Modeling Multigrid Algorithms for Variational Imaging

Isabel Dietrich, Reinhard German
Computer Networks and Communication Systems
University of Erlangen
Martensstr. 3, 91058 Erlangen, Germany
{isabel.dietrich,german}@informatik.uni-erlangen.de

Harald Koestler, Ulrich Ruede
Chair for System Simulation
University of Erlangen
Cauerstraße 6, 91058 Erlangen, Germany
{harald.koestler,ulrich.ruede}@informatik.uni-erlangen.de

***Abstract*—UML-based modeling is becoming increasingly popular in many software development projects. One of the key aspects is the possibility to support automatic code generation from UML models while keeping the easy to use modeling abstraction for the software developer. The framework *Syntony* has been developed to generate discrete-event simulations from standard-compliant UML models in order to support simulation based performance evaluation of systems. In this work, we discuss the extension of *Syntony* to include automatic code generation in the context of large scale continuous simulations that require the numerical solution of partial differential equations (PDE). We choose variational imaging as an example field, and multigrid as numerical solver. Multigrid algorithms exhibit a fixed sequential structure, where the single steps are problem dependent. Typically, they are implemented in C++, and may depend on special hardware since most of their applications require the solution of large numerical systems and therefore high computational performance. Using *Syntony*, we provide a modeling framework that can be extended to cover new applications by providing the basic modules and data structures in C++ and modeling the high-level algorithms and classes in UML class and activity diagrams. We evaluate the applicability of our approach in a case study for image denoising. The generated code is a fully working application that computes a denoised output image from a given input image using the methods specified in the UML model. The key benefit lies in the abstraction from low level programming for building complex denoising algorithms. In addition, we show that the code generation and compilation process runs significantly faster than the compilation of the entire framework. We also show that the run-time overhead introduced by the generated code is neglible.**

***Keywords*-UML modeling, multigrid algorithms, Syntony, variational image processing**

I. Introduction

Model-based software development has become a major trend in software engineering. In order to support automatic code generation for computationally intensive applications, we envision an approach where we model all high-level structures and algorithms in an application using standard compliant UML models, and combine this high-level model with automatic code generation and inclusion of existing low-level code.

We apply our approach to a variational imaging framework with applications in medical image processing and computer vision [1]. The framework is based on different kinds of efficient multigrid solvers, runs in parallel, and some parts also support non-standard architectures like Cell processors or GPUs. The main task of the framework is to provide a platform for researchers and developers of new image processing algorithms. This framework offers high computational performance and can easily be extended to support different architectures, and to extract parts of the framework into stand-alone code. The last aspect is useful if, for example, external users or developers need only one application from the framework. It results in smaller and better readable code, since mechanisms like templates are not necessary, and shorter compile times. By now, the framework has grown far beyond 100.000 lines of code. The code makes heavy use of templates and ifdef constructs to support the various applications and architectures. Of course, this impedes the usability goals we require.

We aim to provide both a better usability combined with abstractions based on standard compliant UML models to overcome these shortcomings. Our approach should help to deal with the framework's complexity while keeping the original performance.

In principle, for the creation of high-level models, we have to choose either a general-purpose or a domain-specific modeling language. In a domain-specific language (DSL) [2], like, for example, Matlab[1], the language contructs can be fitted to the problem domain and domain experts can easily understand and develop domain-specific language programs. A DSL also allows for an easy validation at domain level. Nevertheless, a DSL has some disadvantages. First, every user has to learn a new language. Second, it has only limited applicability. Finally, it is a huge effort to design, implement, and maintain a DSL. UML [3], however, is a standardized general-purpose modeling language. Therefore, knowledge about the language constructs is already widely spread. In addition, tools and editors for UML are readily available. Therefore, we choose UML for the high-level modeling of our framework.

In this paper, we show how our approach can be used to restructure and to improve the software development process for numerical algorithms by combining high-level

[1] http://www.mathworks.com/products/matlab/

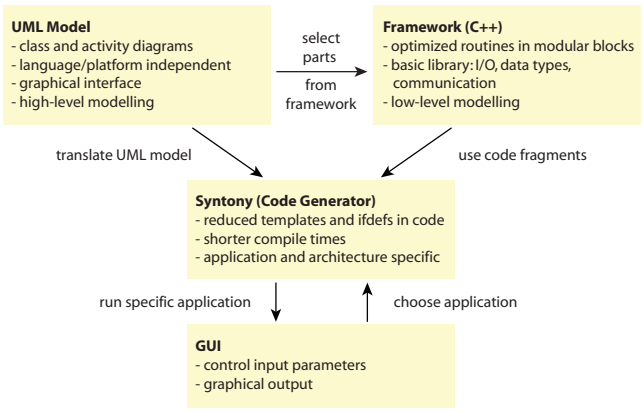


Figure 1: Overview of the improved software development process realized in our approach

modeling with automatic code generation and inclusion of existing low-level code. For the code generation, we rely on our framework *Syntony* [4]. *Syntony* is able to translate standard-compliant UML models to executable discrete-event simulations. We expand *Syntony*'s modeling paradigm and translation functionality to fully automatically generate working application code for parts of our image processing framework.

Figure 1 shows how the improved software development process works in detail. In the first step, we model the classes and high-level algorithms from the framework in UML. We use activity diagrams to model the algorithms. Each image processing application selects a subset from the model, i.e. only those parts of the model that are relevant for the application. Then, *Syntony* translates this subset of the UML model into C++ code. During the translation, code is generated from class or activity diagrams, and optimized C++ routines for the low-level modules are directly incorporated from the framework. Code from the framework that is not necessary to run a specific application is not found in the generated code. This results in shorter compile times and reduces templates and ifdef constructs. Finally, the generated code runs on a command line or within a graphical user interface that allows to set input parameters and visualizes the resulting images. This software development process has several additional advantages besides enabling graphical high-level code design. It forces the framework code to be modular with small building blocks, avoids loosing computational performance through the automatic code generation, and keeps the flexibility to work also directly in generated C++ code.

