Model-Based Software Design*

Neil Iscoe, Zheng-Yang Liu, Guohui Feng, Britt Yenne, Larry Van Sickle, Michael Ballantyne
EDS Research, Austin Laboratory
1601 Rio Grande, Ste. 500
Austin, Texas 78701
iscoe@austin.eds.com

Abstract

Domain-specific knowledge is required to create specifications, generate code, and understand existing systems. Our approach to automating software design is based on instantiating an application domain model with industry-specific knowledge and then using that model to achieve the operational goals of specification elicitation and verification, reverse engineering, and code generation. Although many different specification models can be created from any particular domain model, each specification model is consistent and correct with respect to the domain model.

Introduction

Although empirical field studies (Curtis, et al., 1988) have shown that application domain knowledge is critical to the success of large projects, this knowledge is rarely stored in a form which facilitates its use in creating, maintaining and evolving software systems. Capturing and managing this knowledge is a prerequisite to automating software design.

Unfortunately, domain knowledge is implicitly embodied in application code rather than explicitly recorded and maintained in separate documents. Even when documents are maintained separately from the code, the knowledge is stored in voluminous natural language documents in an informal rather than a formal manner. Although problem-specific languages are designed to remedy this situation, domain-specific knowledge is still captured in an ad hoc instead of a systematic manner. Furthermore, these languages are generally not designed in such a way that the results can be generalized or even replicated.

We are attempting to capture the domain-specific knowledge about different industry areas as a set of application domain models. Application domain models are representations of relevant aspects of application domains that can be used to achieve specific software engineering operational goals. Operational goals are always implicit in the construction of a domain model and are essential to understanding the form and content of that model. Unlike generalized knowledge representation projects such as Cyc (Lenat, 1990) that attempt to provide a basis for modeling encyclopedic knowledge, domain modeling explicitly acknowledges the commonly held view (Amarel, 1968) that representations are designed for particular purposes. These purposes–the operational goals–inherently bias any particular solution and dictate the final form of the model.

Many different operational goals and modeling projects are being pursued within the field of domain modeling (Iscoe, et al., 1991). This paper begins with an overview of the domain modeling research at EDS and our corresponding operational goals. We explain our approach to automating software design as a paradigm which facilitates the creation of multiple-specification models from a domain model. Finally, we discuss a set of issues that we have encountered in achieving our goals.

Programming-in-the-Large

EDS produces large software systems for a variety of industries such as utilities, finance, health insurance, and so on. Associated with each industry area is a rich body of knowledge which is critical to specifying and implementing the proper software system. This knowledge includes legal, financial, technical, and other expertise which is acquired by personnel over a period of years. EDS is organized into strategic business units (SBUs) so that the organization's knowledge about a particular industry can be leveraged through reuse.

At the EDS Austin research laboratory, we are building a domain modeling system which is designed to achieve the following operational goals:

- Requirements & Specifications—Eliciting, verifying, and formalizing software requirements and specifications,
- Program Transformation/Generation—Transforming a specification into efficient executable code,
- Reverse Engineering—Identifying the semantics of existing code in terms of a partial specification.

The realization of these operational goals is consistent with our long-term plan for creating knowledge-based tools to support programming-in-the-large (Barstow, 1988). The domain modeling approach provides ample opportunities for creating an automated software development paradigm.

* An earlier version of this paper was presented at the Asilomar Workshop on Change of Representation and Problem Reformulation, April 1992.

