Combining Model-driven and Schema-based Program Synthesis

Ewen Denney, Jon Whittle
QSS, NASA Ames Research Center,
Moffett Field, CA 94035, USA

{edenney,jonathw}@email.arc.nasa.gov

Abstract

We describe ongoing work which aims to extend the schema-based program synthesis paradigm with explicit models. In this context, schemas can be considered as model-to-model transformations. The combination of schemas with explicit models offers a number of advantages, namely, that building synthesis systems becomes much easier since the models can be used in verification and in adaptation of the synthesis systems. We illustrate our approach using an example from signal processing.

Keywords: model-driven architecture, program synthesis, UML, Kalman filters.

1. Introduction

Schema-based synthesis is a technique for automatically generating code from high-level behavioral specifications. The technique has been effectively applied for generating complete implementations in particular domains, for example, signal processing algorithms (AutoFilter [5]), and data analysis applications (AutoBayes [1]). A schema is usually defined as a generic representation of a family of applications. Synthesis then instantiates a number of schemas and combines them in a particular way. Schemas are a good way of representing domain-specific knowledge in a modular and high-level way. Schema-based synthesis has advantages over other forms of code generation in that schemas can be combined in many different ways thus leading to the ability to generate multiple implementations from the same specification. These implementations can be compared against metrics or non-functional requirements before a final choice of implementation is made.

The OMG's Model-Driven Architecture (MDA) [3, 4] advocates the development of systems by transforming platform independent models (PIMs) into platform-specific models (PSMs). From an MDA point of view, schemas can be considered as PIM-to-PSM transformations (from the domain-specific specification language to programming-language independent implementations). Many program synthesis systems do not maintain explicit models of the specification language or of the implementation. However, we do advocate such an approach, both for the advantages of modeling that it brings to program synthesis and for the transference of the advantages of synthesis to MDA. One of the key thrusts of MDA is the automation of model-to-model transformations. Schema-based synthesis can be seen as one way of automating the transformations.

Most approaches to MDA, however, define transformations in terms of rewrite rules which are applied to models to yield new models. In our approach, we propose that the schema (transformation) be defined in terms of an input and an output model. The input model defines a subset of applications in the domain that a schema can operate on. The output model defines the result of applying the schema. In addition, schemas must *instantiate* the output model to create specific artifacts which solve the input problem. Instantiation is not normally considered part of MDA, but is a crucial ingredient in program synthesis. This paper will show how to combine synthesis and modeling, or, put another way, how to include instantiation as part of a domain-specific MDA.

We feel that current approaches for defining transformations in MDA—e.g., those based on XML and XSLT—do not offer enough flexibility for instantiation. Synthesis is highly dependent on the specifics of the particular problem under consideration in a way that MDA is not. For example, different instantiations of models will be generated according to the problem context. Hence, any language to define such transformations must have mechanisms for accessing instances.

The benefits to MDA of merging synthesis and modeling come from the fact that synthesis systems are good at automating transformations. For example, the AutoFilter [5] and AutoBayes [1] systems apply multiple schemas to solve a particular problem, and the correct application order of the schemas can be found through search-based methods. MDA could benefit from these techniques.

2. Schema-based Synthesis

Program synthesis comprises a range of techniques for the automatic generation of low-level executable code from high-level, declarative specifications of program behavior. Traditionally, program synthesis has taken the *deductive* approach, where programs are formally derived within a constructive theorem prover. The *generative* approach, in contrast, automates the combination of program templates. The *schema-based* approach, which we adopt here, is a combination of these two paradigms.

A *schema* is essentially a program template together with applicability conditions. During synthesis, schemas are recursively applied to assemble code in an *intermediate language*. Although the intermediate language is independent of the target implementation language (and so is like a PIM), it forms an (intermediate) platform in its own right so we regard it as a PSM. When a program in the intermediate language has been fully constructed, it is passed to a backend code generator which then translates the program into a given target platform.

3. Schema-based Modeling

In this section, we propose a schema development process which directly incorporates explicit models. Our idea is that schemas should be defined with respect to *explicit* input and output models. First we give a general overview of the schema development process and then we discuss how this impacts on the models and schemas.

