The Tool Factory

Steve Cook, Microsoft Corporation, stcook@microsoft.com

Stuart Kent, Microsoft Corporation, stukent@microsoft.com

Submitted to the OOPSLA2003 Workshop on Generative Techniques in the context of Model Driven Architecture.

Introduction

In this paper we address the problem of building tools to manipulate and process domain specific modeling languages. One way of handling this problem would be to have a single standard language (e.g. the Unified Modeling Language) intended to cover all possible domains; then a single tool would suffice. The development of UML Profiles [1] is an attempt to make this idea work, by providing a limited way to extend the syntax and semantics of the standard language for particular domains.

We believe that the profile approach is at best problematic, and at worst unworkable. Different domains place very different requirements on the languages that can be used to model them. Even mapping simple UML class diagrams effectively into popular programming languages such as C# or Java presents difficult problems that have not been resolved, for example the exact meaning of the notorious "black diamond" symbol, or the definition of the term "interface".

Instead of a "one-size-fits-all" approach, we believe that the best way to address the problem of tooling a wide variety of domain-specific languages is to be able rapidly and economically to specify new languages, and to provide automated support for building tools that implement these languages. We can exploit similarities in languages within a language family to simplify the generation of tools that manipulate members of that family. See [2] for some techniques to define language components that can be combined to create language families.

A *tool factory* is a software system used to build tools that manipulate and process the members of a family of domain specific languages from specifications of the family members. In what follows we describe the architecture of a tool factory.

Tool Factory Architecture

Figure 1 gives a bird's eye view of the architecture of a tool factory for a family of domain specific languages, F. In the diagram, ovals represent tools, rectangles with folded corners represent artifacts, solid arrows represent derivation and dashed arrows represent generation. The Language Designer is a tool for designing family members. The diagram shows a design for some language, X, created in the Language Designer. The design of X includes all of the following pieces:

- concrete syntax – a specification of how the language is physically presented to and manipulated by its users, in graphics and text
- abstract syntax – a specification of the concepts of the language and how they are inter-related
- serialization syntax – a specification of how expressions in the language may be serialized in a textual form for interchange between tools (typically but not necessarily using XML), and
- semantics – a specification of the meaning of expressions in X, expressed in terms of well-formed traces based on a trace language for X.

The design of X is created in the Language Designer from language fragment, Y, and language design pattern, Z, defined for the language family, F. Language designs created in the Language Designer are fed into a Design Tool Generator, which generates part of a design tool for X. We use the term *designer* to mean a design tool, and the term *X-designer* to mean a design tool for the language named X.

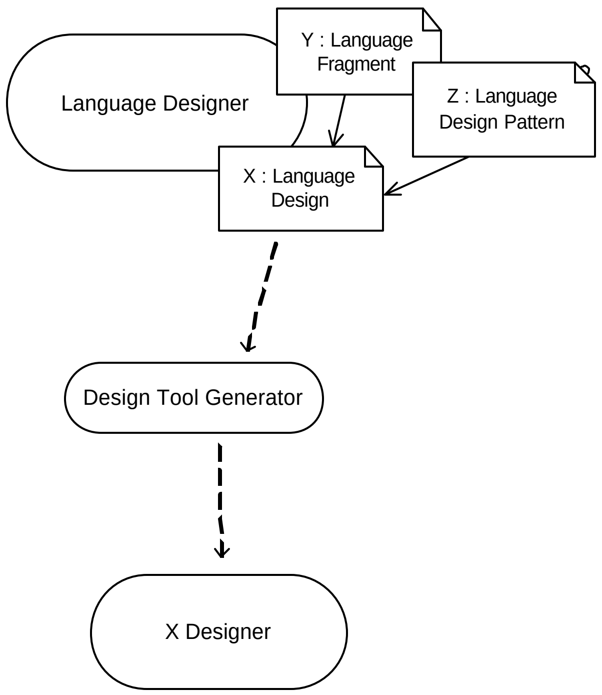

Figure 1 Overview architecture of a tool factory

Figure 2 gives one perspective on the architecture of an X-designer, in terms of functional components. At the heart of the architecture is an *In-Memory Store* (IMS), which stores both the Abstract Syntax Graph (ASG) and layout information for designs and traces of the language X. The format of the information is dictated by the definition of X and its trace language. The IMS also stores a definition of X itself, which can be read by code that needs to interpret the language.

The rest of the components work off the IMS. The *Design Surface* lets the user manipulate expressions and traces of X through their respective graphical and textual concrete syntaxes. It interacts with ASGs and layout information stored in the IMS, and manages graphical layout, as well as the display and parsing of text embedded in diagrams. The *Serializer* serializes ASGs and layout information from the IMS using the serialization syntaxes.

