that point, we have been able to develop a number of CADSEs for different purposes, different clients in different contexts.

In the mid 2000, we reached the point where new problems appeared. First we had to face the dilemma that a powerful CADSE has to be much focused; but conversely, we need and use a broad scope of expertise and we perform different intertwined activities. Second, we face a “strange” demand from the iPOJO and Schneider customers. iPOJO [11] is a framework for dynamic service oriented applications which runtime can be extended with handlers. But handlers are like types, and are developed and maintained by different persons. Using new handlers requires extending the iPOJO metamodel with new concepts. The same need appeared with the Schneider DoCoSoc CADSE [12], where classes of devices are sometimes added, and using these new classes requires changing the metamodel. These examples showed us how critical it is to provide extensibility features, at model AND metamodel levels.

Among the difficulty we faced is the fact that we were using some tools associated to eCore (EMF, GMF and do on), but these tools are not extensible. We realized that the technology currently available in MDD is far to provide the flexibility and the features currently found in “classic” engineering. The most missing feature is related to editor extensions, not really supported today. In particular, users want to use GMF editors and be able to extend them with a CADSE extension. For example, FOCAS [13] which is a CADSE for service orchestration and choreography relies on a GMF editor to edit process model. Adding security and distribution concerns in this environment is performed through annotations on the (GMF based) process model. It was not possible to do so using the process model editor; a new development was required to produce the extended process model editor.

7. Related works

Since our system is an extensible Model Driven Software Engineering environment, it can be compared from one side with the Software Engineering environments (IDES), and from the other side with models and metamodels composition.

Most of current IDE (Eclipse, NetBeans) can be easily extended by third party stakeholders. This facility explains their success, but the large number of contributors and very low granularity level of plug-ins (a property page is a plug-in), produced a plug-ins proliferation with serious management difficulties. The granularity of a CADSE is a task or a specific concern which makes sense for the user. Assembling a handful of tasks and concerns is easier than hundreds of plug-ins.

MetaEdit+ [6] follow the generative approach to produce domain specific environments with powerful model editors. Unfortunately, users work directly on the shared repository which doesn’t scale and prohibits concurrent engineering. It does not provide extensibility facilities.

GMF [14] is an Eclipse extension which allows generating model editors from a model. It also takes advantage of Eclipse extensibility by producing the editors in form of eclipse plug-ins but does not provide extensibility for its technology.

XMF (Xecutable Metamodelling Facility) [9] supports the composition and execution of models conforming to different XCore metamodels; using XSync which is the XMF action language.

Mylyn [15] is an Eclipse extension that facilitates multi-task activity by hiding unneeded information such as unused projects and by providing an easy way to switch from an activity to another one without restarting Eclipse. Mylyn attaches task information to artifacts to enhance software evolution management.

There are three major research trends in model/metamodel composition: Model Management, Aspect-oriented Modeling and Metamodelling.

Model Management is a topic born in the MDE context. This community is interested in the platforms manipulating and managing the models, focusing on the operators to be applied on models (e.g, compare, merge, match, weave, sew etc.) [16]. Several platforms have been developed, like AMMA [17], Rondo [18], EOL [19] and MOMENT [20]. These Model Management approaches do not support extensions.

Aspect Oriented Modeling [4] allows developers to separate non primary concerns such as security or fault tolerance and functional ones in a primary model and model aspects. A weaving mechanism produces the composite model. Since some CADSE extensions can be related to a concern, our work can be seen as an implementation of AOM vision for Software Engineering environments.

Some Metamodelling environments support model compositions, like XMF (Xecutable Metamodelling Facility) [3] or GME (Generic Modeling Environment). XMF supports composing and executing models conforming to different XCore metamodels, through synchronized mappings, written in XSync[9]. GME [5] is a MDE framework aiming at the generation of domain specific design tools. GME focuses on design only, the metamodels can be composed but the produced tools are regenerated, not extended; and concurrent engineering is not supported. The canonical scheme for model composition proposed in [17] uses a weaving model, on ATL (ATLAS Transformation Language) and AMW.
Modeling imposes then some conflicting requirements:
for concurrent modifications of shared parts of a model.