The remainder of the paper is structured as follows. We introduce both the image processing framework and the *Syntony* framework in more detail in Section II. The modeling paradigm used to represent image denoising algorithms in UML is introduced in Section III. In Section IV, we present the extended automatic transformation process which enables *Syntony* to cope with the new models. In Section V, we show some application examples and present a detailed performance evaluation of the modeled algorithms. Finally, Section VI concludes the paper.

II. Software Packages

In this section, we briefly introduce the two existing frameworks that we deal with in our work. The first one is the image processing framework already discussed above [1]. The second one, *Syntony* [4], supports code generation from standard compliant UML models.

A. Variational Imaging Framework

Image processing and computer vision are large fields with many applications, e.g. in robotics, cars, or medicine (see. e.g. [5]).

1) Variational Approaches in Imaging: Variational approaches are a common way of designing models for imaging problems based on minimizing some kind of energy functional that usually consists of two terms. On the one hand the data term provides (local) information about the problem, on the other hand the regularizer provides global information and ensures certain properties of the solution like uniqueness. Mathematically, one tries to find a mapping $\boldsymbol{u} : \mathbb{R}^d \mapsto \mathbb{R}^m, d, m \in \mathbb{N}$, where d is the problem dimension and $m = 1$ for the examples throughout this paper, such that the energy functional

$$\mathcal{E}[\boldsymbol{u}] := \underbrace{\mathcal{D}[J, \boldsymbol{u}]}_{\text{Data term}} + \alpha \underbrace{\mathcal{S}[\boldsymbol{u}]}_{\text{Regularizer}} \tag{1}$$

is minimized. $\alpha > 0$ is a weighting parameter and J denotes a d-dimensional image. Since the energy functionals are often given as an integral expression over the image domain Ω, the solution

$$\boldsymbol{u}^*(\boldsymbol{x}) = \text{argmin}_{\boldsymbol{u} \in H} \mathcal{E}[\boldsymbol{u}](\boldsymbol{x}) = \text{argmin}_{\boldsymbol{u} \in H} \int_\Omega \mathcal{L} d\boldsymbol{x} \tag{2}$$

of (1) in some suitable functional space H (e.g. a Sobolev space) fulfills a PDE, the so-called Euler-Lagrange equations

$$\mathcal{L}_{u_i} - \sum_{p=1}^{m} \partial_{x_p} \mathcal{L}_{(u_i)_{x_p}} = 0 \ , i = 1, \ldots, d. \tag{3}$$

2) Application: Image Denoising: Our imaging framework incorporates many different energy functionals and PDEs e.g. to model image denoising, image restoration, image segmentation, optical flow, or image registration. In this paper, we choose two methods for image denoising as an example.

Image denoising removes noise from a given, noisy image [6], [7], [8], [9], [10]. Usually images contain a certain amount of noise dependent on the image acquisition system. Since the kind of noise for a given real image is usually unknown we assume that the relation between an original,

unknown image $u : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}$ and an observed image u^0 can be expressed by

$$u^0 = u + \eta. \tag{4}$$

Here, η stands for the noise, where we restrict ourselves to a white additive Gaussian noise for simplicity.

Denoising by Homogeneous Diffusion: Assumption (4) provides us with local information and thus can be used as data term. The simplest variational approach for image denoising is based on Tikhonov regularization [11] with a linear homogeneous diffusion regularizer. Hence, we try to minimize the energy functional

$$\mathcal{E}[u] = \int_\Omega |u^0 - u|^2 + \alpha |\nabla u|^2 d\boldsymbol{x} \tag{5}$$

with $\boldsymbol{x} \in \mathbb{R}^d$ and regularization parameter $\alpha \in \mathbb{R}^+$ in the image domain $\Omega \subset \mathbb{R}^d$. A necessary condition for a minimizer $u : \Omega \mapsto \mathbb{R}$, the denoised image, is characterized by the Euler-Lagrange equations

$$-\alpha \Delta u + u = u^0 \quad \text{in } \Omega \tag{6a}$$

$$\langle \nabla u, \boldsymbol{n} \rangle = 0 \quad \text{on } \partial\Omega \tag{6b}$$

with homogeneous Neumann boundary conditions at the image boundary $\partial\Omega$.

Denoising by Complex Diffusion: One disadvantage of the homogeneous diffusion process is that is blurs image edges. Therefore, we consider a more advanced nonlinear isotropic complex diffusion process [12], [13] as an alternative. It keeps image edges by reducing the strength of the diffusion at high gradients in the solution. Here, the diffusion process is modeled by a time-dependent PDE

$$\operatorname{div}\left(g\left(Im(u)\right)\nabla u\right) = u_t \tag{7}$$

with Neumann boundary conditions, initial condition $u(0) = u^0$ and time $t = 0$ which is motivated by a simplified time-dependent Schroedinger equation. $Im(u)$ denotes the imaginary part of u and the complex diffusivity function is given by

$$g\left(Im(u)\right) = \frac{e^{i\theta}}{1 + \left(\frac{Im(u)}{k\theta}\right)^2}. \tag{8}$$

with small angle θ and scaling parameter $k > 0$.

For these (non)linear partial differential equations, analytical solutions are most often not available. Therefore, we first have to discretize them in order to compute an approximate (numerical) solution. We do this by finite volumes in space and by an implicit Euler scheme in time [14]. The implicit method enables us to use only one (discrete) time step since it keeps stable also for arbitrarily large time step sizes. Then, we solve the arising (non)linear system of equations

$$A^h u^h = f^h\,, \quad \sum_{j \in \Omega^h} a_{ij}^h u_j^h = f_i^h\,, i \in \Omega^h \tag{9}$$

numerically in order to obtain an approximate solution to its exact solution u^* from eq. (2). Here, we have the system matrix $A^h \in \mathbb{R}^{N \times N}$, unknown vector $u^h \in \mathbb{R}^N$ and right hand side (RHS) vector $f^h \in \mathbb{R}^N$ on a discrete grid Ω^h with mesh size $h = \frac{1}{2^N}$. $N \in \mathbb{N}$ denotes the number of unknowns. The number of unknowns N may be quite large for big images (e.g. about 4 million for a 2048^2 image) and remember that we consider PDEs that are inherently based on some kind of diffusion process. This leads to a large, sparse, and positive definite system matrix A^h that allows us to use an efficient iterative multigrid solver to compute a numerical solution for u^h.