Figure 1 shows the artifacts involved in model-based schema development. The input model is a representation of the key concepts that can be included in specifications and their inter-relationships. The output model, on the other hand, defines a model of the generated code. The action of schemas is to gradually instantiate the output model. Thus its instantiation can be regarded as representing the *synthesis state* so, in addition to code fragments, records any design decisions that have been made in the course of synthesis, plus any extra information that the schemas need.

The input/output models are independent of a particular specification language or intermediate programming language. Rather, they are domain-specific representations of the structure and relationships of the generated artifacts[1] — i.e., an abstract syntax for the domain-specific artifact generated.

Access to the models is mediated via front- and back-ends. The input model must come with an extraction function that defines how input model elements can be derived from elements of a particular specification language. Similarly, the output model requires a translation function that describes how to obtain code in the intermediate language given an instantiated model.

In addition to providing models of the input and output of the synthesizer, it is often useful to optionally provide "snapshots" at various stages of the synthesis process. These *internal models* can specify additional entities, which do not appear in the final model. Moreover, models may have additional constraints specified between them, shown by dotted arrows in Figure 1, which can be used for verification purposes both during and after synthesis.

The upper half of Figure 1 shows the process for developing a schema. A schema takes as input two models — an instantiated input model and a partially instantiated output model[2] and returns a partially instantiated output model. Scoping mechanisms can be used to limit the input model that a schema has access to or to limit the output model that can be instantiated. This can be used, for example, to indicate that a schema only constructs a certain fragment of the program. In principle, access to the input specification can also be scoped, but non-compositionality often means that this is not appropriate. Schemas typically need access to most of the input model to construct code fragments

Ideally, models should be developed before schema writing begins. In practice, however, things are likely to be less clear-cut, with model and schema development proceeding in parallel. It is precisely because of this incremental development of models that we need schemas to be defined with respect to explicit models.

We now illustrate these ideas with an example from the state estimation domain. We discuss how to define models and give a schema following the methodology set out so far.

---

1 which may, in general, be something other than code. Here, we use "code" in a general sense.

2 more generally, a schema could take internal models.

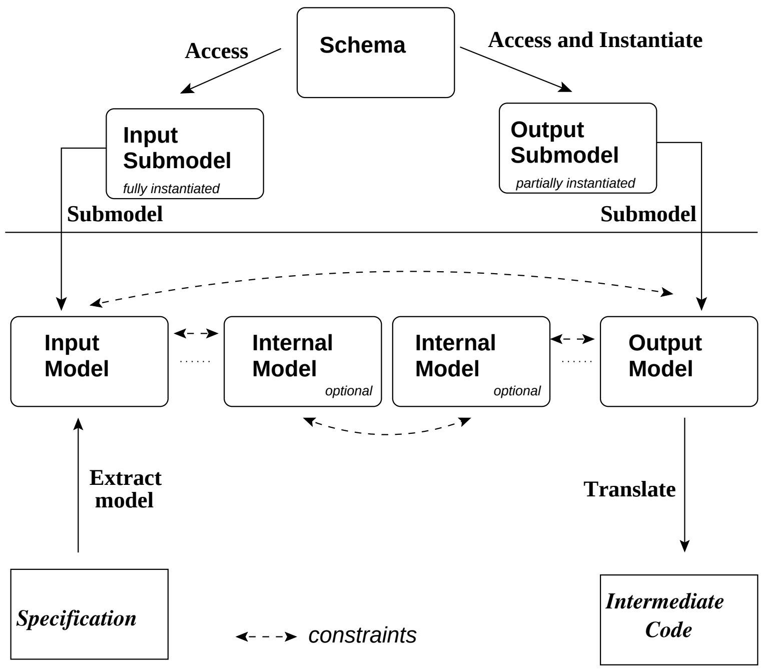

Figure 1: The Schema Development Process.

3.1. Kalman Filter Models

We will use UML class diagrams as our modeling language. An alternative would be some form of grammar notation although this is more appropriate for syntactic domains. A graphical notation like class diagrams is less prescriptive, and more appropriate for underspecified domains.

We use *Kalman filters* as a motivating example. These are recursive signal processing algorithms used to estimate system state from noisy sensor data. AutoFilter is a schema-based program synthesis system which can automatically derive a range of Kalman filters from high-level specifications. This is a suitable domain for program synthesis (not least because of its relevance for NASA) since there is a wide range of algorithms used to solve mathematically well-defined problems in this area; yet it is precisely the variability and complexity of potential solutions which makes implementations laborious and error-prone.

