Managing Intent: Propagation of Meaning During Model Transformations

Jonathan Sprinkle
Vanderbilt University
jonathan.sprinkle@vanderbilt.edu

Abstract:
The Model Driven Architecture (MDA) maintains that systems can be abstractly specified, and specifically instantiated. The abstract specification is created using formally defined models; thus in order to fulfill these ambitions, model transformations must be employed on some level. The definition of these transforms—however—is not a trivial task. In addition to the basic problems of the transformation syntax, and an intuitive interface, there is a more significant and difficult problem: the translation of the platform-independent models (PIMs) into platform-specific models (PSMs) such that they are "correct" in the language of the PSM. This paper philosophically discusses the root causes of transformations, specifically the syntax and semantics changes that require MDA-like transformations between modeling languages. It also examines the difference between the intent of the modeler, and that of the model—an important distinction when creating transformations.

Introduction

Throughout the entire history of computing, there have been two seemingly contrary approaches to programming: the archetypal approach and the specific approach. The first example of this tension is the battle between programmers dedicated to machine code, and those in favor of high-level languages. Machine code was fast to execute, and its examination yielded not how the machine *should* act but instead how the machine *would* act, as the program was in the machine's native language. High-level language programmers, however, favored an approach that allowed the expression of the *intent* of a program in a language independent of that of the machine, and the subsequent transformation of that intent into the machine's language. Once these transformations were proved to be correct the era of machine code written by hand began to end, and the dawn of high-level languages began [1].

Of course, one person's high-level language is another person's low-level language. Today, FORTRAN and COBOL are considered to be tediously low-level for general GUI programming, but they were revolutionary in their time. Likewise, a reconfigurable von Neumann architecture seemed a radical notion when all electronics were built to be application (and at times, even machine) specific. The march of time in the history of computing has produced more languages each year, each one with its own benefits, each one championed (at least by its inventors) as the best one for the domain for which it was designed.

Each of these languages abstracts the concepts of a semantic domain into a programmable ontology set and syntax, and provides a semantic translator that gives meaning to the "sentences" of the language—eventually turning an abstract program into specific execution on a machine [2]. This can happen either through the gen-

eration of a configuration file, or perhaps the generation of compilable code. Regardless of the target artifact's form, each generated artifact is in the language of some semantic domain that has meaning which can be interpreted by a machine somehow.

The creation of the semantic translator—traditionally called the compiler in textual programming—is the most tedious task in the development of the language. This is more obvious when semantic translator creation is contrasted with modeling language creation, where metamodeling languages allow the rapid design and generation of modeling languages. This semantic translator faces the same requirements and hardships of the original FORTRAN compiler—namely that its generated output should be *correct*, in the sense that it means the same thing in the semantic domain as the language claims it did in language form.

This paper examines the metrics by which *correctness* should be defined for transformed models between two very similar languages. Further, it also examines the applicability of *modeler's intent* versus *model's intent* when determining which should be applied in the semantic domain. In order to do this, the paper examines the possibility of ambiguous PIM specifications, and how—if they exist—these specifications could be transformed into unambiguous platform specific models.

Determination of Proper Intent

Charles Simonyi announced in 1999 that "The Future Is Intentional" [5]. Intentional Programming (IP) [6][7] is an example of a growing trend to build software solutions *by* design rather than *referring to* the design. All programming languages are used to encode of the intention of the programmer (hence the original name of programs, "codes"). Sometimes the intention is obfuscated in difficult to understand syntax (in the case of languages like LISP) or in optimized behavior obtained by low-level instruction (e.g., pointer arithmetic in C). Domain-specific modeling attempts to circumvent the details of implementation and focuses instead on the details of the design—depending on the semantic translator for optimization and guarantee of behavior. In other words, domain-specific modeling tries to merge the concepts of how a system *should* behave, and how it *would* behave.

There are two driving forces behind model transformation: changes to the paradigm (the tuple of syntax, constraints, ontology, and semantic translator) and changes to the semantic domain. These are explained below, and abbreviated in Table 1.