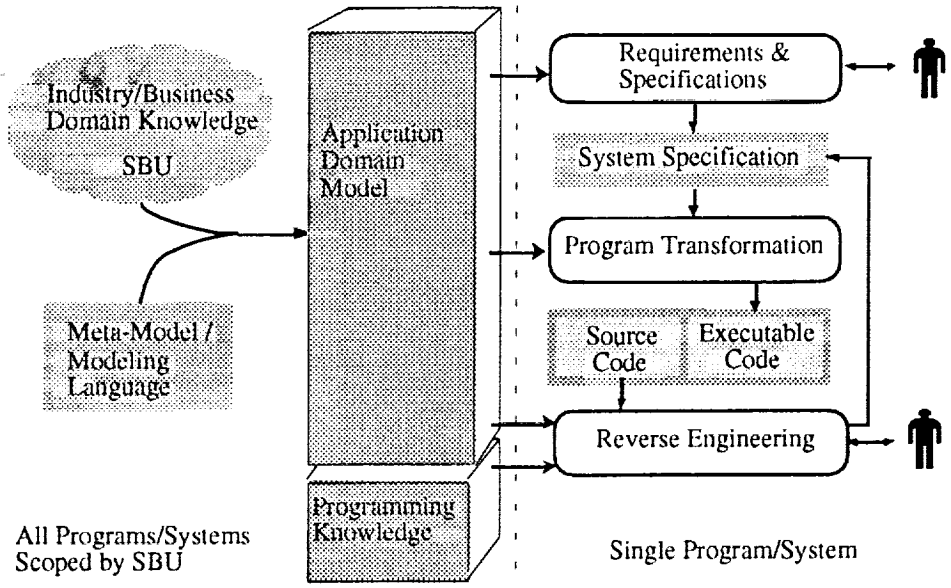

Figure 1 — Domain Modeling with Operational Goals

Figure 1 illustrates the context in which we operate. The industry knowledge for each SBU is instantiated into a domain model, which then serves as a source of knowledge for programs (the ovals) to achieve operational goals, such as reverse engineering source code or eliciting system specifications. The figure actually illustrates two different processes. The left side of figure 1 shows the process of domain model instantiation while the right side illustrates the domain model being used to produce a single specification. The *System Specification* (rectangle) illustrates a specification for a single specific system within an application domain. However, a multitude of system specifications can be created from a domain model.

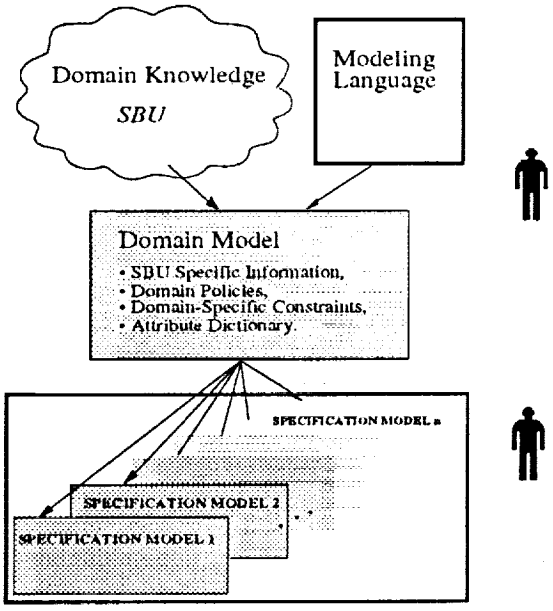

Figure 2 — Instantiating Specification Models

Figure 2 illustrates the two separate modeling tasks required by our approach. Domain experts interact with a system to represent their knowledge in terms of domain modeling constructs. Specification designers then use the system to build specification models which satisfy constraints in the domain model. In order to create a system specification, the application designer selects a set of relevant policies and constraints from the domain model that must be included and enforced in the specification model. The constraints include intra-attribute as well as inter-attribute relationships within and across classes relevant to the task at hand.

Because one of our goals is to generate executable code, we require that a particular specification model be consistent. A very large but finite number of specification models can be created which are consistent and correct with respect to a particular domain model.

Reverse Engineering

We are using reverse engineering to help instantiate both domain and specification models. Figure 1 illustrates how application domain knowledge and programming knowledge are used to extract partial specifications from source code. The box labeled "programming knowledge" currently represents knowledge of COBOL syntax, coding conventions, and program plans and structures (Van Sickle, 1992). This knowledge crosses all of the targeted application domains and is the basis of a separate code browser that operates independently of the operation shown in Figure 1.

We are also attempting to mechanically pre-instantiate domain models by using the data gathered from the applications of an EDS entity-relationship-based CASE tool that is used by SBUs for data modeling and code generation. By analyzing data models, we have access to tens of thousands of specific entities, relationships, and

constraints which have been used to specify programs and are useful for partially instantiating domain models.

Modeling Considerations

Models are inevitably abstractions of reality that capture information to achieve specific goals. A domain model determines the items of interest that exist in the world and sanctions the types of inferences allowed [Liu and Farley, 1990; Davis, 1991]. A model is the result of conscious decisions about what to describe and what to ignore. No model is complete or correct in the sense that it is applicable to all tasks.