The *Rule Checker* checks well-formedness rules over information stored in the IMS. Static code could be generated from rules language expressions, but a more powerful alternative is to enable the Rule Checker to interpret the rules language expressions dynamically. This lets the language designer experiment with rules before fixing them in the language definition. It also lets the language user build queries that retrieve information from models, and filters that control the display of models. Interpretation also makes the pattern engine, and external tools, such as the translators shown on the diagram, more powerful and easier to build.

The *Pattern Engine* is a language independent component that implements pattern-based model mapping techniques. It uses the IMS, the design surface and language specific extensions, to provide those automations. A critical component of the Pattern Engine is a Merge Engine, which merges the results of pattern evaluations with each other, and with the existing elements in the target models. The Pattern Engine can also store pattern applications in the IMS, so that they can be edited and reevaluated when the values of their arguments change.

Models can be animated or interpreted, and used to trace execution. The *Animator/Interpreter* and *Model by Example* components implement these automations using the trace-based semantics for X.

- The *Animator/Interpreter* implements the mapping from models to traces to provide either an interpreter, for an imperative language, or an animator, for a declarative language. An animator asks the user to decide any non-deterministic choices that it can not resolve, while an interpreter always makes a choice.
- *Model by Example* implements the mapping in the opposite direction. It can automatically construct a partial model from example traces developed by the user (e.g. to illustrate particular problem scenarios) or recorded by an execution tracing tool.

The *XY Translator* and *XY Trace Translator* are examples of tools that interact with designers. If Y is a general purpose programming language, then the XY Translator might generate Y based code from X based models. However, if X is a business processes modeling language, and Y is a software architecture modeling language, the XY Translator might reconcile X and Y based models manipulated independently and concurrently by different users. In either case, the XY Translator would interact with the designers, but would not be embedded in them.

The *XY Trace Translator* might be used, for example, to monitor execution traces generated by an X based system, and to display them through X based models. If X is a declarative language, then the XY Trace Translator might also be used for testing, by checking the traces against the X based models, to see if they satisfy them. If X is a business process modeling language, then the XY Trace Translator might also be used to generate a dashboard, where traces of the running business processes could be tracked.