Table 1 Paradigm (α) changes and semantic domain (Ω) changes, with actions to be taken

Case	Action
(α == α′) and (Ω != Ω′)	The intention of the modeler should be preserved.
(α != α′) and (Ω == Ω′)	The intention of the models should be preserved
(α != α′) and (Ω != Ω′)	The intention of the modeler and the models should be taken into consideration, but will both be revised

If changes are made to the semantic domain Ω only, and the paradigm α is preserved, then the intent of the modeler should be preserved (i.e., the domain models will have the same meaning in the semantic domain, but after modifications to account for changes in the semantic domain). An example of this is the evolution of a model database that uses English measurement units to one that uses SI measurement units. Although no portion of the paradigm changes, the values of all unit-

based attributes will not be correct in the semantic domain any longer, and should be modified.

If the changes are made to the paradigm α only, and the semantic domain Ω is preserved, then the intent of the models should be preserved (i.e., the domain models will have a different meaning in the semantic domain after modifications to account for changes to the CBS domain). An example of this is where the types of objects of a paradigm change, and they now have a different meaning in the semantic domain.

If changes are made to the paradigm and to the semantic domain, then some combination of these changes will be used. At this point, the modeler creating the domain evolution specification is actually assuming the role of domain modeler and revising the intent both of the modeler, and of the models, to reflect the new system.

Intention Types: Model vs. Modeler

As a kind of intentional programming, domain-specific modeling provides an interface that is most like the design of the final system, and transforms that design into the semantic domain through a translator or *interpreter*. The abstractions of a well-designed domain-specific language are convenient for expressing the *existence* of object instances in a particular domain. The structure and behavior of domain objects—traditionally specified in the design, and then encoded into the semantic domain—can be immediately translated into the semantic domain if a semantic translator exists for the modeling domain. In this way, the intention of the system is specified by its existence, rather than encoded into a domain-independent language.

Modeling the system using a domain-specific language should be simple, due to the fact that the domain-specific language should provide domain concepts as language primitives, and that it should be designed to be correct by construction—meaning that if the system is modeled as it exists, then the generated output of the domain model (the name given to the model of the system) will be an accurate reflection of the system. However there are two different types of intentions encoded into any given domain model: that of the modeler, and that of the model.

The intent of the modeler should not be different from the intent of the model, when the domain model is examined in the context of its metamodel (i.e., the domain). However, once the domain model it is applied to another metamodel (in another form, of course) then these two intentions are not necessarily in accord—i.e., the formal model "puns" become evident.

The *modeler's intent* is the encoding of a system as it exists, using the syntax, ontology, static semantics, and knowledge of the semantic translator of some (domain-specific) language. The modeler's intent is not to write a program, but instead represent the system *by construction*. The modeler uses the appropriate portions of the paradigm in which the domain-specific language is encoded to properly construct the domain model of the system.

The *model's intent* is the semantic representation of a system encoded with the inherent syntax, ontology, and static semantics of a (domain-specific) language. This semantic representation is achieved by the semantic translator especially. The intent of the models is to encode the system into the semantic domain (rather than the domain of the domain-specific language) and it uses the paradigm of the semantic domain rather than the paradigm in which it plays the part of semantic translator.

These wordy descriptions can be differentiated in the following terse statement: the intent of the modeler is to represent the reality of system existence, and the intent of the model is to represent the meaning of domain model existence. As long as the system and the domain model are encoded in the same paradigm there is no difference between them. Outside the context of the original paradigm, however, transformations according to one (or the other) of the intentions should be made to guarantee correctness by construction.

Modeling definitions

The domain of modeling could very easily be summed up as the formal development of software. Metamodels are used to describe types of models, and models are created *according to* metamodels (meaning that they satisfy the syntax and constraints set forth by those metamodels). Metamodels are the basis on which model instances are determined to be syntactically correct, and as such play an important role in the model transformation process.