3) Multigrid Basics: For good introductions and a comprehensive overview on multigrid methods, we refer to [15], [14]. Multigrid is not a single algorithm but a general approach to find a solution by using information from several scales or levels.

The multigrid idea is based on two principles:

Smoothing Property: Classical iterative methods like Gauß-Seidel are able to smooth the error $e = u^h - u^*$ after very few steps. That means the high frequency components of the error are removed well by these methods. But they have little effect on the low frequency components. Therefore, the convergence rate of classical iterative methods is good in the first few steps and decreases considerably afterwards.

Coarse Grid Principle: It is well known that a smooth function on a fine grid can be approximated satisfactorily on a grid with less discretization points (we choose a coarse mesh size $H = 2h$), whereas oscillating functions would disappear. Furthermore, a procedure on a coarse grid is less expensive than on a fine grid. The idea is now to approximate the low frequency error components on a coarse grid.

The usual multigrid efficiency is achieved through the combination of two iterations, the *smoother*, or *relaxation*, and the *coarse grid correction*. This leads to recursive algorithms which traverse between fine and coarse grids in the grid hierarchy. Since ultimately only a small number of relaxation steps independent of the number of unknowns must be performed on each level, multigrid can reach an asymptotically linear complexity $\mathcal{O}(N)$ [14]. In order to derive the multigrid algorithm let an approximate solution u^h computed by a few smoothing iterations be given. Then, the relationship between the algebraic error e^h and the residual $r^h = A^h u^h$ can be expressed by the *error equation*. It reads

$$A^h e^h = r^h. \tag{10}$$

This equation is approximated (recursively) on coarser grids within the multigrid method and leads to the so-called multigrid V-cycle listed in algorithm 1.

Combined with a traversion from the coarsest grid to the finest grid this results in a full multigrid (FMG) algorithm visualized in Figure 2. Each of the steps is denoted again in the following:

Algorithm 1 MG correction scheme (V-cycle): Compute $u_h^{(k+1)} = M_h(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2)$

1: $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}(u_h^{(k)}, A^h, f^h)$ {pre-smoothing}
2: $r^h = f^h - A^h \bar{u}_h^{(k)}$ {compute residual}
3: $r^H = \mathcal{I}_h^H r^h$ {restrict residual}
4: **if** number of coarse grid points $< \epsilon_{min}$ **then**
5: solve $A^H e^H = r^H$ exactly
6: **else**
7: $e^H = M_H(0, A^H, r^H, \nu_1, \nu_2)$
8: **end if**
9: $e^h = \mathcal{I}_H^h e^H$ {interpolate error}
10: $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ {coarse grid correction}
11: $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}(\tilde{u}_h^{(k)}, A^h, f^h)$ {post-smoothing}

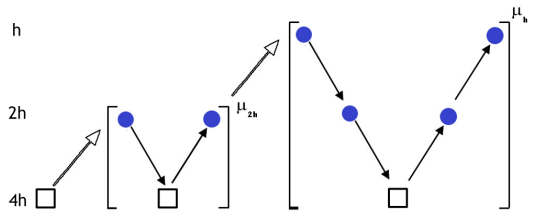


Figure 2: FMG algorithm with V-cycles and mesh size h

- Exact solution: □, Interpolation of solution: ↗
- Smoothing: ●, Restriction of Residual ($f - Au$): ↘
- Interpolation of the error and Correction of the solution: ↗
- $\mu_h = 1$ results in a V-cycle.

In order to treat nonlinear PDEs typically two approaches are common in combination with multigrid methods. Either one performs a global linearization of the problem, usually by Newton's method or some of its inexact variants and applies a linear multigrid solver to the resulting linear system, or one adapts the MG correction scheme resulting in the *Full Approximation Scheme (FAS)* [16], [17]. Here, only local linearization is performed within the smoothing process.

The error equation (10) in the nonlinear case becomes

$$A^h(u^h + e^h) = r^h + A^h(u^h). \tag{11}$$

Approximating eq. (11) on the coarse grid means that not an error, but a coarse solution of the nonlinear problem is computed there. The FAS scheme is shown in algorithm 2.

The smoothing steps in the nonlinear case are based on an inexact lagged-diffusivity method, i.e. we just fix the nonlinear coefficients during a Gauss-Seidel iteration and update them afterwards.

B. Syntony

Syntony [4] is an Eclipse-based model transformation framework. Its main application focus lies within discrete-

Algorithm 2 FAS scheme (V-cycle): Compute $u_h^{(k+1)} = M_h^{FAS}\left(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2\right)$

1: $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}\left(u_h^{(k)}, A^h, f^h\right)$ {pre-smoothing}
2: $r^h = f^h - A^h\left(\bar{u}_h^{(k)}\right)$
3: $r^H = \mathcal{I}_h^H r^h$
4: $\bar{u}^H = \tilde{\mathcal{I}}_h^H \bar{u}_h^{(k)}$
5: $f^H = r^H + A^H\left(\bar{u}^H\right)$
6: **if** number of coarse grid points $< \epsilon_{min}$ **then**
7: solve nonlinear problem $A^H(w^H) = f^H$ exactly
8: **else**
9: $w^H = M_H^{FAS}\left(\bar{u}^H, A^H, f^H, \nu_1, \nu_2\right)$
10: **end if**
11: $e^H = w^H - \bar{u}^H$
12: $e^h = \mathcal{I}_H^h e^H$
13: $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ {coarse grid correction}
14: $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}\left(\tilde{u}_h^{(k)}, A^h, f^h\right)$ {post-smoothing}

event simulation, however, as we show in this paper, it can be extended to cover continuous simulation. *Syntony* uses standard compliant UML models as input. From the variety of UML diagrams, *Syntony* supports those that are particularly suitable to model the structure and behavior of an entire system in detail, namely class, composite structure, state machine and activity diagrams.

