Using Domain Models in Extensible Schema-based Software Synthesis

Eugen C. Nistor
Department of Informatics
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
enistor@ics.uci.edu

1. Introduction

The software used in NASA missions is growing more and more complex, but at the same time it needs to satisfy increasingly stringent affordability and reliability requirements. Software synthesis provides a way of achieving these goals by generating code from high-level specifications. Two applications developed within the ASE group, AutoBayes [7] and AutoFilter [15], use schema-based software synthesis to generate code for the data analysis and state estimation domains, respectively. The goal of our project for the summer internship was to research an extensible framework that would allow the use of software synthesis technology to other application domains within NASA.

Schema-based software synthesis uses generic code templates called *schemas* to represent general knowledge about software generation in a reusable format. A synthesis system takes as an input a problem specification, and applies schemas recursively in an exhaustive fashion until a complete implementation of the problem in a chosen implementation language is created.

One of the main advantages of using schema-based synthesis is the ability to easily explore the design space. If there are different solutions to the same problem, different schemas can be created to implement each solution. In this way, the result of the synthesis process will be a tree of equivalent implementations, and the user can choose the one that satisfies the required performance metrics.

The synthesis system is divided into two distinct parts: the frontend and the backend (Figure 1). The frontend is responsible for applying the schemas necessary to generate an implementation that satisfies the problem specification. This implementation is written using an internal intermediate language. The backend translates the solution from the intermediate language in a specific programming language and for a specific architecture. The frontend and the backend are similar in the sense that they both use schemas to transform one representation of the generated program into other type of representation, with more information added. However, one thing that makes the frontend different is the fact that the schemas involved in the frontend are application domain oriented: an important part of the information of how to solve the problems and which algorithms to apply is an integral part of the application domain.

The work presented in this report is concerned with how to achieve an extensible synthesis system to new application domains. Since the frontend is the part where the domain information is used, the work is scoped to the frontend part of the synthesis system. The problem of how to make the synthesis system extensible to support new target implementation platforms in the backend was studied by another intern in our team, Tomas Bures [2]. In order to experiment with different modeling aproaches, we have designed AutoLinear, a synthesis application in the domain of matrix calculations. In particular, AutoLinear generates code that solves systems of linear equations, commonly described in mathematical notation as $Ax{=}b$.

2. Analysis of the Synthesis Process

AutoBayes and AutoFilter were developed for two specific NASA domains: data analysis and signal processing, and all the corresponding domain knowledge was implicit and partially embedded inside their schemas. With the addition of new schemas, the two applications have reached a critical point. The effects of not having the knowledge explicitly described in the

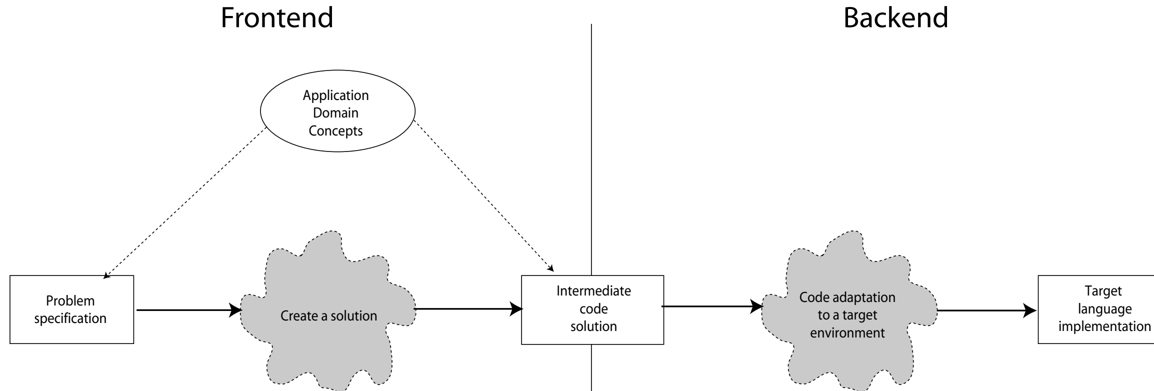

Figure 1. An overview of the software synthesis system.

system has led to a number of problems.

Comprehensibility : It is hard to understand the purpose of a schema and the consequences of modifying it by just looking at its implementation.

Scalability : Since there is no description of how the existing schemas are related, it is difficult to determine how will the introduction of a new schema affect the system.

Reusability : When a new schema needs to be written, there is no mechanism of searching the existing schema libraries and find out if there is already another schema that implements either the same functionality or some partial functionality that can be reused.