Metamodeling is the formal definition of the modeling concepts that may be used to define systems within a domain. Modeling concepts are not only the actual domain concepts (e.g., processes in a signal processing domain, or assembly lines in a factory domain) but also standard modeling abstractions—patterns that provide a prototypical solution to a modeling problem—directly supported by the tools. Many such modeling abstractions exist in engineering but are often focused on a particular solution space or sub-domain. A precept of metamodeling is the existence of a core set of fundamental modeling abstractions that (as a set of archetypes) is adequate to express the design concepts, notions, and artifacts used across all semantic domains. This set is generally accepted in the UML [3] community to be made up of *types*, *associations*, *generalizations*, and *constraints*.

Using these four basic archetypal modeling concepts, metamodels are created. Members of the set of metamodeling archetypes (that is, the ontology of the metametamodel) are instantiated in a metamodel to create an instance of an archetypal formalism. For example, the archetypal formalism of *containment* can be instantiated to immediately define a relationship between two types. Metamodeling provides a rapid way in which to specify the abstract syntax of a *domain-specific modeling language*. The domain-specific modeling language is in turn used to specify the structure and behavior of domain applications [4].

In the scenario of model transformations, it is required to consider each modeling language as being domain-specific. Without the notion of a domain in which these models are correct by construction, then no translation could be made that is semantically correct. Even metamodels have their own domain, which is the domain of domain-specific language specification, and platform-independent models have their own domain, which is the specification of abstract concepts.

Formal modeling: unambiguous?

One claim of formal modeling is that it removes the possibility of an ambiguous specification. Especially when compared to English language specifications, formal models are more concise, and (when specified using the appropriate domain modeling language) unambiguous. However, it is incorrect to presume that an unambiguous specification in the domain modeling language can be immediately transformed

into an unambiguous specification in any other domain modeling language for two reasons: (1) the other domain modeling language may not be rich enough, and (2) a kind of platform-independent "pun", which is a specification that can be interpreted only one way in the PSM language, but multiple ways when transforming to the PIM.

We can immediately dismiss the first reason, because we must acknowledge that if a translation from a rich language into a less-descriptive language is undertaken, that it is done with the knowledge of the designer of the transformation. That designer then either knows that the translation is not "exact", or is performing a partial translation (e.g., transforming Simulink models into a PowerPoint document for presentation purposes only—not simulation).

The second reason is more subtle. Although the English language is abominable when it comes to unambiguous specification, it can create humorous examples, one of which is the following pun: "Time flies like an arrow, but fruit flies like a banana." This is a double pun, on [fruit] *flies* (as in a piece of fruit flying across the room, versus more than one "fruit fly" insect) and *like* (as in a simile, versus an expression of approval or attraction). This pun does not make sense in a language such as French, where the noun for housefly does not coincide with the verb for flying, and the word for approval is not similar to the word for similarity.

While this example may seem to have nothing to do with formal modeling, let us consider as an example the Simulink/Stateflow visual description language [9]. This language is used to create Statecharts [10], and these Statechart models may be simulated to examine their behavior. Statecharts specify that nondeterministic systems can exist through multiply-enabled transitions from an active state, and that the execution of a nondeterministic transition is not guaranteed by the Statechart semantics. However, the simulator provided by Simulink/Stateflow can (and does) guarantee the execution semantics of multiply-enabled transitions from an active state—specifically that the first clockwise enabled state from the 12 o'clock position of the current active state is chosen for execution.

Now, when creating a Simulink/Stateflow diagram using the visual language provided by Mathworks, two "meanings" are attached to every transition/state combination. First, that this transition governs whether the active is active; second, that the clockwise relation of this state to all other states in which it plays an associative role governs the "tiebreaker" status of the state. In other words, there is the *explicit* placement of the state and the *implicit* meaning of that state's placement.

If models built in Simulink/Stateflow are translated into another language, say, Carnegie Mellon's SMV [11] (a model verification tool), these *implicit* semantics must be considered when performing the transformation. The "pun" here is that the Simulink/Stateflow language uses the same "words" (here, states in a statechart) to mean two different things at the same time—whereas transformation to another language with a different way to guarantee execution may require additional specifications (i.e., more "words") to achieve the same meaning that was present in the Simulink/Stateflow models.

