The Role of Ontologies in Schema-based Program Synthesis

Tomáš Bureš
Charles University, Prague
bures@nenya.ms.mff.cuni.cz

Ewen Denney
QSS, NASA Ames Research Center
edenney@email.arc.nasa.gov

Bernd Fischer
RIACS, NASA Ames Research Center
fisch@email.arc.nasa.gov

Eugen C. Nistor
University of California, Irvine
enistor@ics.uci.edu

1 Introduction

Program synthesis is the process of automatically deriving executable code from (non-executable) high-level specifications. It is more flexible and powerful than conventional code generation techniques that simply translate algorithmic specifications into lower-level code or only create code skeletons from structural specifications (such as UML class diagrams). Key to building a successful synthesis system is specializing to an appropriate application domain. The AutoBayes [2] and AutoFilter [5] systems, under development at NASA Ames, operate in the two domains of data analysis and state estimation, respectively.

The central concept of both systems is the *schema*, a representation of reusable computational knowledge. This can take various forms, including high-level algorithm templates, code optimizations, datatype refinements, or architectural information. A schema also contains applicability conditions that are used to determine when it can be applied safely. These conditions can refer to the initial specification, to intermediate results, or to elements of the partially instantiated code. *Schema-based synthesis* uses AI technology to recursively apply schemas to gradually refine a specification into executable code. This process proceeds in two main phases. A *front-end* gradually transforms the problem specification into a program represented in an abstract intermediate code. A *backend* then compiles this further down into a concrete target programming language of choice. A *core engine* applies schemas on the initial problem specification, then uses the output of those schemas as the input for other schemas, until the full implementation is generated. Since there might be different schemas that implement different solutions to the same problem this process can generate an entire solution tree.

AutoBayes and AutoFilter have reached the level of maturity where they enable users to solve interesting application problems, e.g., the analysis of Hubble Space Telescope images. They are large (in total around 100kLoC Prolog), knowledge-intensive systems that employ complex symbolic reasoning to generate a wide range of non-trivial programs for complex application domains. Their schemas can have complex interactions, which make it hard to change them in isolation or even understand what an existing schema actually does. Adding more capabilities by increasing the number of schemas will only worsen this situation, ultimately leading to the "entropy death" of the synthesis system.

The root cause of this problem is that the domain knowledge is scattered throughout the entire system and only represented implicitly in the schema implementations. In our current work, we are addressing this problem by making explicit the knowledge from different parts of the synthesis system. Here, we discuss how Gruber's definition of an ontology as "an explicit specification of a conceptualization" [4] matches our efforts in identifying and explicating the domain-specific concepts. We outline the roles ontologies play in schema-based synthesis and argue that they address different audiences and serve different purposes. Their first role is *descriptive*: they serve as explicit documentation, and help to understand the internal structure of the system. Their second role is *prescriptive*: they provide the formal basis against which the other parts of the system (e.g., schemas) can be checked. Their final role is *referential*: ontologies also provide semantically meaningful "hooks" which allow schemas and tools to access the internal state of the program derivation process (e.g., fragments of the generated code) in domain-specific rather than language-specific terms, and thus to modify it in a controlled fashion.

For discussion purposes we use AutoLinear, a small synthesis system we are currently experimenting with, which can generate code for solving a system of linear equations, $Ax = b$.

2 Sources of knowledge and concepts

Program synthesis systems are knowledge-rich and work with concepts from wide range of different sources. Each of the sources introduces concepts on a different level of abstraction. However, as the synthesis process is in fact a gradual refinement (although not necessarily in the technical sense of the word), concepts from the different sources are often related (e.g., matrices as mathematical concept, as elements of problem specifications, as datatypes in an intermediate language, and as two-dimensional arrays in a target language). We use ontologies to formally declare and then categorize these diverse concepts, and link related concepts together.

Generative programming [1] distinguishes between the *application space* and the *solution space* as the two sources of concepts. We will keep this as a top-level categorization, but argue that we need a more refined distinction. In the rest of this section we identify the different concept sources and the major concepts for each of these sources.

2.1 Application space

Intuitively, the application space contains all concepts that are required to describe the application problems. For AutoLinear, this includes *data types* like vector and matrix, *operations* like matrix element selection and matrix addition, and *predicates* like is-diagonal and is-invertible. These concepts have to be defined by application domain experts. Since they constitute the common vocabulary of discourse for the different parts of the system and for the user, the format must be understandable by the different participants. For example, we aim to explicitly link the domain concepts to representations from the specification language, to intermediate language constructs and to target programming language constructs. In this way, the final program can be documented with meaningful annotations.

However, not all concepts are equal. While some concepts are available to the users of the synthesis system to formulate their problem specifications (e.g., matrix), others are only used by the system developers to implement the schemas (e.g., diagonal). In general, we thus refine the application space into the

problem specification domain i.e., concepts that are used to formulate the high-level problem specifications, and the

background theory domain i.e., concepts that are not expressible in problem specifications but are required to formulate semantic constraints such as the correctness of specifications or schema applicability conditions.

It is important to note that the problem specification domain is *not* the specification language itself but rather defines its abstract syntax. The different domain concepts such as *vector* and *matrix* can have different associated syntaxes ("syntactic sugar") in different specification languages. For example, AutoLinear uses a Matlab-like notation for specifying matrix literals, but this could be changed easily by adapting a new parser for the specification language.