Our goal is to create a synthesis system that is extensible to other domains. In order to achieve this, we have to explicitly describe the knowledge that is currently implicit in AutoBayes and AutoFilter, and determine how is the application domain knowledge related to other types of information in the synthesis process.

Our analysis revealed the following types of information currently embedded into a synthesis system:

1. The *application domain* describes the concepts that are used in the particular domain for which the synthesis system is applied. The application domain comprises the general knowledge in the area and it is expected to be described by a domain expert who might have very little knowledge about the synthesis process. A description of how to capture domain knowledge through domain engineering is described in [4]. In our AutoLinear example, it contains descriptions of the concepts of *matrix* and *vector*, and a set of algorithms that can be used with matrices and vectors.

2. The *problem specification* describes the problem that the synthesized program needs to solve. The specification languge uses concepts from the application domain and can additionally assign values for their properties. In AutoLinear, the problem of solving a system of linear equations, denoted in mathematics as $Ax{=}b$, is specified using a matrix A and a vector b as inputs, and the vector x as output.

3. The *intermediate code* uses the concepts in the solution space. The intermediate code is a pseudocode, and uses programming language constructs such as for-loops, variable declarations, expressions etc. The concepts from application domains can be used as data types, and their properties can be accessed as functions.

4. The *schema language* is used to implement schemas. As in AutoBayes and AutoFilter, the schemas in AutoLinear are implemented in Prolog, but now they use specific high-level APIs to access the problem specification and the intermediate code.

5. The *schema application language* describes how should schemas be selected and applied. AutoLinear uses Prolog because its unification and backtracking features, and can apply either a specific schema, or apply recursively all schemas from a certain family.

Any of these types of knowledge can be modeled independently, using different languages or technologies.

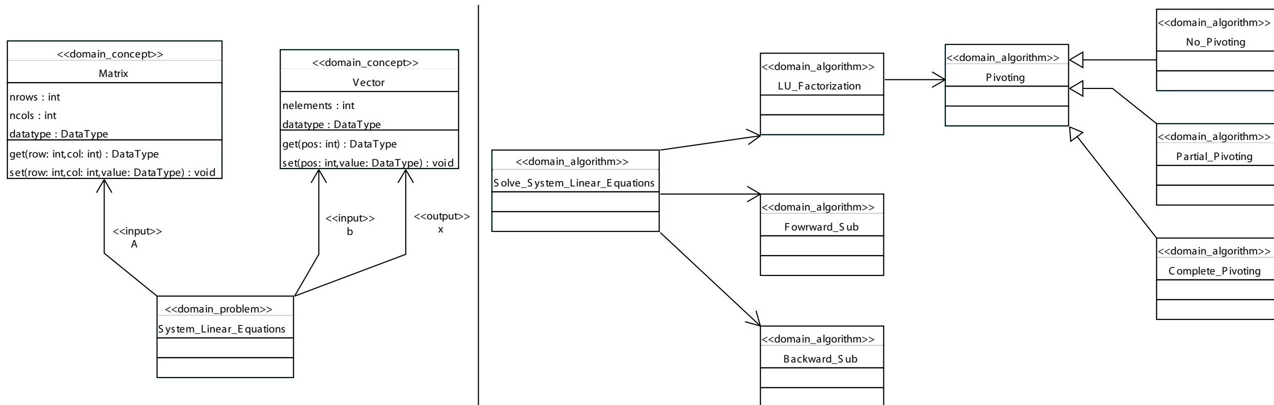

Figure 2. Partial view of the model for matrix domain in UML: a) Specification of the "system of linear equations" algorithm; b) An overview of all algorithms in AutoLinear.

Neither AutoBayes nor AutoFilter currently uses any type of explicit models for any of these parts.The next section discusses how we have used explicit models for the application domain, problem specification and intermediate code in AutoLinear.

3. Use of Explicit Models

A model is a formal description of the structure and properties of a piece of information. Our extensible synthesis frontend uses explicit models for the application domain information, the problem specification and the intermediate code. The relationship between the three models in the frontend is shown in Figure 1.

The application domain concepts are an integral part of both problem specification and intermediate code solution. Because the application-specific part will change with every new domain that the synthesis application will be applied to, a requirement for the model representations we use is *extensibility support*.