This example may seem contrived, but it is a hard problem that automotive companies (who have used Simulink/Stateflow to check correctness of hardware designs) are anxious to solve [12]. Until there is a solution, "real-world" users cannot be sure that any models they create can be verified by external solvers. After all, if the

models do not mean *exactly the same thing* in the verification language, then there is no guarantee that the results of the verification can be trusted.

Examples

Let us now examine two particular cases in which this difference between the intent of the model and of the modeler is shown.

Synchronous versus asynchronous I/O

Data inside computer registers either becomes overwritten, or written out to some other memory storage location (e.g., cache). In order to transfer this data, a chip designer may choose between two major types of input/output timing mechanisms: synchronous, and asynchronous (see [8] for more details). Briefly, synchronous I/O is governed by a clock, whereas asynchronous I/O is governed by handshake protocols. A consequence of this difference is that the actual time to execute synchronous and asynchronous transfers is different, and the machine will be in different states at different times.

Now, consider a chip designed, and implemented, using synchronous I/O protocols. The synchronous designs are typically used for register to register transfers, where a global clock is available, and transfer times are considered to be small. However, the model of the synchronous design may be useful when creating an asynchronous design for use in register to cache, or cache to memory data transfer.

When transforming the synchronous design model to be an asynchronous design, it is the *semantics* of the model of computation that governs how the structural model should be interpreted. The model transformer should *not* require that the asynchronous design behave exactly as the synchronous design—in fact, the whole purpose of choosing an asynchronous design was to modify the behavior to be appropriate for the new domain. When the design was created by the modeler, it was created to transfer data between memory locations, and it does this regardless of the model of computation chosen to implement the data transfer.

This is an example of the *modeler's intent* prevailing over the *model's intent*. In this case, the existence of the system in reality took precedence over the semantic meaning of the domain model of the system.

Language syntax differences

The domain-specific nature of many languages can be tedious and confusing to programmers unfamiliar with the domain. For instance, many legacy configuration files are optimized for minimal storage, and occasionally purposefully obfuscated or encrypted to prevent unauthorized end-user access. In many cases, the ordering of values is critical in the semantic interpretation of those values.

If a modeling paradigm uses ontological types to differentiate between certain domain concepts, then the domain modeler can instantiate those types to specify the existence of those domain concepts. Assume that such a domain model is correct by construction. Now, if the domain model is translated into another modeling paradigm, there are two possibilities, (1) that the new modeling paradigm uses the same ontology *and* attaches the same semantics to those ontological entities, or (2) that there is another way in which those semantics are encoded into the new paradigm.

For instance, all of the objects of a certain type may be required to be defined in a common repository for instantiation. Or, objects of a certain type may require additional attributes or may need to be sorted to be correctly interpreted by the new paradigms semantic translator.

This is an example of the *model's intent* prevailing over the *modeler's intent*. The modeler created the system as it existed—ostensibly with no knowledge of the new paradigm. However, the new paradigm requires more input from the modeler—not just different input. That input is provided through the meaning that is captured in the original domain models, and the translation can be performed using the semantic encodings of the original modeling paradigm.

This is similar to—yet different than—the previous example. Both examples involve a modification of the domain model to be correct in the new paradigm.

Classification of examples from the MDA Guide

Several examples from the MDA Guide [13] are given and classified in Table 2, without details of their context (details are available in the guide).

Table 2. Examples from the MDA Guide [13] are classified as to the type of intent that they should preserve. An example preserving *only* the intent of the modeler was not contained in the document.