The modeling paradigm used in *Syntony* is that of communicating automata. The system's building blocks are described in class diagrams. Composite structure diagrams are used to model the inner structure of these building blocks, i.e. how the building blocks are nested and connected with each other. The high-level behavior of each building block is described in a state machine diagram. Activity diagrams may be used to specify the detailed, low-level behavior. Finally, performance attributes and measures are annotated using the standardized profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE).

Syntony can automatically generate discrete-event simulations for system models following this paradigm. When a user has modeled a system in this way, *Syntony* performs the remaining tasks completely automatically. First, it transforms the input model into executable C++ code for the simulation software OMNeT++. Then, the code is compiled and executed. Finally, the user may analyze the simulation results to gain insight into the performance of the modeled system.

Independent from the application to discrete-event simulation, *Syntony* contains the algorithms needed to transform UML models to generic C++ code. This code is ready to be compiled and run automatically. *Syntony* therefore provides a common basis for code generation from UML models, provided the models consist of the supported diagram types.

This functionality is exactly the basis we rely on in this paper: we use the existing algorithms and adapt them to cover applications in continuous simulation.

In addition, *Syntony* comes with an extensible graphical user interface which can easily be adapted to fit the needs of the different image processing applications.

III. UML Modeling

To support code generation for numerical algorithms, we extended the original modeling paradigm used in *Syntony* as well as the transformation algorithms implemented in *Syntony*. In this section, we describe the modeling approach we use to represent multigrid algorithms in UML.

First of all, the variational imaging framework consists of a number of different components or modules. We categorize them into basic, solver, and application modules and distinguish whether they are common to many or all applications, or specific to some applications. Table I provides examples for each of the possible module combinations. Note that especially the specific solver modules are computationally intensive and therefore good candidates to make them specific for certain hardware.

Table I: Examples of different combinations of module category and module type.

Module category Module type	Basic	Solver	Application
Common	IO	FMG algorithm	Program flow
Specific	Data types	Smoother	Diffusivity function

Our approach to represent imaging applications in UML basically consists of two aspects. First, we model the numerical algorithms, i.e. all modules specifying control flows, like the common solver modules, in activity diagrams. Activity diagrams are the most suitable UML diagram type to represent the method calls and control flows in a typical algorithm. Second, we copy the existing specific solver modules. To achieve this, it is necessary to reflect the class hierarchy of the entire variational imaging framework in UML. Fortunately, this is not as time-consuming as it sounds because several UML editors allow to automatically extract class hierarchies from programming language code. Using the modeled class hierarchy, we specify the C++ sources for all classes and methods that should be copied without modification from the framework. We also specify which methods are represented by activity diagrams.

Of course, this approach requires that the existing code is designed modularly and that its classes have suitable interfaces. If the code does not satisfy this requirement, restructuring may be necessary. However, from a software engineering perspective, this will positively influence the reusability, readability and maintainability and can therefore be considered a desirable side-effect.

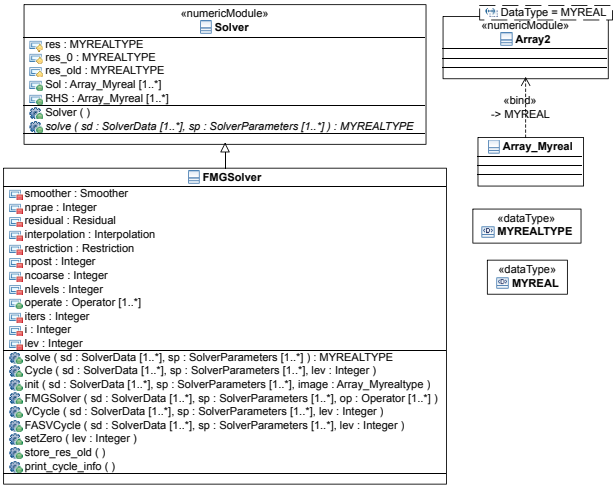


Figure 3: Class hierarchy for the FMGSolver

To further clarify our modeling approach, we explain more details using some of the diagrams from our multigrid model in the following sections.

A. Classes, Data Types, Templates

Figure 3 shows a part of the framework's class diagram containing four classes and two data types modeling the FMG algorithm from figure 2. All of the information visible in the diagram is also used in the transformation process. The classes may contain most of the features specified in the UML standard.

The class *Solver* at the top of the diagram contains several properties with protected (yellow, diamond-shaped icon) and public (green circular icon) visibility. Its two public properties, *Sol* and *RHS*, respresenting the solution u^h and the RHS f^h from eq. 9 are both of type *Array_Myreal*. They may have an unlimited number of values, but must contain at least one, i.e. the number of unknowns $N \geq 1$. The other properties must have exactly one value. The class also contains two operations. The operation printed in italics is abstract. *Solver* serves as a base class for *FMGSolver*. The figure also shows the class *Array2* which defines a template parameter called *DataType*, and the class *Array_Myreal* which binds this template parameter to the concrete type *MYREAL*. The definition of the data type *MYREAL* is also shown in the diagram. In a concrete application *Array2* can be for example a 2D float array.

B. Inclusion of Existing Modules

To specify which modules and methods should be copied from the existing code base, we make use of a custom UML profile. Profiles are a mechanism described in the UML standard that provides the possibility to extend the semantics of arbitrary UML elements. To achieve this, a profile may consist of several stereotypes. Similar to a class,

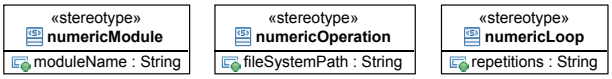


Figure 4: Profile for the inclusion of existing modules

a stereotype may have attributes. In addition, a stereotype specifies which elements it may be applied to. The semantics of these stereotypes may be freely defined by the user. For our purpose, we developed a small profile consisting of three stereotypes. Figure 4 shows these stereotypes in a class diagram. The first stereotype, «numericModule», may be applied to UML classes. Its meaning is that code for the corresponding class should not be generated, but instead taken from an existing C++ source file. The name of the C++ source file is given in the *moduleName* attribute of the «numericModule» stereotype. The second stereotype, «numericOperation», may be applied to UML operations. It means that the implementation of the corresponding operation should be taken from existing C++ sources. Like the «numericModule» stereotype, «numericOperation» also has an attribute, *fileSystemPath* to specify the name of the source file. The third stereotype, «numericLoop», can be applied to all UML actions. It specifies that the action it is applied to should be executed in a loop. The number of repetitions is given in its *repetitions* attribute.