Domain models in our system are structured to represent the type of information that is used within EDS SBUs to achieve our operational goals. Although EDS serves a wide range of industries, we are not attempting to model real-time or other application areas which diverge from standard business transaction processing. A general issue of interest in this research is the extent to which any particular representation/model can be mutated to hold different types of information for different tasks while still effectively achieving the original operational goals.

One requirement for our models is that they be consistent. Domain and specification model consistency is maintained by a specialized theorem prover. The theorem prover, *STR+VE*, is an upgraded version of the prover presented in (Bledsoe, 1980) for proofs of theorems in general inequalities. A TMS is being constructed to interface between the modeling system and the theorem prover.

Dynamic Knowledge Structure

The remainder of this paper presents one aspect of domain model representation and gives a glimpse of the relationship between specification and domain models and the organization of domain models.

While most would agree that hierarchical organizational strategies provide a reasonable way to structure knowledge within complex domains, the creation of a hierarchical structure, like any type of representational scheme, imposes a particular view of the world. Unfortunately, there is no particular view that is optimal for every application. Although the programs within a particular application share the same legal, physical, and economic constraints, the construction of any particular specification model depends upon a set of policy decisions that determine how cases are handled. Furthermore, *software in the large* systems are continually changing in such a manner that the concept of a static hierarchy is insufficient to capture the process of system evolution.

Consider software systems that manage the payment of health insurance claims. Although conceptually simple, these systems handle hundreds of thousands of different cases. One way to represent these cases is to enumerate the leaf nodes of the hierarchies created by the appropriate partitioning of attributes such as gender, age, family_status, previous_condition, employment, deductibles, copayments, prognosis, and so on. Unfortunately, the tree structure created by case expansion not only obscures relevant and interesting cases, but is also a monolithic structure. A paradox of object-oriented approaches is that well-adapted structures are not adaptable to new situations.

Because of the combinatorial explosion of the leaf nodes, it makes sense to handle the cases at as high a level as possible. Term subsumption systems such as CLASSIC (Borgida, et al., 1989) automate this process by determining the place in a hierarchy in which terms are subsumed. But subsumption systems assume a single structure in which all sub-models can belong. In the case of applications such as health insurance, individual modules may have different hierarchical structures and still maintain the integrity and constraint rules of the domain model.

Attribute Definitions

Attributes are normally considered as data values or slot fillers within a class or frame. However, the standard treatment of attributes as lists of data values with some underlying machine representation fails both to capture sufficient semantic information from the application domain and to state definitions with sufficient formality to allow semantics-related consistency checks.

Attributes are functions which define how a set of objects is mapped within a class. One type of attribute has a value set represented by a nominal scale which consists of a set of values, $\mathcal{V}(A) = \{C_1, \ldots C_n\}$.

One of the ways that the modeling process maps the world into a domain model is by creating categories in such a way that items to be categorized with respect to a particular attribute are as homogeneous as possible within a category and as heterogeneous as possible between categories. Examples of nominal scales abound and map cleanly to the notion of enumerated type as shown below:

```
(Colors
   :type     nominal_scale
   :values   (Red Yellow Green Blue)
```

The next type of attribute is an ordinal scale—a nominal scale in which a total ordering exists among the categories. Interval and ratio scales are the more quantitative scales and add definitions of dimensions, units, and granularity.

This brief description of attribute type was included to allow the reader to understand the examples in this paper. Attributes have additional types and a number of other properties which are explained in (Iscoe, et al., 1992).

Hierarchical Decomposition

Hierarchies are a natural way to view and organize information and, at some level of abstraction, are a part of most object-oriented and knowledge representation languages. Unfortunately, the simplicity of these concepts can sometimes obscure the semantics that a model is attempting to capture. That one's needs dictate one's

ontological choice is a fundamental premise of knowledge engineering. The ability to systematically define a new set of attributes by partitioning the value sets of old attributes and then using these new attributes to reclassify the domain in accordance with the new requirements is an important aspect of our attribute characterization. By preserving the "ontological map" as a component of the attribute, the domain modeler can shift between the differing paradigms modeled by various classes of objects.