Example number, and page	Classification
Section 3.4, Ex. 1, p. 3-2	Combination of Modeler and Model's intent (the modeling paradigm changes, and interpretation is required to determine whether the transformed models will have the same semantics as intended by the modeler)
Section 3.4.1, Ex. 1, p. 3-3	Model's intent (the paradigm changes, but no changes are made to the meaning of the model in the semantic domain)
Section 3.4.1, Ex. 2, p. 3-3	Model's intent (the paradigm changes, but no changes are made to the meaning of the model in the semantic domain)

Notice that none of these examples are classified as the intent of the modeler. While this may call into question the viability of the model/modeler intent argument, consider (a) that there are only a handful of examples in the text, frankly of limited breadth, and (b) that it is usually the case that changes to the semantic domain often prompt designers to also make changes to the paradigm. More often than not, some varying degree of combination of the model/modeler's intent must be considered. The lack of previous description of this model/modeler distinction is a testimony to the rareness of a "pure" modeler's intent, but the aim of this paper is to bring it to the attention of the community, so that in those rare cases it is not overlooked (for example, the costly Mars Polar Lander debacle by the Jet Propulsion Laboratory [14]).

Conclusions

Acceptance of these two types of catalysts requires a re-examination of the methods currently used to create model transformations. The painstaking process of creating

a model transformation between two paradigms should be undertaken only when the exact source and destination semantics are understood, and there is a clear and sufficiently proved algorithm to affect that transformation.

The methods used to create model transformations are certainly adequate; the intention of this paper is not to revoke the ability of such transforms to be specified. What this paper should do, however, is reinforce the notion that these transformations must be driven by *semantics*, where syntax transforms are the output of a semantic-driven approach. And since semantics plays such a large role in the transformation process, the designers of the transform should consider which semantics should be disregarded, and which semantics should be preserved, but are perhaps obfuscated through a "pun" in the original language.

More research is required to examine further the interrelation between model and modeler in the domain model encoding of a modeling paradigm. One thing is clear, however: that the intent of the model and that of the modeler are not identical when compared outside the domain in which they were originally created, and should be considered whenever the domain model is transformed.

References

[1] S. Lohr, *The Programmers who Created the Software Revolution—Go To*, Basic Books, 2001.

[2] D. Harel, B. Rumpe, "Modeling Languages: Syntax, Semantics, and All That Stuff. Part I: The Basic Stuff", Technical Report MCS00-16, Mathematics & Computer Science, Weizmann Institute Of Science, Rehovot, Israel.

[3] OMG Unified Modeling Language Specification, ver. 1.4, Object Management Group, et al., September 2001.

[4] G. Karsai, G. Nordstrom, A. Ledeczi, J. Sztipanovits, "Specifying Graphical Modeling Systems Using Constraint-based Metamodels", *Proceeding of the IEEE Symposium on Computer Aided Control System Design*, Anchorage, AK, 2000.

[5] Simonyi, C. "The Future Is Intentional", *IEEE Computer* Volume 32, Number 5, May 1999. pp. 56-57.

[6] W. Aitken, et al. "Transformation in Intentional Programming". *Proceedings of the Fifth International Conference on Software Reuse*, June 2-5, 1998. pp. 114-123.

[7] Röder, L. "Transformation and Visualization of Abstractions using the Intentional Programming System", Bauhaus–University, Weimar, Germany.

[8] V. Heuring, H. Jordan, *Computer Systems Design and Architecture*, Addison Wesley Longman, Inc., Menlo Park, CA, 1997. pp. 383-384.

[9] *Stateflow Users Guide.* The Mathworks, Inc., May 1997.

[10] D. Harel, "StateCharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, Vol. 8, 1987. pp. 231-274.

[11] C. Banphawatthanarak, B. Krogh, "Verification of Stateflow Diagrams Using SMV: sf2smv 2.0", Technical Report, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, February 1999.

[12] S. Sims, R. Cleaveland, K. Butts, S. Ranville, "Automated Validation of Software Models", *16th IEEE International Conference on Automated Software Engineering*, San Diego, CA, 26-29 November 2001. pp. 91-97.

[13] "MDA Guide Version 1.0", Eds. J. Miller, J. Mukerji. Document number OMG/2003-05-01. Object Management Group, 9 May 2003.

[14] Mars Polar Lander website. http://mars.jpl.nasa.gov/msp98/news/mco990930.html