The input to the AutoLinear frontend is an instance of the problem specification model, and the result of the synthesis is an instance of the intermediate code model. With each schema application, some new information is added to the instance of the intermediate code model. The models describe the structure in a formal way, and therefore it is possible now to verify that the schemas create intermediate code that conforms to its model. In this way, most of the static errors in the generated code can be discovered at the synthesis time, by verifying the intermediate code after each schema is applied. Without the use of explicit models, some of these errors could not be discovered

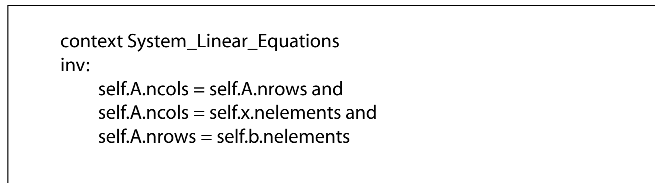

Figure 3. OCL constraint for the system of linear equations.

until the synthesized program was executed, and at that time it was very hard to determine which schema was wrong from the ones that were applied. Therefore, two other requirements for the modeling notation are *support for describing constraints* and *ability to check conformity* of the model instance to the constraints.

We have used two different modeling notations, UML and XMLSchema, to describe the three models in AutoLinear.

3.1. UML as Modeling Language

UML [10] is a graphical modeling language used in object-oriented systems. It consists of a number of different static and dynamic diagrams that describe important aspects in the design of a system. The most important diagram in UML is the *class diagram*, that shows the system as composed of classes and associations. A class represents an entity with encapsulated behavior, described by a name, a set of attributes and a set of operations. If we represent domain concepts as classes, then class diagrams can be used to describe domain models.

The main strength of UML comes from the use of a

```
%check that every variable that is used was declared in a block or a code unit above
<xsl:template match="il:variableName">
        <xsl:variable name="val" select="text()"/>
        <xsl:variable name="blocks" select="./ancestor::il:block/il:localDecl/il:variableDecl/il:variableName[text()=$val] |
                    ./ancestor::il:codeUnit/il:inputDecl/il:variableDecl/il:variableName[text()=$val] |
                    ./ancestor::il:codeUnit/il:outputDecl/il:variableDecl/il:variableName[text()=$val] |
                    ./ancestor::il:codeUnit/il:externalDecl/il:variableDecl/il:variableName[text()=$val] "/>
        <xsl:if test="count($blocks)=0">
            <xsl:message>
                The variable <xsl:value-of select="$val"/> was used but never declared
                in codeunit '<xsl:value-of select="./ancestor::il:codeUnit/@il:name"/>' !
            </xsl:message>
        </xsl:if>
</xsl:template>
```

Figure 4. XSLT file snipped that uses XPath to check a consistency constraint.

graphical notation. In this way, concrete designs can be shared between technical and non-technical people. A number of different UML design environments are available for creating class diagrams. For AutoLinear, we have used ArgoUML [1].

The class diagram graphical syntax has only limited power in expressing structural constraints. More complex constraints need to be expressed using OCL, the object constraint language. OCL was developed to describe invariants, preconditions and postconditions for the operations of a class, and has a powerful expression language. Most of the UML tools, including ArgoUML, have support for writing OCL constraints. However, support for checking the OCL constraints on an model instance is not widespread. Tools such as Dresden OCL Toolkit [6] can embed code that checks OCL constrainsts in an object-oriented implementation generated from the UML model.

In order to assign semantics to a class diagram, the classes and associations need to be tagged with meaningful information. The standard mechanism in UML is to use labels called stereotypes. For the application domain part in AutoLinear, we have used two types of stereotypes: << *domain concepts*>> and <<*domain algorithms*>>. Figure 2 shows two class diagrams that we have created to define the matrix domain for AutoLinear in UML. The diagram on part a) shows a description of the two domain concepts in AutoLinear, a *Matrix* and a *Vector*, and a specification of the systems of linear equations problem in terms of its inputs and outputs, differentiated by stereotyped associations. The OCL expression in Figure 3 describes the constraints for the system of linear equations: the matrix A has to be square and its dimensions must match the dimensions of b and x. The diagram in part b) of Figure 2 shows the hierarchy between all the algorithms in this matrix calculations domain. The simple associations denote “uses” or “calls” relationship, while the generalizations are used for “is-a” relationships between algorithms.

3.2. XML as Modeling Language

The other model representation we have investigated is XMLSchema [14]. In this setting, XML [12] is used as the representation for the input and output of the schemas. XML is a markup language for describing arbitrary structured data. It is widely used as a format for data interchange between different applications, but its explicit format makes it suitable for representing internal application data as well.

XMLSchema [1] is a language for defining the structure and contents of XML files. XMLSchema language has the necessary features to describe structural properties, such as types for elements, types and number of sub-elements, or value ranges for attributes. It also has natural support for extensibility, by allowing derivations of types in different namespaces.