Attribute characterization provides a representation and systematic methodology for the partitioning of attributes that facilitates the way they are organized, subdivided, and built into hierarchies. An attribute restriction is a new attribute whose value set and set of applicable relations are subsets of the original attribute.

Creating a new attribute serves the dual purpose of creating a set of views on the old attribute as well as creating a new attribute. Often, new attributes are defined in terms of old attributes by partitioning the original value set and then equating each new attribute value with an element of the partition. As an example, an accounts receivable (AR) system may use the attribute days_to_payment whose value is the average number of days it takes for the client to pay a bill.

```
(days_to_payment:
:type          ratio_scale
:dimension     time
:unit          days
```

From the standpoint of AR applications, a more useful attribute might be:

```
(type_of_payer:
:type          Ordinal_scale
:Ordered_by    lateness_of_payment
:values  (pays_on_time  slow_pay  dead_beat))
```

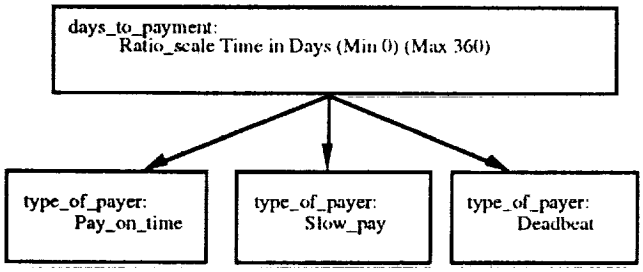

Figure 3 — Partitioning days_to_payment

This new attribute will be defined by partitioning the value set of days_to_payment by subdividing the range of values, then equating each value with one of the elements of the partition as illustrated in figure 3 and described as follows:

```
(type_of_payer
  :mapped_from  days_to_payment
    (pays_on_time   (<=30)
    (slow_pay
        (AND (> 30) (< 90)))
    (dead_beat      (>= 90))))
```

Note that the days_to_payment attribute is based on a quantitative attribute while the type_of_payer attribute is based on a qualitative attribute. In general, an attribute mapping represents a loss of information (in this example, the number of days overdue) in return for a more useful and inherently less detailed category.

Using Population Parameters

Population parameters are used to help automate the process of creating new attributes from old ones. For example, some graduate admissions committees use GRE scores to separate applicants into acceptance categories. Population parameters allow application designers to create new attributes based on restrictions to the original attribute as shown below:

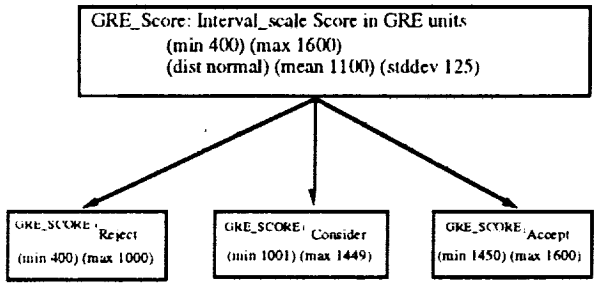

Figure 4 — Using Population Parameters to Restrict an Attribute

Figure 4 shows the GRE score as an attribute which could be attached to a student. Understanding the distribution of values within the value set of GRE scores allows application designers to create partitions in any one of a variety of ways. For example, assume that an application designer wanted to create an initial partition based on the requirement *"accept all students who score in the top x% on the GRE, consider those who score between x% and y%, and reject those who score in the bottom y%."* Given this type of requirement, the domain model contains the appropriate information to use and an algorithm to produce the correct raw score numbers to achieve such a partition.

Another way that these requirements are sometimes stated is to build a partition based on an absolute raw score. For example, a requirement like "*accept all students who score above 1450 on the GRE*" is easily displayed and modeled. Furthermore, this type of specification can be used interactively so that the designer can juggle between raw scores and percentiles until the partitions appropriate for the application domain are produced.