Our system proved to be able to dynamically compose, extend and update our models and metamodels, without model migration and without regenerating existing tools; which clearly improve significantly the extensions reusability and the flexibility since extensions and updates can be applied at any time during the execution. This was made possible because our metamodels (data models) are rather simple; they are structural only; and our models are conforming to a tiny set of statically defined metamodels: interaction, evolution, mapping, build, etc. It is the static knowledge of these model semantics that makes possible to dynamically extend and update them.

8. Conclusion and future work

Tackling the complexity and sheer size of current software applications requires specialization and expertise. It is no longer possible for a generalist developer to understand all aspects of a complex application through all of its life-cycle. Knowledgeable experts must work independently on different aspects of an application to collectively produce, deploy and operate a complex system. This trend has driven a renewed interest in domain specialization, as we realize the enormous gains of productivity that can be achieved by targeting the specific needs of an expert in a narrow area.

Recent advances in meta-modeling and generative approaches have enabled the semi-automatic production of Domain Specific Modeling Environments in a timely and economical way. It is now possible to tailor and adapt the modeling environment to better fit the needs of experts in a given domain. However, a crucial aspect of an Engineering Environment, often oversight, is that the ultimate goal of an expert is to collaborate to produce the global, integrated system, and as such, the specialized environment must support cooperative, concurrent engineering. One naïve approach to handle concurrent engineering is to simply treat model as any other textual artifact (file), and manage it with the existing source code management systems, loosing the structured and semantically rich nature of models.

Another common alternative is to use a centralized multi-user repository to store models, using the traditional locking mechanisms, which defeats the need for concurrent modifications of shared parts of a model.

Concurrent engineering for Domain Specific Modeling imposes then some conflicting requirements:
from the one hand, we would like to have a consistent, global view of the complete system, with fine grained versioning of model elements and traceability links among the related parts of the model; on the other hand, we would like to support experts, providing them with environments closely tailored to the task at hand and shielded from the global complexity.

Traditional Code Driven Software Engineering, after decades of experience, conciliates these conflicting needs with a very simple paradigm. The user works in an isolated workspace containing only the objects and concepts needed for the task to perform; and the workspace is synchronized through the import/commit actions with a global repository containing all the shared artifacts and their versions. The IDE works on the artifacts present in the workspace, and the Configuration Management system supports artifact evolution and repository coherence. Our work borrows to CDE this simple and successful metaphor (private specific workspace and shared global repository), and leverages it to Model Driven Domain Specific Environments. However, this is far from trivial, since it has profound impacts on the way a domain is designed.

A common scenario, that motivated the work presented in this paper, is the seamless integration of different phases of the life-cycle of a domain specific application. This led us to define a CADSE not as a monolithic environment, but as a federation of task or concern specific environments. We have shown that a domain specific environment can be built dynamically, at user will, combining freely the specialized CADSEs of interest. Our main contribution is the definition of the CADSE extension mechanism, which is capable of extending the domain meta-model (the structural part) and the models expressing the engineering environment semantics, without model and data migration and without tool re generation. It is not enough to modularize and extend the engineering environment itself; we also need to modularize and extend the (generated) tools used by the environment.

A major contribution of our approach is the generalization to Model Driven Engineering of the traditional workspace paradigm. That generalization is much more difficult than it appears; among other things, it required us to define a specific evolution model which defines the versioning semantics to be applied on model elements at commit time.

These two mechanisms together (CADSE extensions and model driven workspaces) solve the issues and conflicts identified so far. From the domain expert point of view, it allows to incrementally define the different aspects (tasks or concerns) relevant in the domain. From developer’s point of view, it allows to freely and dynamically constitute the desired working environment simply importing extensions. From management point of
view, it allows evolution, adding new extensions, but also refining (or fixing) those extensions already developed. Users are free to get the updates when found convenient.

We believe that our work is an important step toward the maturity of the Model Driven Engineering approach.

CADSEg can be downloaded from [http://cadse.imag.fr](http://cadse.imag.fr).

References