For expressing complex constraints, a more complex solution is needed. For this, we have used XPath expressions in an XLST companion file. XPath [13] is a complex pattern matching language that can be used to express complex conditions between different elements in an XML file. A simple XSLT file checks a set of XPath expressions and generates an error message if they are not met. The constraint checking for AutoLinear has thus two steps: the first step is the verification of the simple structural constraints by invoking an XML validator, and the second step is the evaluation of the complex consistency conditions by executing the XSLT file. The example in Figure 4 shows an example XSLT code verifies the condition that “ev-

[1]XMLSchema is a W3C language, and is not related to the schemas in schema-based software synthesis

ery variable that is used has been previously declared" for a program written in the intermediate code.

4. Related Work

Refine [11] was a knowledge-based software synthesis system. The domain knowledge was described as a logical theory, and a transformation as a mapping between different theories. Refine had an object oriented representation of domain concepts, but required all domain knowledge to be described in its own format. We chose to investigate using UML and XML as model representations in order to create a program synthesis system that will easily interoperate with other applications, and thus avoid one of the pitfalls that led Refine to become extinct.

OMG's *Model Driven Architecture* (MDA) [9] advocates writing software as model transformations from platform independent models (PIMs) to platform specific models (PSMs). These types of transformations are very similar to the schemas in software synthesis [5]. However, the main difference between MDA transformations and schemas is that the transformations are much more coarse grained than schemas, and correspond to a number of schemas that are applied together to transform a problem specification model into an intermediate language model.

Ontologies are a way of describing a shared knowledge among different software systems [8]. From a software perspective, ontologies will provide a general vocabulary of concepts that can be reused at company or research-community level. In this respect, our application domain models satisfy the role on a domain specific software ontology. A more complete discussion of how ontologies can help in software synthesis can be found in [3].

5. Conclusions

In this paper, we have introduced an infrastructure that uses explicit domain models to address the problem of extensibility of schema-based software synthesis. We have used explicit models for the problem specification, intermediate code, and for the general application domain-specific concepts.

We have also investigated two different modeling notations for representing the explicit models. While both UML and XMLSchema have comparable abilities, UML is more suited for high-level domain modeling while XMLSchema has the power of expression to be used in describing complex low-level models. A complete solution will need to combine the best features from both these modeling notations.

The ideas described in this paper, and the AutoLinear prototype, can be used in a future work to create a next-generation extensible synthesis system.

References

[1] ArgoUML, http://argouml.tigris.org/.

[2] T.Bures, "Using Explicit Models in An Extensible Program Synthesis Backend", September 2004.

[3] T. Bures, E. Denney, B. Fischer, and E. Nistor, "The role of ontologies in schema-based program synthesis", presented at OOPSLA Workshop on Ontologies as Software Engineering Artifacts, October 2004.

[4] K. Czarnecki and U. Eisenecker, "Generative Programming : Methods, Tools, and Applications", Addison-Wesley, June 2000.

[5] E. Denney, J. Whittle. "Combining Model-driven and Schema based Program Synthesis", Proceedings of the 1st International Conference on the Applications of UML and MDA to Software Systems (UMSS'2004), June 21-24, 2004, Las Vegas, Nevada.

[6] Dresden OCL Toolkit, http://dresden-ocl.sourceforge.net/.

[7] B. Fischer and J. Schumann, "AutoBayes: A system for generating data analysis programs from statistical models", Journal of Functional Programming, vol. 13, no. 3, pp. 483-508, May 2003.

[8] T.R. Gruber, "Towards principles for the design of ontologies used for knowledge sharing" ,Formal Ontology in Conceptual Analysis and Knowledge Representation. Kluwer Academic Publishers, 1993.

[9] Object Management Group, MDA Guide Version 1.0.1, http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf .

[10] Object Management Group, Unified Modeling Language Specification, http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf .

[11] D.R. Smith, G.B. Kotic, and S.J. Westfold, "Research on knowledge-based software environments at Kestrel institute", IEEE Transactions on Software Engineering, vol. 11, no. 11, 1985.

[12] W3C, Extensible Markup Language (XML) 1.0 (Third Edition) ,http://www.w3.org/TR/REC-xml .

[13] W3C, XML Path Language (XPath) Version 1.0, http://www.w3.org/TR/xpath .

[14] W3C, XML Schema Part 0: Primer, http://www.w3.org/TR/xmlschema-0/ .

[15] J. Whittle and J. Shumann, "Automating the implementation of Kalman-filter algorithms", 2004, in review.