Domain and Specification Models

In this section we focus on relations between attributes within a single domain model class. For the purposes of this discussion we define the following attributes:

```
(Name       :type   identifier)
(Gender     :type   nominal_scale
    :values (male   female))
```

```
(Eye_color  :type   nominal_scale
    :values  (brown, blue, green))
(Benefits   :type   nominal_scale
    :values (Soc_sec, RR, none)
(Age            :type   ratio_scale
    :dimension  (time)
    :unit       (year)
    :granularity (1)
    :derived    (diff_date cur_date birth_date)
(Medicare_payment :type  ratio_scale
  :dimension  (money)
  :unit       (dollar)
  :granularity (.01))
  :popparms   ((min 1)(max 10000)(mean 225)))
(Age_m  type:      ordinal_scale
  :values          (under65  65_and_over)
  :mapped_from     age
                   (under65      (< 65))
                   (65_and_over  (>= 65)))
```

Although many other constraints exist, domain model classes can be regarded as consisting of sets of attributes which are either required or might be included within a particular domain model. These constraints are expressed as follows:

must_have(c, a,) — attribute *a* must be used in class *c* in a model.

applicable(c, a) — attribute *a* can be used in class *c* in a model depending on the choice of specification designer.

cond_must_have(c, a, cond) — attribute *a* must be used in class *c* in a model if condition *cond* evaluates to true.

cond_applicable(c, a, cond) — attribute *a* can be used in class *c* a model if condition *cond* evaluates to true.

Within any particular specification model, an attribute is simply classified as used within a class.

used(m, c, a) — within model m, attribute *a* is used in class *c* in model m.

The most straightforward relationship between a domain model and a specification model is that must_have attributes are used in all specification models and applicable attributes are selected by the specification designer. The following rules formalize the semantics of the four constraints on the use of attributes within classes listed above.

(1) must_have(c, a) $\rightarrow \forall$m used(m, c, a)

(2) applicable(c, a) $\rightarrow \exists$m used(m, c, a)

(3) (cond_applicable c a ($(p_1\ a_1\ v_1)\ldots(p_n\ a_n\ v_n)$))
$\rightarrow \forall$m, object
[(used m c a) $\rightarrow$
(used m c a_1) $\wedge \ldots \wedge$ (used m c a_n) $\wedge$
[(instance m c object) $\wedge$ (in (a object) $\mathcal{U}$(a))
$\rightarrow$ (p_1 (a_1 object) v_1) $\wedge \ldots \wedge$
(p_n (a_n object) v_n)]]

(4) (cond_must_have c a ($(p_1\ a_1\ v_1)\ldots(p_n\ a_n\ v_n)$))
$\rightarrow \forall$m,object
[(used m c a_1)$\wedge\ldots\wedge$(used m c a_n)
$\rightarrow$ (used m c a) $\wedge$
[(p_1 (a_1 object) v_1) $\wedge$
$\ldots \wedge$
(p_n (a_n object) v_n) $\wedge$ (instance m c object)
$\rightarrow$ (in (a object) $\mathcal{U}$(a))]]

For example, in a domain model, name might be required for all specification models, while eye_color could be selected only if it were appropriate for the particular specification model.

```
(person
  :must_have ((Name ())
  :applicable ((eye_color ())
  ...)
```

The application of these constraints when *cond* is vacuously true is a fairly standard feature in most modeling languages of this type. However, name and eye_color are attributes which are total functions and are not as interesting as the cases that occur when the attributes are partial functions.

Conditions for Function Evaluation

Recalling that an attribute is a function which maps objects to a particular property, *cond* can be interpreted as the condition which must be satisfied for the attribute to be a total instead of a partial function. In other words, *cond* defines the subset which is the domain of applicability of the partial function. For example for a person class medicare_payment is only applicable if age is 65 or over and benefits is none.

```
(cond_applicable  person  Medicare_payment
    ((= Age_m 65_and_over) (= Benefits none)))
```

The domain modeling system is designed so that the conditions required to establish the proper domain for an attribute are automatically maintained. These conditions are constrained in such a way that tractability is maintained and are of the form $((p_1\ a_1\ v_1)\ldots\ (p_n\ a_n\ v_n))$, where p_i is the name of a predicate, a_i is the name of an attribute, and v_i is a value of the attribute.

A user can create a specification model with any particular class hierarchy as long as the domain policies and constraints are satisfied.

We are currently experimenting with ways to capture and verify domain modeling constraints by presenting redundant information in a variety of ways. We believe that many of the specification problems in large systems are created when value set changes cause a single case to be changed but fail to correct cases that were identified from a previous inference.