In figure 3, the stereotype «numericModule» is applied to the *Solver* class. So, the base class for the specialized solvers in the model (and its data structures) are not specified in the UML model, but rather taken from an existing module. In contrast, the *FMGSolver* class, including the operations contained in it, is generated entirely from the UML model.

C. Activities

As mentioned above, we use UML activities to model algorithms on a detailed level. A UML activity consists of nodes and edges. The nodes can be further classified into control nodes, object nodes, actions, and groups. The edges may be either control flows or object flows, depending on whether control tokens or objects travel along the edge.

To illustrate the basic concepts of UML activities, we show the activity diagrams for two algorithms from the image processing framework in Figures 5 and 6 representing the behavior of the *solve()* method in the FMGSolver class and the V-cycle presented in algorithm 1, respectively. The activity has a well-defined starting point, the initial node. It also has an activity final node which terminates the activity execution. Strictly speaking, these nodes are not necessary because the execution of an activity starts at any node without incoming edges and stops if there are no more outgoing edges. However, the nodes provide useful visual orientation for the user. At the top of the diagram, there are two activity parameter nodes called *sd* and *sp* which provide input values for the activity. Activity parameter nodes may also be used to specify return values, like the node *ret* at the bottom of the diagram. The main part of the diagram shows two nested structured activity nodes called *outer loop* and *inner loop*. These nodes may contain other activity nodes and edges and can be used to structure the diagram. Several actions are contained in the activity. Actions are represented by rounded rectangles. Their types may be distinguished using the small icon next to their name. In this diagram, there are opaque actions (*init lev* and *decrement lev*), several call operation actions (e.g., *cycle*), and a read structural feature action (*read res*).

Actions may have input and output pins denoting the flow of objects to and from actions. They are represented using small squares that are attached to the action. For example, the *residual* action in the inner loop has one output pin called *res*. The value on this pin is used as input to the next action, *save res*. All edges that are connected to pins are called object flows because only objects may travel along these edges. The other edges are called control flows. Control tokens travel exclusively along control flows.

In summary, the *Solve_act* activity does the following: First, the *init lev* action is executed. Then, control flow passes to the *outer loop*. According to the number of repetitions specified in the «numericLoop» stereotype, the *outer loop* is executed several times. It first executes the *decrement lev* action, then all nodes within the *inner loop*. Again, the *inner loop* is executed several times according to the number of repetitions given in its «numericLoop» stereotype. When the *inner loop* has finished execution, control is passed to the decision node. Depending on the guards on its outgoing edges, the decision node passes control either to the *interpolate* action, or stops execution of the *outer loop*. Finally, after the *read res* action has passed its return value to the activity parameter node *ret*, execution of the activity finishes.

IV. Model Transformation

In the following, we explain how the elements from valid input models described in the previous section are transformed to C++ code. The transformation is controlled from a graphical user interface that is realized as a plug-in to the Eclipse platform. It relies on the Eclipse UML2[2] plug-in to access the UML model elements.

Basically, the transformation process walks through the input model and generates C++ code for the supported elements depending on their type. Nested elements, such as attributes or operations, are translated in the context of their owner. For example, the translator for classes takes care to translate all properties and operations owned by a class. In turn, the translator for operations translates the operation's method, for example an activity. The translator for activities

[2]http://www.eclipse.org/uml2/

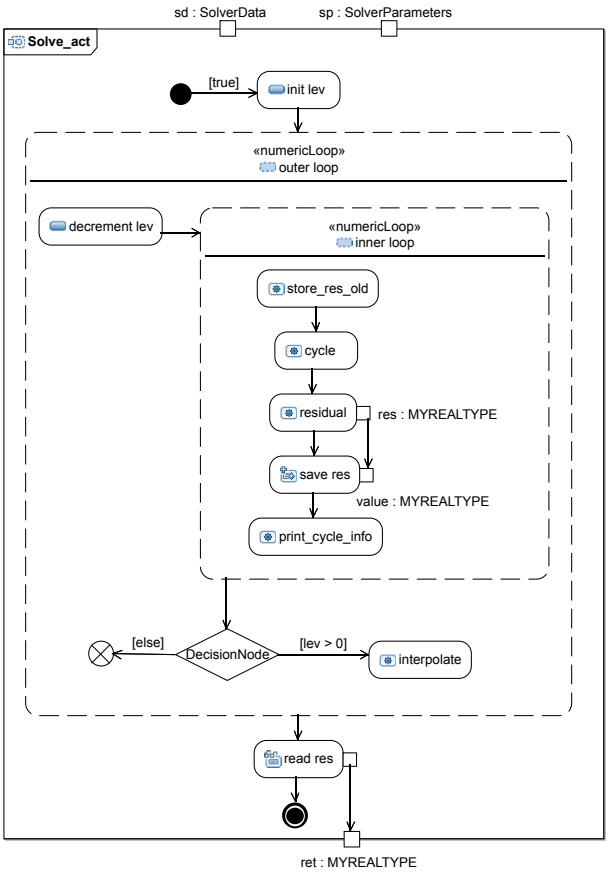


Figure 5: Activity diagram for the solve method

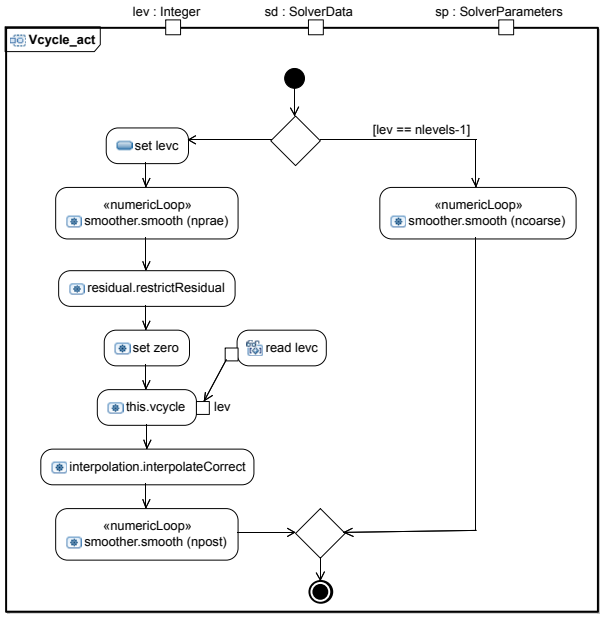


Figure 6: Activity diagram for the Vcycle method representing the V-cycle in a multigrid algorithm

then translates all actions and flows contained in the activity. In the following sections, we present the algorithms for the various model elements, i.e. classes, data types, templates, and activities.