2.2 Solution space

Intuitively, the solution space contains all concepts that are required to formulate the generated programs. However, since the semantic gap between specifications and generated programs is large (i.e., specifications are much more concise than programs), the solution space is not a simple dual of the application space but provides a much wider variety of concepts.

Like the application space, the solution space is refined into several domains:

intermediate language domain i.e., elements of the intermediate language or languages,

target language domain i.e., elements of the target language and environment,

algorithm domain i.e., the different algorithms and sub-algorithms that are implemented as schemas,

search control domain i.e., concepts that are not expressible in the language domains but that are used to control the search for applicable schemas, and

meta-programming kernel i.e., concepts expressing operations on objects defined in any of the other domains that can be used to implement schemas.

These domains are interrelated, and at several levels (such as control versus implementation). Here, we will discuss the two implementation language domains. AutoLinear's intermediate language contains the usual procedural programming constructs (e.g., variable declaration, for-loop, if-then-else branch, function definitions) and basic data types (e.g., int, double, bool). In addition, it also contains higher-level programming constructs (e.g., finite summations and convergence loops) and data types and operations representing application domain concepts (e.g., matrix and matrix operations). However, the intermediate language domain also contains concepts reflecting programming language semantics (e.g., lvalue and scope) which are used to define and enforce the well-formedness of the generated code fragments. AutoLinear uses different versions of the intermediate language at different stages of the refinement process. Each version is more restricted in its use of the higher-level concepts. The final stage of this produces executable code for the chosen target. This is an actual programming language (i.e., C or C++) together with a suitable implementation of the relevant application domain concepts (e.g., matrix), either using target libraries (e.g., the Matrix-class from the Octave library) or using concepts available in the language (e.g., two-dimensional array). Note that it is a consequence of our design that we distinguish the intermediate language used in the front-end from the target languages used in the backend. In principle, the synthesizer could directly generate code in the target language.

3 Benefits of Ontologies

We are currently re-engineering our system so that the concepts it manipulates are grounded with respect to explicit ontologies. The idea is that the synthesis system will manipulate a sequence of *models* (input model, intermediate models, platform-independent model, platform specific model, and so on), where each model is built from concepts from a given ontology. We believe that the use of explicit models and ontologies will bring numerous benefits:

1. *The models and ontology act as documentation for programmers.* A large part of the difficulty in understanding a schema is determining the input requirements and what structures are expected for output. Schema inputs are often very complex and usually contain intermediate results from previous schemas, as well as parts of the original problem specification, and must comply with the specification of the target language and the target architecture.

2. *It makes writing schemas easier.* Using a formally defined ontology, we can automatically generate an API for model manipulation which can then be used in the schema code, so that programmers do not have to deal with the internal representation of models. The API has constructors for concepts and getter/setter methods for concept attributes. It also lets schemas access higher-lever predicates built on ontological reasoning (e.g., *a matrix is square*, *a matrix is sparse*).

3. *It controls schema interaction.* Schemas produce partially instantiated code. In a sense, they can be regarded as constructing code families. Thus, for schemas to interact consistently they need to agree on the properties of these families, and this is achieved by using an ontology.

4. *It allows search control.* By classifying schemas into families based on a conceptual hierarchy, a synthesis engine or user can search for schemas which have a specified functionality.

5. *It facilitates extensions.* By making explicit which domain concepts are used in the various languages, the system can more easily be ported to new domains, possibly reusing some parts of the ontology.

6. *It allows us to validate the output of a schema.* If the ontology is formally defined, the system can automatically check whether the output of a schema is well-structured and whether any additional constraints which may be required hold.

7. *It ensures consistency throughout the synthesis process.* Schemas can be seen as gradually refining abstract concepts from the specification domain into a concrete implementation (e.g., *matrix*). Although we do not formally define schemas as refinements, linking abstract and concrete concepts to a common concept in the ontology can be used to enforce consistency.

8. *It enables generation of additional knowledge-based artifacts.* Schemas can be augmented with ontological information. This allows the possibility of generating informative artifacts relating to the derivation process, such as documentation explaining how the code has been derived. This can be

recorded either as in-line comments, or as a separate document. Besides mark-up intended for humans, we can also generate logical annotations which can be used for automated verification of the code.

There are several possible languages which can be used to represent the ontologies and we are currently evaluating OWL, KIF, and UML, with respect to two main criteria:

Expressiveness The language should be able to express complicated structural constraints and support higher-level concepts (e.g., square matrix, sparse matrix).

Tool support Tools should integrate easily with the synthesis environment and be able to verify that models produced by schemas comply with a given model. Generation of parts of the synthesis system from the ontologies (e.g., APIs) is an advantage. Editing and viewing capabilities are especially valuable for complex ontologies.

Since the existing ontology frameworks require substantial customization before they can be applied as domain modeling languages in schema-based program synthesis systems, we have begun, in parallel, to develop our own XML-based language.

References

[1] K. Czarnecki and U. W. Eisenecker, "Generative Programming: Methods, Tools, and Applications", Addison-Wesley, 2002.

[2] 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.

[3] M. R. Genesereth and R. E. Fikes, "Knowledge Interchange Format, Version 3.0, Reference Manual", June 1992

[4] T. R. Gruber, "Toward Principles for the Design of Ontologies Used for Knowledge Sharing", Proceedings of International Workshop on Formal Ontology, Padova, Italy, March 1993.

[5] J. Whittle and J. Schumann, "Automating the implementation of Kalman-filter algorithms", 2004. In review.