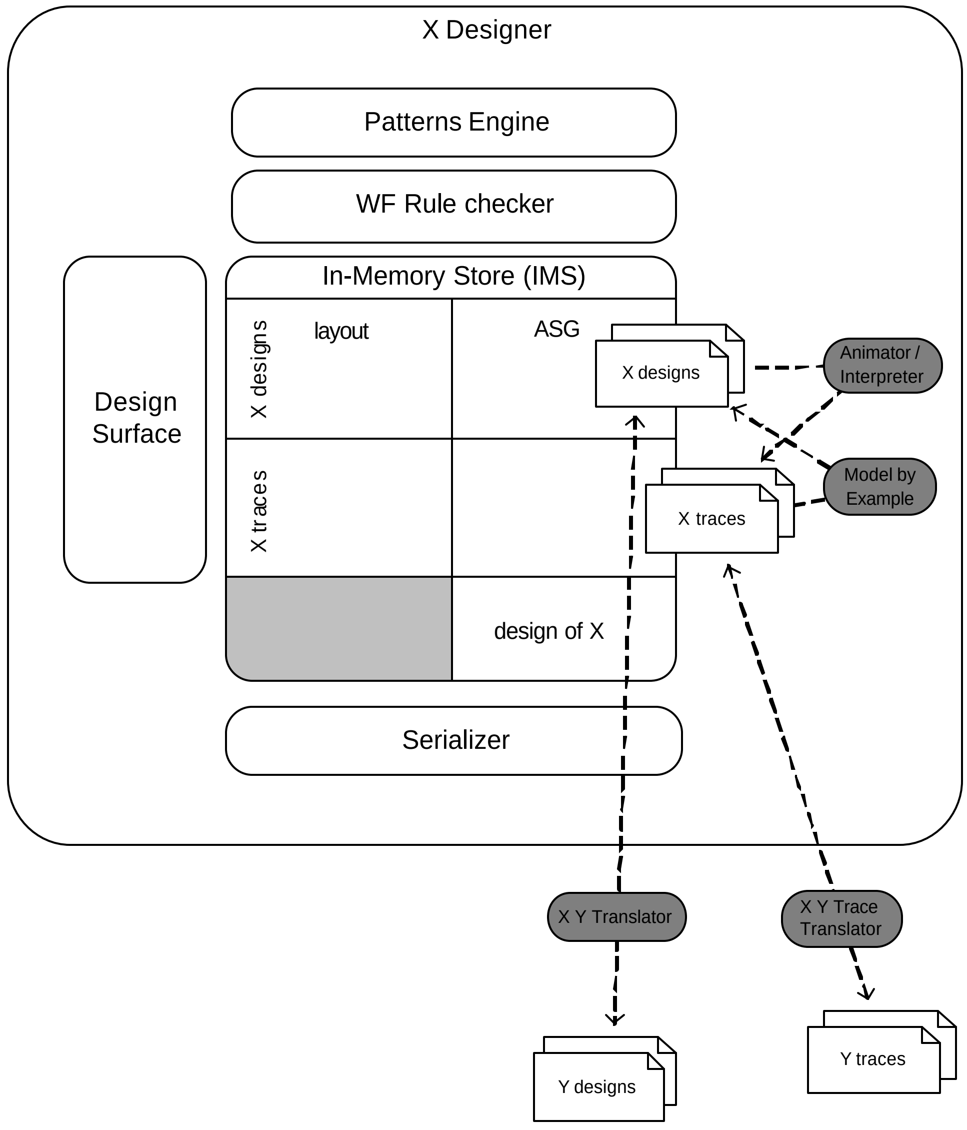

Figure 2 The Detailed Architecture Of A Designer

Figure 3 gives another perspective of the makeup of the X-designer. While our long-term vision is to generate complete designers from language definitions, this will certainly not happen soon, so we expect some hand-coding to take place. Code generation works best when the generated code completes an existing framework. Of course, there is a trade-off between placing functionality in the generated code, and placing it in the framework. In practice, a combination of the two approaches usually works best.

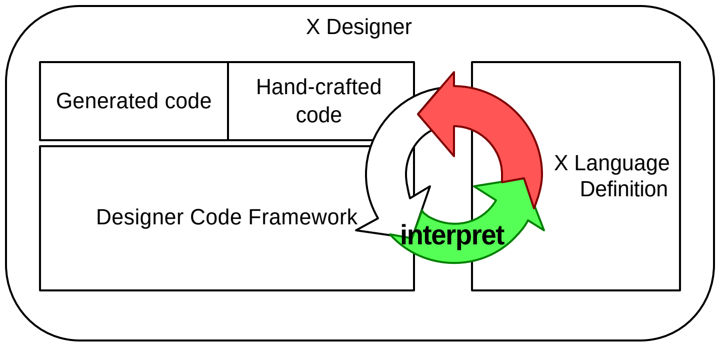

Figure 3 Where The Code Comes From

The Language Designer is just another X-designer, where X is the language for designing languages (LDL). Hence, it should have the same architecture as other designers, as illustrated in Figure 4. In particular, it is worth noting that in this case the pattern engine can be used to support the definition of language families. Figure 4 provides the last twist in the tale, observing that it should be possible to generate the LDL-designer from a definition of the language for designing languages (LDL). This allows the tool factory to bootstrap itself from one version to the next. In addition, provided the LDL is rich enough, it should be possible to define a mapping from language designs into the programming languages and frameworks used to implement designers, allowing the design tool generator to be bootstrapped, as well.

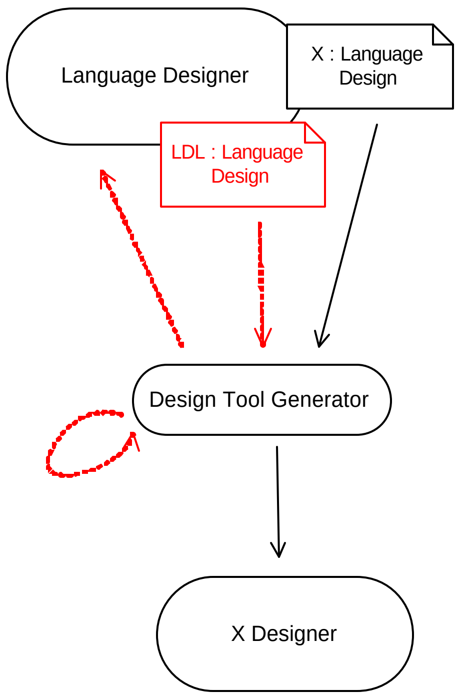

Figure 4 The Bootstrap

References

[1] S. Cook. The UML Family: Profiles, Prefaces and Packages. Proceedings of UML2000, edited by A. Evans, S. Kent and B. Selic. 2000, Springer-Verlag LNCS.

[2] A. Clark, A. Evans and S. Kent. A Metamodel for Package Extension with Renaming. In J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.): UML 2002 - The Unified Modeling Language 5th International Conference, Dresden, Germany, September 30 - October 4, 2002. Proceedings LNCS 2460, 305-320, 2002.

Possible Contribution to Workshop

Before joining Microsoft earlier this year, both Steve and Stuart were closely involved in the definition of the UML 2 standard, and published several papers about model driven approaches to software development. We have views about all of the topics of interest of the workshop. We could readily present in-depth material on any of the following topics:

- The tool factory (content of this paper)
- Problems with UML as a basis for MDA
- Approaches to pattern-driven model transformation