After the transformation, *Syntony* invokes the C++ code formatter, which is available from the CDT[3] plug-in, on the generated files. It also creates a Makefile for the generated files and compiles the code without further user interaction. Errors that occur during the compilation process are communicated to the user via the graphical interface.

A. Classes

For the translation of UML classes to C++ code, we have to differentiate two cases: classes stereotyped as «numericModule», and regular classes.

For the first case, the existing C++ code (both header and implementation files) is simply copied from the file system. The name of the copied files may either be identical to the UML class name, or be specified in the stereotype's *moduleName* tag.

For each regular UML class, *Syntony* creates a C++ class with a sanitized name, i.e. stripped of white space and other characters that are allowed in UML, but not in C++. The details of the class transformation process are given in Algorithm 3. If a property or operation parameter has a multiplicity other than [1..1], the element is represented as a fixed-size array if the upper bound is a fixed value, or as a pointer if the upper bound is unlimited.

B. Data Types

UML data types are represented in C++ as simple typedef statements. The desired C++ type is specified in the data type's *type* attribute. The typedef statement is printed in a C++ header file named according to the data type's sanitized name.

C. Templates

UML allows to represent both classes that have template parameters and classes that bind template parameters to concrete types. In our models, classes that have template parameters are always stereotyped as «numericModule» (like the *Array2* class in Figure 3). Therefore, the transformation process does not have to care about these classes. Classes binding template parameters are translated to typedef statements. As an example, take the *Array_Myreal* class from Figure 3. It binds the template parameter from its base class (*Array2*) to the concrete data type *MYREAL*. The generated typedef statement for this class is

```
typedef Array2<MYREAL> Array_Myreal
```

[3]http://www.eclipse.org/cdt/

Algorithm 3 Transformation: Class

```
Create C++ class with sanitized name
Add class inheritance
Create constructor
for all Properties do
  Create C++ attribute with correct visibility, type, and multiplicity
  if Property has default value then
    Initialize property in constructor
  end if
end for
for all Operations do
  if Operation has «numericModule» stereotype then
    if Operation is used in the input model then
      Copy operation body from file system
    end if
  else
    Create C++ method with correct visibility and return type
    for all Operation parameters do
      Add method parameter with correct type and multiplicity
    end for
    if Operation has template parameters then
      Add template definition to the operation
    end if
    if Operation is abstract then
      Make operation virtual
    end if
    if Operation method is present then
      Create operation body (i.e. invoke activity translator)
    end if
  end if
end for
```

D. Activities

The semantics of activity execution can be summarized as follows (for more details, see the Unified Modeling Language (UML) specification [3], Section 12.3.4). First, all nodes that are currently executable are gathered. A node is executable if it has received control tokens from all its predecessors connected via control flows, and objects from all its predecessors connected via object flows. For example, the *save res* action in Figure 5 is executable if it has received a control token from the *cycle* action as well as an object on its input pin. Then, one of the executable nodes is chosen for execution. The order in which executable nodes are chosen is not specified. The specific actions that take place when a node is executed vary depending on the node type. After the node has been executed, control tokens are placed on its outgoing control flows, and objects are placed on its outgoing object flows. In the example, the *print_cycle_info* action would receive a control token after the *save res* action has completed its execution. This procedure is repeated until there are no more executable nodes, or until an activity final node is encountered.

Our translation of UML activities to C++ code works along the lines of this token-based semantics. For every node contained in an activity, *Syntony* creates a C++ class to handle the tokens and objects the node receives as well as its execution. Algorithm 4 shows in detail how the activity node classes are built. The *offer()* method may be called by predecessor nodes to offer control tokens or objects which are then saved in the corresponding class member. The *canExecute()* method may be called to find out if enough tokens or objects have been received from all predecessors, i.e. if the current node is executable. The *execute()* method controls the actual execution of a node. Details for this method are given in Algorithm 5.

Algorithm 4 Transformation: Create Activity Node class

```
for all Predecessors do
  Create a member pointing to the predecessor
  if Predecessor is connected with control flow then
    Create boolean member to record control token reception
  end if
  if Predecessor is connected with object flow then
    Create member to save received objects
    Set member type to expected object type
  end if
end for
for all Successors do
  Create a member pointing to the successor
end for
Create offer() method
Create canExecute() method
Create execute() method
```

Algorithm 5 Transformation: Create *execute()* method

```
Call node execution method depending on node type
for all Successors do
  Offer token or object to the successor
  if Successor is executable then
    insert successor into the list of executable nodes
  end if
end for
```

To control the execution of the entire activity, *Syntony* creates a separate class for the activity. For every activity node contained in the activity, the activity class contains one member. The class also contains a list that holds all activity nodes that are currently executable. Finally, the

class contains a *start* method to start the activity execution. Algorithm 6 gives details about the *start* method.

Algorithm 6 Transformation: Activity Class (*start()* method)

```
for all ActivityNodes do
  if Node has no predecessor then
    Offer a control token to the node
  end if
  if The node is currently executable then
    Insert the node into the list of executable actions
  end if
end for
while List of executable actions is not empty do
  Call execute() for the first node in the list
  Remove the node from the list
end while
if The activity has a return parameter then
  Return the last object that was offered to the return
  parameter node
end if
```

V. Results

To evaluate the applicability of our approach, we modeled the two image denoising approaches discussed in Section II in UML. For homogeneous diffusion, we have to solve eq. (6) by algorithm 1 and for complex diffusion we solve eq. (7) by algorithm 2.