The output model for Kalman filters is given in Figure 2. It describes the "solution space" in terms of the high-level structure of the possible solutions. Input models can be given similarly but, in contrast, describe the mathematical structure of the "problem space" in terms of the physics of the problem.

The model given here simply defines the static syntactic structure of the generated code. We can also enforce semantic constraints on the input and output models (as well as between them) by annotating models with OCL constraints. Schemas (i.e. model transformations) would then be required to satisfy these constraints.

There are two ways of checking these constraints: dynamically, at synthesis time [2], and statically, at schema definition time. A dynamic check ensures that the application of schemas to a given input specification satisfies the constraints. A static check would ensure that this holds for any valid input.

Although the tradition within program synthesis (especially the deductive approach) has been to completely axiomatize the problem domain and reason formally about the derivation process, we aim, rather, to allow users to choose their level of formalism, by allowing optional annotations.

3.2. A Kalman Filter Schema

In keeping with the OO-style we are following, we use a Java-like syntax to define a schema for a standard Kalman Filter (Figure 3). This schema generates fully instantiated code but, in general, a schema need only partially instantiate a model. Schemas have two inputs: a fully instantiated input model, and a partially instantiated output model. Schemas are scoped to restrict access to the full output model — the notation *schema_name(in_model, out_model :: Class)* means call the schema with name *schema_name* with input model *in_model* and output model defined as the directed acyclic graph in `out_model` with *Class* as root.

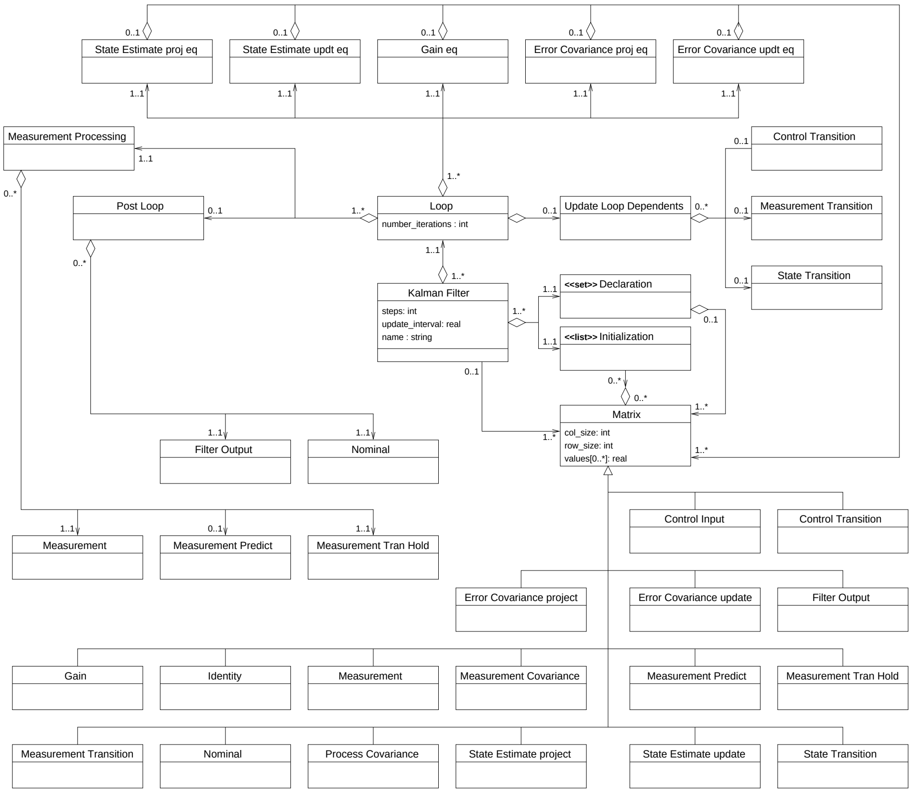

Figure 2: Domain description of Kalman Filter output (from [2]).

Declarations are similarly scoped. In the third line, `kf` is declared as a new kalman filter scoped to the output model (`kf_out`), and later `kf_loop` is declared as a new loop (again, in `kf_out`). We then link the two by assigning `kf_loop` to be the loop of the filter (i.e. `kf.loop`).