For example, if we assume that Medicare_payment is only applicable if age is 65 or over and benefits is none, the system can infer that Medicare_payment cannot apply to a person who is younger than 65.

In fact, assume
(cond_applicable person Medicare_payment
((= Age_m 65_and_over) (= Benefits none))),
then
∀m, object
((used m person Medicare_payment) →
(used m person Age_m)∧(used m person Benefits)∧
((instance m person object) ∧
(in (Medicare_payment object) [1 10000])
→ (= (Age_m object) 65_and_over) ∧
(= (Benefits object) none))) (5)

After Medicare_payment is used in a model, if user is trying to assign a Medicare_payment to a person who is younger than 65, using rule (5) will lead to a contradiction.

A key point is that when people are presented with value sets they automatically and unconsciously perform substitutions such as the ones listed above. This is a reasonable way to build a model until a value set changes. In large systems, value sets are frequently changed. Consequently, conclusions that were drawn by using negation to infer values become invalid. We use the applicability of conditions and the system's knowledge of value sets to attempt to provide the proper cases for the domain modeler to check when conditions change.

Discussion

In this paper, we have presented the concept of modeling application domains in order to achieve the operational goals of program specification, code generation, and reverse engineering. The main concept is that multiple specification models can be created that are consistent and "correct" with respect to a domain model. Domain models represent information about a particular industry area. Specification models represent information about a particular system.

The middle oval on the right side of figure 1 represents the process of code generation through program transformation. Given a specification model, executable code can be generated by performing a series of correctness-preserving transformations on the specification. The goal of this part of the project, which is not yet active, is to produce efficient code that satisfies the original specification.

Domain and specification models are constructed by using a graphical interface to interactively create a set of rules based on attribute value set partitions and the preceding axioms. The system is being implemented using Motif GUI on SPARC workstations. Although it is currently operating in a single user mode, it is being designed to be accessed simultaneously by multiple domain modelers. We are also trying to accelerate the knowledge capture process by reverse engineering data models that have been captured by an existing EDS case tool and instantiating them into the appropriate domain models.

Acknowledgments

We wish to thank Betty Milstead and Raman Rajagopalan for their comments on earlier drafts of this paper.

References

Amarel, S. 1968. "On Representations of Problems of Reasoning About Actions," in *Machine Intelligence 3*, D. Michie, Ed., American Elsevier, New York, pp. 131-171.

Barstow, D. 1985. "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1321-1336.

Barstow, D. 1988. "Artificial Intelligence and Software Engineering," in Shrobe, H., Ed., *Exploring Artificial Intelligence*. AAAI. Morgan Kaufmann, San Mateo, CA.

Bledsoe, W. W., and Hines, L. M. 1980. "Variable Elimination and Chaining in a Resolution-Base Prover for Inequalities," *Proceedings of the 5th Conference on Automated Deduction*, Les Arcs, France, Springer-Verlag, pp. 70-87.

Borgida, A., Brachman, R.J., McGuinness, D.L., and Resnick, L.A. 1989. "CLASSIC: A structural data model for objects," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pp. 59-67.

Curtis, B., Krasner, H. and Iscoe, N. 1988. "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268-1287.

Davis, R. 1991. "Knowledge Representation: Broadening the Perspective," AAAI-91 Panel, Anaheim, CA.

Iscoe, N., Browne, J.C., Werth, J., and Liu, Z.Y. 1992. "Attributes – Building Blocks for Modeling Application Domains," Submitted to IEEE TSE.

Iscoe, N., Williams, G. and Arango, G., Eds. 1991. *Domain Modeling for Software Engineering, Proceedings of Domain-Modeling Workshop*, Austin, Texas.

Lenat, D.B. , Guha, R.V., Pittman, K., Pratt, D., and Shepherd, M. 1990. "Cyc: Toward Programs with Common Sense," *CACM*, vol. 33, no. 8, pp. 30-49.

Liu, Z.Y. and Farley, A. 1991. "Tasks, Models, Perspectives, Dimensions," *The 5th International Workshop on Qualitative Reasoning* Austin, Texas, pp. 1-12.

Van Sickle, L. 1992. "Reconstructing Data Integrity Constraints from Source Code," *Proceedings of Workshop on Artificial Intelligence and Automated Program Understanding*, Tenth National Conference on Artificial Intelligence, San Jose, CA.