In the following, we evaluate code generation and compile times, as well as the run-time overhead of the generated code. We also include a discussion of the advantages and disadvantages of our modeling approach.

A. Code generation

Only small changes in the UML model are necessary to switch between the homogeneous and complex diffusion. First, the computations have to be done with complex numbers instead of real ones for complex diffusion. This can be controlled by exchanging the basic data type *MYREAL* in the UML model. Second, the activity diagram for algorithm 1 has to be replaced by the activity diagram for algorithm 2. Finally, some function calls, for example for setting up the solver coefficients and computing the diffusivity function, have to be changed.

We measured the generation and compilation times taken by *Syntony* for the two algorithms on a Pentium IV with 3.4 GHz. The process takes about 15 seconds for homogeneous diffusion; complex diffusion takes slightly longer, see Table II. In comparison, the compile time for the original framework from [1] varies between one and two minutes depending on the compiler and settings of the code, e.g. if support for parallelization or 3D applications are included. Because of template constructs, compile times cannot be reduced much by using multiple cores in parallel.

Table II: Generation (with/without code formatter) and compile times for homogeneous (HD) or complex (CD) diffusion in seconds. The shown values are mean values over 10 single measurements.

	Generation w	Generation w/o	Compilation
HD	10.9 s	5.1 s	9.9 s
CD	10.4 s	4.9 s	11.3 s

B. Test Runs

The automatic code generation produces some overhead especially for handling the different paths in activity diagrams. But since the solver requires heavy computations which grow linearly with the image size, this overhead can be neglected (see table III).

Table III: Runtimes for denoising test images of different sizes with homogeneous (HD) or complex (CD) diffusion. In brackets we give the percentage of the runtime overhead for the generated code. The solver performs two FMG(2,2) cycles.

Method (overhead)	256×256	512×512	1024×1024
HD	0.03 s (2 %)	0.13 s (1 %)	0.44 s (0.5 %)
CD	0.1 s (1 %)	0.38 s (0.5 %)	1.5 s (0.2 %)

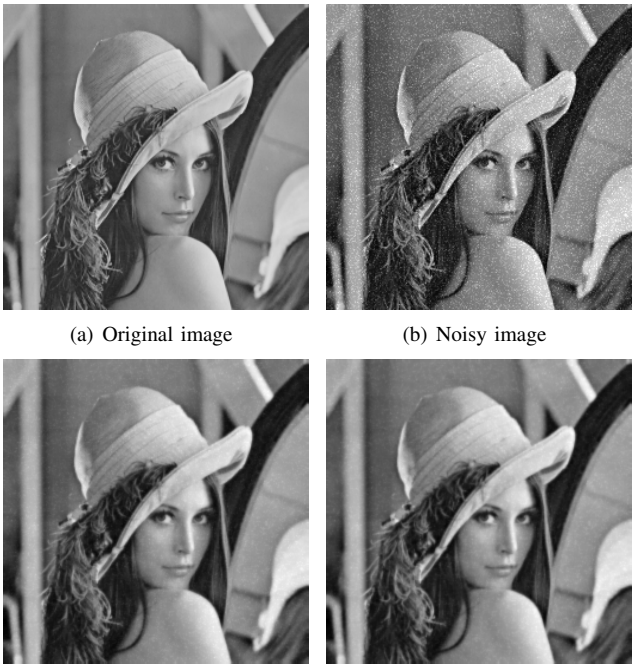
In Figure 7 and Figure 8, we show denoising results for homogeneous and complex diffusion. As parameters we set $\theta = 1$ and $k = 1$. The solver performs two FMG(2,2) cycles. In Figure 7, we added Gaussian noise ($\sigma = 10$) to the original image before denoising it. To quantify the quality of our methods, we compute the peak signal to noise ratio (PSNR) defined as

$$\text{PSNR} := 10 \lg \frac{J_{\max}^2}{\text{MSE}\left(u_{\text{orig}}, u_{\text{denoised}}\right)}, \qquad (12)$$

where $J_{\max}$ denotes the maximum gray value (typically 255) and MSE denotes the mean squared error between the original and the denoised image over all pixels. The PSNR does not take into account the nonlinear behavior of the human visual system and thus does not have to correlate with the perceived visual quality. The more advanced complex diffusion exhibits a better PSNR, whereas homogeneous diffusion tends to blur image edges. As can be seen in Figure 8, the imaginary part of the denoised image by complex diffusion is a kind of smoothed second derivative or edge image [12] that can be useful for further image processing like segmentation.

C. Discussion

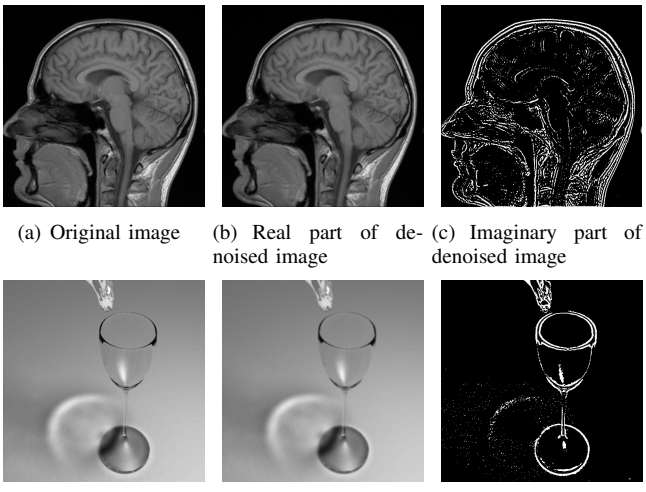
We have shown that our approach to model numerical algorithms with UML and automatically generating special code for different applications instead of using the original framework works not only in theory, but also in practice.