The schema calls a number of sub-schemas, each of which constructs a fragment of the program text. For example, `kf_declarations` constructs the appropriate variable declarations and makes this information available to the other schemas. Finally, the schema returns the (partially) instantiated output model.

Hence the output model is gradually instantiated by the schemas. Objects are created, their attributes are filled in with code, and this code is used, in turn, by other schemas to create more code. The details of the synthesis process can be found in [1, 5].

Schemas also contain assumptions and preconditions (omitted here). The informal distinction is that preconditions can be checked for satisfiability from the specification whereas assumptions cannot (because nothing has been said about them in the specification). It may be desirable to identify properties as first class operators in the models and to restrict assumptions and preconditions to expressions defined over those operators only. Both assumptions and preconditions can refer to the original specification, as well as to what has been constructed already.

4. Comparison of MDA and Schemas

The MDA transformation language suggested by the authors in [3] differs from our schema-based synthesis approach in a number of respects.

One difference is that schemas can be applied nondeterministically, returning a number of distinct solutions. Our system uses backtracking to search for solutions. In contrast, MDA transformations are deterministic. Also, schemas call explicit sub-schemas, rather than using implicit mappings and, since they are written in a general-purpose programming language, schemas can be composed in complex combinations (e.g. recursively or iteratively).

Also, the typical application domains differ. Whereas synthesis is concerned with mathematically and algorithmically complex domains, the MDA, to date, has primarily been applied to automating the generation of interface and database access code for data-processing applications.

However, we believe that these differences are more historical than fundamental. Synthesis can benefit from

```
public schema: linear_discrete_kalman_filter (ddkf_in kf_in, ddkf_out kf_out) {
      /* Declare new filter and give its name from spec */
     kf_out::Kalman Filter kf = new kf_out::Kalman Filter();
     kf.name = kf_in::Model.name;

     /* ASSUMPTIONS */
           ...
     /* PRECONDITIONS */
           ...
     /* Instantiate main KF loop */

     kf_out::Loop kf_loop = new kf_out::Loop();
     kf.loop = kf_loop;
     kf_loop.lower_bound = 0;
     kf_loop.upper_bound = kf_in::Model.steps;

     /* Instantiate rest of output model by calling subschemas.
        Each subschema is restricted to a submodel of output model.  */

     kf.declaration = kf_declarations(kf_in, kf_out::Declaration);
     kf.initialization = kf_initialization(kf_in, kf_out::Initialization);
           ...
     return kf_out; }
```

Figure 3: Top-level schema for standard Kalman Filter.

using explicit modeling, and MDA would benefit from the richer features of schema-based synthesis.

5. Conclusions

Our current efforts lie in developing language support for a schema-based synthesis system, explicitly linking schemas to models. We anticipate a core schema language, together with various optional extensions, such as a means of specifying an architecture, a way to incorporate comments and correctness annotations into the synthesis process, optional postconditions in schemas, or a means of referring to the synthesis state. We are also formalizing a semantics for the schema language.

We believe that a model-based schema language for program synthesis offers a number of advantages. First, it makes it possible for domain experts to adapt and extend existing schemas, and to create new ones. In the current implementation of AutoFilter, assumptions about the domain model are implicitly distributed throughout the code, so it is not always clear where structural assumptions have been made. Second, we can enable some form of correctness checking on the well-formedness of schemas.

Finally, there are several interesting extensions to the modeling languages that might be useful for program synthesis, such as hierarchy, scoping, and ordered aggregations.

References

[1] B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 13(3):483–508, May 2003.

[2] E. Grant and J. Whittle. Checking program synthesizer input/output. In *Workshop on Domain-Specific Modeling, OOPSLA '03*, Los Angeles, CA, 2003.

[3] A. Kleppe, J. Warmer, and W. Bast. *MDA explained (The Model Driven Architecture: Practice and Promise)*. Addison Wesley, 2003.

[4] J. Mukeri and J. Miller. *MDA Guide Version 1.0*. Object Management Group Specification, 2003.

[5] J. Whittle and J. Schumann. Automating the implementation of Kalman filter algorithms, 2004. Accepted for publication.