(a) Original image (b) Noisy image

(c) Homogeneous diffusion (d) Complex diffusion

Figure 7: Denoising results of an image of size 256×256 (upper left) with added Gaussian noise (upper right, $\sigma = 10$, PSNR = 25.5 dB) for homogeneous diffusion (lower left, PSNR = 26.3 dB, $\alpha = 0.5$) and complex diffusion (lower right, PSNR = 27.2 dB, $t = 0.5$)



(a) Original image (b) Real part of denoised image (c) Imaginary part of denoised image

(d) Original image (e) Real part of denoised image (f) Imaginary part of denoised image

Figure 8: Results for complex diffusion with $t = 1$ and image sizes (from top to bottom) 1024×1024, 512×512, and 400×400

However, what are the advantages and disadvantages of our approach? There are two main aspects we need to address: the properties of the generated code and the properties of the UML modeling process.

Concerning the properties of the generated code, a disadvantage of our approach is that the user does not have full control over the generated code. In addition, generated code can be quite complex and not easily readable especially for translated activity diagrams. We also did not achieve complete code generation, since the low-level modules still have to be implemented in C++. However, we have the advantage that the generated code is more compact than the original framework because only those parts of the framework are included in the code that are required for the selected application. This leads to shorter compile times of the generated code. In addition, the run-time overhead for the generated code is negligible.

The most important drawback concerning the UML modeling process is that our approach requires a UML editor that fully respects the UML standard. At the moment, these editors are rare, and their usability is somewhat lacking. However, once a suitable UML editor has been found, graphical modeling is much more intuitive than writing code by hand. As a result, new models and applications can easily be added to the framework also by non-experts in C++. In addition, UML models are reusable and independent of the target language and platform. Therefore, porting to different architectures or programming languages is simplified. Finally, it is possible to integrate UML editor, code generator, and graphical user interface into a single platform, for example Eclipse. Therefore, the user does not have to switch between different programs to use and extend the framework.

In summary, our approach brings us closer to reaching the goals stated in the introduction. The generated code does not have a negative effect on the required performance, while the UML modeling process allows us to efficiently extract parts of the framework and easily create new applications.

VI. Conclusion

In this paper, we propose an improved model-driven software development process for numerical algorithms. Our approach consists of two parts: First, the high-level system behavior is modeled using UML class and activity diagrams. Then, the modules which should be copied from the existing code base are specified in the UML model. Typically, all hardware-specific and computationally intensive modules will be included. Our tool *Syntony* is able to fully automatically generate working code from these UML models. In detail, the process consists of several steps:

1) Implementation of basic data types, low level algorithms, and I/O routines efficiently in C++ or even on specific hardware like GPUs

2) Design of classes in UML or auto-generate class model from existing framework
3) Design of high level algorithms in UML activity diagrams
4) Translation of UML model into C++ code (*Syntony*)
5) Compilation and execution of the code on a specific platform or from the GUI

Note that it is clear that this process is not suitable for a single application. Instead, it is meant to be used for a big collection of similar applications like present in the discussed framework.

Our next step is to include additional applications from the variational framework described in [1] into our UML model. We also intend to extend our approach to generate code for different hardware platforms, for example GPUs.

ACKNOWLEDGMENT

This research was funded in part by the German Federal Ministry of Education and Research under grant number 01IA08001C.

REFERENCES

[1] H. Köstler, *A Multigrid Framework for Variational Approaches in Medical Image Processing and Computer Vision*. Verlag Dr. Hut, München, 2008.

[2] M. Mernik and A. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[3] Object Management Group (OMG), "UML 2.2 Superstructure Specification," OMG, Tech. Rep., February 2009.

[4] I. Dietrich, V. Schmitt, F. Dressler, and R. German, "SYNTONY: Network Protocol Simulation based on Standard-conform UML 2 Models," in *2nd ACM International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2007): 1st ACM International Workshop on Network Simulation Tools (NSTools 2007)*. Nantes, France: ACM, October 2007.

[5] B. Jähne, *Digitale Bildverarbeitung*, 6th ed. Springer-Verlag, Berlin, Heidelberg, New York, 2006.

[6] A. Buades, B. Coll, and J. Morel, "A Review of Image Denoising Algorithms, with a New One," *Multiscale Modeling and Simulation*, vol. 4, no. 2, pp. 490–530, 2006.

[7] L. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," in *Proceedings of the eleventh annual international conference of the Center for Nonlinear Studies on Experimental mathematics : Computational issues in nonlinear science*. Amsterdam, The Netherlands: Elsevier Science Publishers, Amsterdam, The Netherlands, 1992, pp. 259–268.

[8] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 7, pp. 629–639, 1990.

[9] J. Weickert, "Theoretical foundations of anisotropic diffusion in image processing," *Computing*, vol. 11, pp. 221–236, 1996.

[10] G. Aubert and P. Kornprobst, *Mathematical Problems in Image Processing: Partial Differential Equations and the Calculus of Variations*, 2nd ed., ser. Applied Mathematical Sciences. Springer-Verlag, Berlin, Heidelberg, New York, 2006, vol. 147.

[11] A. Tikhonov and V. Arsenin, *Solution of ill-posed problems*. Winston and Sons, New York, NY, USA, 1977.

[12] G. Gilboa, N. Sochen, and Y. Zeevi, "Image enhancement and denoising by complex diffusion processes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 8, pp. 1020–1036, 2004.

[13] O. Honigman and Y. Zeevi, "Enhancement of Textured Images Using Complex Diffusion Incorporating Schroedinger's Potential," in *2006 IEEE International Conference on Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings*, vol. 2, 2006.

[14] U. Trottenberg, C. Oosterlee, and A. Schüller, *Multigrid*. Academic Press, San Diego, CA, USA, 2001.

[15] W. Briggs, V. Henson, and S. McCormick, *A Multigrid Tutorial*, 2nd ed. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2000.

[16] A. Brandt, "Multi-Level Adaptive Solutions to Boundary-Value Problems," *Mathematics of Computation*, vol. 31, no. 138, pp. 333–390, 1977.

[17] W. Hackbusch, *Multi-Grid Methods and Applications*. Springer-Verlag, Berlin, Heidelberg, New York, 1985.